Parallel Computing Toolbox™

User's Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

November 2004	Online only	New for Version 1.0 (Release 14SP1+)
March 2005	Online only	Revised for Version 1.0.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.0.2 (Release 14SP3)
November 2005	Online only	Revised for Version 2.0 (Release 14SP3+)
March 2006	Online only	Revised for Version 2.0.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.0 (Release 2006b)
March 2007	Online only	Revised for Version 3.1 (Release 2007a)
September 2007	Online only	Revised for Version 3.2 (Release 2007b)
March 2008	Online only	Revised for Version 3.3 (Release 2008a)
October 2008	Online only	Revised for Version 4.0 (Release 2008b)
March 2009	Online only	Revised for Version 4.1 (Release 2009a)
September 2009	Online only	Revised for Version 4.2 (Release 2009b)
March 2010	Online only	Revised for Version 4.3 (Release 2010a)
September 2010	Online only	Revised for Version 5.0 (Release 2010b)
April 2011	Online only	Revised for Version 5.1 (Release 2011a)
September 2011	Online only	Revised for Version 5.2 (Release 2011b)
March 2012	Online only	Revised for Version 6.0 (Release 2012a)
September 2012	Online only	Revised for Version 6.1 (Release 2012b)
March 2013	Online only	Revised for Version 6.2 (Release 2013a)
September 2013	Online only	Revised for Version 6.3 (Release 2013b)
March 2014	Online only	Revised for Version 6.4 (Release 2014a)
October 2014	Online only	Revised for Version 6.5 (Release 2014b)
March 2015	Online only	Revised for Version 6.6 (Release 2015a)
September 2015	Online only	Revised for Version 6.7 (Release 2015b)
March 2016	Online only	Revised for Version 6.8 (Release 2016a)
September 2016	Online only	Revised for Version 6.9 (Release 2016b)
March 2017	Online only	Revised for Version 6.10 (Release 2017a)
September 2017	Online only	Revised for Version 6.11 (Release 2017b)
March 2018	Online only	Revised for Version 6.12 (Release 2018a)
September 2018	Online only	Revised for Version 6.13 (Release 2018b)
March 2019	Online only	Revised for Version 7.0 (Release 2019a)
September 2019	Online only	Revised for Version 7.1 (Release 2019b)
March 2020	Online only	Revised for Version 7.2 (Release 2020a)
September 2020	Online only	Revised for Version 7.3 (Release 2020b)

Contents

Parallel for-Loops (parfor)

2

3

Single Program Multiple Data (spmd)

Math with Codistributed Arrays

4

Programming Overview

5

Program Independent Jobs

6

Program Communicating Jobs

7

Parallel Computing Toolbox Examples

9

Objects

10

Functions

11

Getting Started

Parallel Computing Toolbox Product Description

Perform parallel computations on multicore computers, GPUs, and computer clusters

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters. High-level constructs—parallel for-loops, special array types, and parallelized numerical algorithms—enable you to parallelize MATLAB® applications without CUDA or MPI programming. The toolbox lets you use parallel-enabled functions in MATLAB and other toolboxes. You can use the toolbox with Simulink® to run multiple simulations of a model in parallel. Programs and models can run in both interactive and batch modes.

The toolbox lets you use the full processing power of multicore desktops by executing applications on workers (MATLAB computational engines) that run locally. Without changing the code, you can run the same applications on clusters or clouds (using MATLAB Parallel Server™). You can also use the toolbox with MATLAB Parallel Server to execute matrix calculations that are too large to fit into the memory of a single machine.

Parallel Computing Support in MathWorks Products

Parallel Computing Toolbox provides you with tools for a local cluster of workers on your client machine. MATLAB Parallel Server software allows you to run as many MATLAB workers on a remote cluster of computers as your licensing allows.

Most MathWorks products enable you to run applications in parallel. For example, Simulink models can run simultaneously in parallel, as described in "Run Multiple Simulations" (Simulink). MATLAB Compiler™ and MATLAB Compiler SDK™ software let you build and deploy parallel applications; for example, see the "Parallel Computing" section of MATLAB Compiler "Standalone Applications" (MATLAB Compiler).

Several MathWorks products now offer built-in support for the parallel computing products, without requiring extra coding. For the current list of these products and their parallel functionality, see Parallel Computing Support in MATLAB and Simulink Products.

Create and Use Distributed Arrays

In this section...
"Creating Distributed Arrays" on page 1-4
"Creating Codistributed Arrays" on page 1-5

If your data is currently in the memory of your local machine, you can use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool. `Distributed` arrays use the combined memory of multiple workers in a parallel pool to store the elements of an array. For alternative ways of partitioning data, see "Distributing Arrays to Parallel Workers" on page 3-10. You operate on the entire array as a single entity, however, workers operate only on their part of the array, and automatically transfer data between themselves when necessary. You can use `distributed` arrays to scale up your big data computation. Consider `distributed` arrays when you have access to a cluster, as you can combine the memory of multiple machines in your cluster.

A `distributed` array is a single variable, split over multiple workers in your parallel pool. You can work with this variable as one single entity, without having to worry about its distributed nature. Explore the functionalities available for `distributed` arrays in the Parallel Computing Toolbox: "Run MATLAB Functions with Distributed Arrays" on page 4-18.

When you create a `distributed` array, you cannot control the details of the distribution. On the other hand, `codistributed` arrays allow you to control all aspects of distribution, including dimensions and partitions. In the following, you learn how to create both `distributed` and `codistributed` arrays.

Creating Distributed Arrays

You can create a distributed array in different ways:

- Use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool.

- You can directly construct a distributed array on the workers. You do not need to first create the array in the client, so that client workspace memory requirements are reduced. The functions available include `eye(___,'distributed')`, `rand(___,'distributed')`, etc. For a full list, see the `distributed` object reference page.

- Create a `codistributed` array inside an `spmd` statement, see "Single Program Multiple Data (spmd)" on page 1-12. Then access it as a `distributed` array outside the `spmd` statement. This lets you use distribution schemes other than the default.

In this example, you create an array in the client workspace, then turn it into a distributed array:

```
parpool('local',4)   % Create pool
A = magic(4);         % Create magic 4-by-4 matrix
B = distributed(A);   % Distribute to the workers
B                     % View results in client.
whos                  % B is a distributed array here.
delete(gcp)           % Stop pool
```

You have createdB as a `distributed` array, split over the workers in your parallel pool. This is shown in the figure.

Creating Codistributed Arrays

Unlike `distributed` arrays, `codistributed` arrays allow you to control all aspects of distribution, including dimensions and partitions. You can create a `codistributed` array in different ways:

- "Partitioning a Larger Array" on page 4-6 — Start with a large array that is replicated on all workers, and partition it so that the pieces are distributed across the workers. This is most useful when you have sufficient memory to store the initial replicated array.

- "Building from Smaller Arrays" on page 4-6 — Start with smaller replicated arrays stored on each worker, and combine them so that each array becomes a segment of a larger codistributed array. This method reduces memory requirements as it lets you build a codistributed array from smaller pieces.

- "Using MATLAB Constructor Functions" on page 4-7 — Use any of the MATLAB constructor functions like `rand` or `zeros` with a codistributor object argument. These functions offer a quick means of constructing a codistributed array of any size in just one step.

In this example, you create a `codistributed` array inside an `spmd` statement, using a nondefault distribution scheme. First, define 1-D distribution along the third dimension, with 4 parts on worker 1, and 12 parts on worker 2. Then create a 3-by-3-by-16 array of zeros.

```
parpool('local',2) % Create pool
spmd
    codist = codistributor1d(3,[4,12]);
    Z = zeros(3,3,16,codist);
    Z = Z + labindex;
end
Z           % View results in client.
whos        % Z is a distributed array here.
delete(gcp) % Stop pool
```

For more details on codistributed arrays, see "Working with Codistributed Arrays" on page 4-4.

See Also

Related Examples

- "Distributing Arrays to Parallel Workers" on page 3-10
- "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46
- "Single Program Multiple Data (spmd)" on page 1-12

Determine Product Installation and Versions

To determine if Parallel Computing Toolbox software is installed on your system, type this command at the MATLAB prompt.

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

If you want to run your applications on a cluster, see your system administrator to verify that the version of Parallel Computing Toolbox you are using is the same as the version of MATLAB Parallel Server installed on your cluster.

Interactively Run a Loop in Parallel Using parfor

In this example, you start with a slow `for`-loop, and you speed up the calculation using a `parfor`-loop instead. `parfor` splits the execution of `for`-loop iterations over the workers in a parallel pool.

This example calculates the spectral radius of a matrix and converts a `for`-loop into a `parfor`-loop. Find out how to measure the resulting speedup.

1 In the MATLAB Editor, enter the following `for`-loop. Add `tic` and `toc` to measure the time elapsed.

```
tic
n = 200;
A = 500;
a = zeros(n);
for i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

2 Run the script, and note the elapsed time.

```
Elapsed time is 31.935373 seconds.
```

3 In the script, replace the `for`-loop with a `parfor`-loop.

```
tic
n = 200;
A = 500;
a = zeros(n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

4 Run the new script, and run it again. Note that the first run is slower than the second run, because the parallel pool takes some time to start and make the code available to the workers. Note the elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
...
Elapsed time is 10.760068 seconds.
```

The `parfor` run on four workers is about three times faster than the corresponding `for`-loop run. The speed-up is smaller than the ideal speed-up of a factor of four on four workers. This is due to parallel overhead, including the time required to transfer data from the client to the workers and back. This example shows a good speed-up with relatively small parallel overhead, and benefits from conversion into a `parfor`-loop. Not all `for`-loop iterations can be turned into faster `parfor`-loops. To learn more, see "Decide When to Use parfor" on page 2-2.

One key requirement for using `parfor`-loops is that the individual iterations must be independent. Independent problems suitable for `parfor` processing include Monte Carlo simulations and parameter sweeps. For next steps, see "Convert for-Loops Into parfor-Loops" on page 2-6.

In this example, you managed to speed up the calculation by converting the `for`-loop into a `parfor`-loop on four workers. You might reduce the elapsed time further by increasing the number of workers in your parallel pool, see "Scale Up parfor-Loops to Cluster and Cloud" on page 2-20.

You can modify your cluster profiles to control how many workers run your loops, and whether the workers are local or on a cluster. For more information on profiles, see "Discover Clusters and Use Cluster Profiles" on page 5-11.

Modify your parallel preferences to control whether a parallel pool is created automatically, and how long it remains available before timing out. For more information on preferences, see "Specify Your Parallel Preferences" on page 5-9.

You can run Simulink models in parallel with the `parsim` command instead of using `parfor`-loops. For more information and examples of using Simulink in parallel, see "Run Multiple Simulations" (Simulink).

See Also
`parfor` | `parpool` | `tic` | `toc`

More About
- "Decide When to Use parfor" on page 2-2
- "Convert for-Loops Into parfor-Loops" on page 2-6
- "Scale Up parfor-Loops to Cluster and Cloud" on page 2-20

Run Batch Parallel Jobs

Run a Batch Job

To offload work from your MATLAB session to run in the background in another session, you can use the `batch` command inside a script.

1 To create the script, type:

```
edit mywave
```

2 In the MATLAB Editor, create a `for`-loop:

```
for i = 1:1024
  A(i) = sin(i*2*pi/1024);
end
```

3 Save the file and close the Editor.

4 Use the `batch` command in the MATLAB Command Window to run your script on a separate MATLAB worker:

```
job = batch('mywave')
```

5 `batch` does not block MATLAB and you can continue working while computations take place. If you need to block MATLAB until the job finishes, use the `wait` function on the job object.

```
wait(job)
```

6 After the job finishes, you can retrieve and view its results. The `load` command transfers variables created on the worker to the client workspace, where you can view the results:

```
load(job,'A')
plot(A)
```

7 When the job is complete, permanently delete its data and remove its reference from the workspace:

```
delete(job)
clear job
```

`batch` runs your code on a local worker or a cluster worker, but does not require a parallel pool.

You can use `batch` to run either scripts or functions. For more details, see the `batch` reference page.

Run a Batch Job with a Parallel Pool

You can combine the abilities to offload a job and run a loop in a parallel pool. This example combines the two to create a simple batch `parfor`-loop.

1 To create a script, type:

```
edit mywave
```

2 In the MATLAB Editor, create a `parfor`-loop:

```
parfor i = 1:1024
  A(i) = sin(i*2*pi/1024);
end
```

3   Save the file and close the Editor.

4   Run the script in MATLAB with the `batch` command. Indicate that the script should use a parallel pool for the loop:

```
job = batch('mywave','Pool',3)
```

This command specifies that three workers (in addition to the one running the batch script) are to evaluate the loop iterations. Therefore, this example uses a total of four local workers, including the one worker running the batch script. Altogether, there are five MATLAB sessions involved, as shown in the following diagram.

5   To view the results:

```
wait(job)
load(job,'A')
plot(A)
```

The results look the same as before, however, there are two important differences in execution:

- The work of defining the `parfor`-loop and accumulating its results are offloaded to another MATLAB session by `batch`.

- The loop iterations are distributed from one MATLAB worker to another set of workers running simultaneously (`'Pool'` and `parfor`), so the loop might run faster than having only one worker execute it.

6   When the job is complete, permanently delete its data and remove its reference from the workspace:

```
delete(job)
clear job
```

Run Script as Batch Job from the Current Folder Browser

From the Current Folder browser, you can run a MATLAB script as a batch job by browsing to the file's folder, right-clicking the file, and selecting **Run Script as Batch Job**. The batch job runs on the cluster identified by the default cluster profile. The following figure shows the menu option to run the script file `script1.m`:
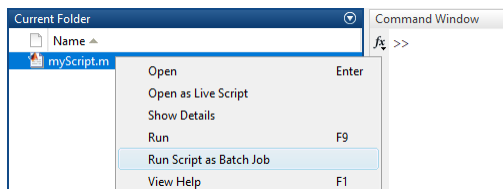
Running a script as a batch from the browser uses only one worker from the cluster. So even if the script contains a `parfor` loop or `spmd` block, it does not open an additional pool of workers on the cluster. These code blocks execute on the single worker used for the batch job. If your batch script requires opening an additional pool of workers, you can run it from the command line, as described in "Run a Batch Job with a Parallel Pool" on page 1-9.

When you run a batch job from the browser, this also opens the Job Monitor. The Job Monitor is a tool that lets you track your job in the scheduler queue. For more information about the Job Monitor and its capabilities, see "Job Monitor" on page 5-24.

See Also
`batch`

Related Examples
- "Run Batch Job and Access Files from Workers" on page 9-34

Distribute Arrays and Run SPMD

Distributed Arrays

The workers in a parallel pool communicate with each other, so you can distribute an array among the workers. Each worker contains part of the array, and all the workers are aware of which portion of the array each worker has.

Use the `distributed` function to distribute an array among the workers:

```
M = magic(4) % a 4-by-4 magic square in the client workspace
MM = distributed(M)
```

Now `MM` is a distributed array, equivalent to `M`, and you can manipulate or access its elements in the same way as any other array.

```
M2 = 2*MM;  % M2 is also distributed, calculation performed on workers
x = M2(1,1) % x on the client is set to first element of M2
```

Single Program Multiple Data (spmd)

The single program multiple data (`spmd`) construct lets you define a block of code that runs in parallel on all the workers in a parallel pool. The `spmd` block can run on some or all the workers in the pool.

```
spmd      % By default creates pool and uses all workers
    R = rand(4);
end
```

This code creates an individual 4-by-4 matrix, R, of random numbers on each worker in the pool.

Composites

Following an `spmd` statement, in the client context, the values from the block are accessible, even though the data is actually stored on the workers. On the client, these variables are called *Composite* objects. Each element of a composite is a symbol referencing the value (data) on a worker in the pool. Note that because a variable might not be defined on every worker, a Composite might have undefined elements.

Continuing with the example from above, on the client, the Composite R has one element for each worker:

```
X = R{3};  % Set X to the value of R from worker 3.
```

The line above retrieves the data from worker 3 to assign the value of X. The following code sends data to worker 3:

```
X = X + 2;
R{3} = X; % Send the value of X from the client to worker 3.
```

If the parallel pool remains open between `spmd` statements and the same workers are used, the data on each worker persists from one `spmd` statement to another.

```
spmd
    R = R + labindex  % Use values of R from previous spmd.
end
```

A typical use for `spmd` is to run the same code on a number of workers, each of which accesses a different set of data. For example:

```
spmd
    INP = load(['somedatafile' num2str(labindex) '.mat']);
    RES = somefun(INP)
end
```

Then the values of `RES` on the workers are accessible from the client as `RES{1}` from worker 1, `RES{2}` from worker 2, etc.

There are two forms of indexing a Composite, comparable to indexing a cell array:

- `AA{n}` returns the values of `AA` from worker `n`.
- `AA(n)` returns a cell array of the content of `AA` from worker `n`.

Although data persists on the workers from one `spmd` block to another as long as the parallel pool remains open, data does not persist from one instance of a parallel pool to another. That is, if the pool is deleted and a new one created, all data from the first pool is lost.

For more information about using distributed arrays, `spmd`, and Composites, see "Distributed Arrays".

What Is Parallel Computing?

Parallel computing allows you to carry out many calculations simultaneously. Large problems can often be split into smaller ones, which are then solved at the same time.

The main reasons to consider parallel computing are to

- Save time by distributing tasks and executing these simultaneously
- Solve big data problems by distributing data
- Take advantage of your desktop computer resources and scale up to clusters and cloud computing

With Parallel Computing Toolbox, you can

- Accelerate your code using interactive parallel computing tools, such as `parfor` and `parfeval`
- Scale up your computation using interactive Big Data processing tools, such as `distributed`, `tall`, `datastore`, and `mapreduce`
- Use `gpuArray` to speed up your calculation on the GPU of your computer
- Use `batch` to offload your calculation to computer clusters or cloud computing facilities

Here are some useful Parallel Computing concepts:

- *Node:* standalone computer, containing one or more CPUs / GPUs. Nodes are networked to form a cluster or supercomputer
- *Thread:* smallest set of instructions that can be managed independently by a scheduler. On a GPU, multiprocessor or multicore system, multiple threads can be executed simultaneously (multithreading)
- *Batch*: off-load execution of a functional script to run in the background
- *Scalability*: increase in parallel speedup with the addition of more resources

What tools do MATLAB and Parallel Computing Toolbox offer?

- MATLAB workers: MATLAB computational engines that run in the background without a graphical desktop. You use functions in the Parallel Computing Toolbox to automatically divide tasks and assign them to these workers to execute the computations in parallel. You can run local workers to take advantage of all the cores in your multicore desktop computer. You can also scale up to run your workers on a cluster of machines, using the MATLAB Parallel Server. The MATLAB session you interact with is known as the MATLAB client. The client instructs the workers with parallel language functions.
- Parallel pool: a parallel pool of MATLAB workers created using `parpool` or functions with automatic parallel support. By default, parallel language functions automatically create a parallel pool for you when necessary. To learn more, see "Run Code on Parallel Pools" on page 2-55.

  For the default local profile, the default number of workers is one per physical CPU core using a single computational thread. This is because even though each physical core can have several virtual cores, the virtual cores share some resources, typically including a shared floating point unit (FPU). Most MATLAB computations use this unit because they are double-precision floating point. Restricting to one worker per physical core ensures that each worker has exclusive access to a floating point unit, which generally optimizes performance of computational code. If your code is not computationally intensive, for example, it is input/output (I/O) intensive, then consider using up to two workers per physical core. Running too many workers on too few resources may impact performance and stability of your machine.

- Speed up: Accelerate your code by running on multiple MATLAB workers or GPUs, for example, using `parfor`, `parfeval`, or `gpuArray`.
- Scale up your data: Partition your big data across multiple MATLAB workers, using tall arrays and distributed arrays. To learn more, see "Big Data Processing".
- Asynchronous processing: Use `parfeval` to execute a computing task in the background without waiting for it to complete.
- Scale up to clusters and clouds: If your computing task is too big or too slow for your local computer, you can offload your calculation to a cluster onsite or in the cloud using MATLAB Parallel Server. For more information, see "Clusters and Clouds".

See Also

Related Examples
- "Choose a Parallel Computing Solution" on page 1-16
- "Identify and Select a GPU Device" on page 8-19
- "Decide When to Use parfor" on page 2-2
- "Run Single Programs on Multiple Data Sets" on page 3-2
- "Evaluate Functions in the Background Using parfeval" on page 1-22
- "Distributing Arrays to Parallel Workers" on page 3-10
- "Run Batch Parallel Jobs" on page 1-9

Choose a Parallel Computing Solution

Process your data faster or scale up your big data computation using the capabilities of MATLAB, Parallel Computing Toolbox and MATLAB Parallel Server.

Problem	Solutions	Required Products	More Information
Do you want to process your data faster?	Profile your code.	MATLAB	"Profile Your Code to Improve Performance"
	Vectorize your code.	MATLAB	"Vectorization"
	Use automatic parallel computing support in MathWorks products.	MATLAB Parallel Computing Toolbox	"Run MATLAB Functions with Automatic Parallel Support" on page 1-19
	If you have a GPU, try `gpuArray`.	MATLAB Parallel Computing Toolbox	"Run MATLAB Functions on a GPU" on page 8-9
	Use `parfor`.	MATLAB Parallel Computing Toolbox	"Interactively Run a Loop in Parallel Using parfor" on page 1-7
Are you looking for other ways to speed up your processing?	Try `parfeval`.	MATLAB Parallel Computing Toolbox	"Evaluate Functions in the Background Using parfeval" on page 1-22
	Try `spmd`.	MATLAB Parallel Computing Toolbox	"Run Single Programs on Multiple Data Sets" on page 3-2
Do you want to scale up your big data calculation?	To work with out-of-memory data with any number of rows, use tall arrays. This workflow is well suited to data analytics and machine learning.	MATLAB	"Big Data Workflow Using Tall Arrays and Datastores" on page 5-46
	Use tall arrays in parallel on your local machine.	MATLAB Parallel Computing Toolbox	"Use Tall Arrays on a Parallel Pool" on page 5-49
	Use tall arrays in parallel on your cluster.	MATLAB Parallel Computing Toolbox MATLAB Parallel Server	"Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52
	If your data is large in multiple dimensions, use `distributed` instead. This workflow is well suited to linear algebra problems.	MATLAB Parallel Computing Toolbox MATLAB Parallel Server	"Run MATLAB Functions with Distributed Arrays" on page 4-18

Problem	Solutions	Required Products	More Information
Do you want to offload to a cluster?	Use `batch` to run your code on clusters and clouds.	MATLAB Parallel Server	"Run Batch Parallel Jobs" on page 1-9
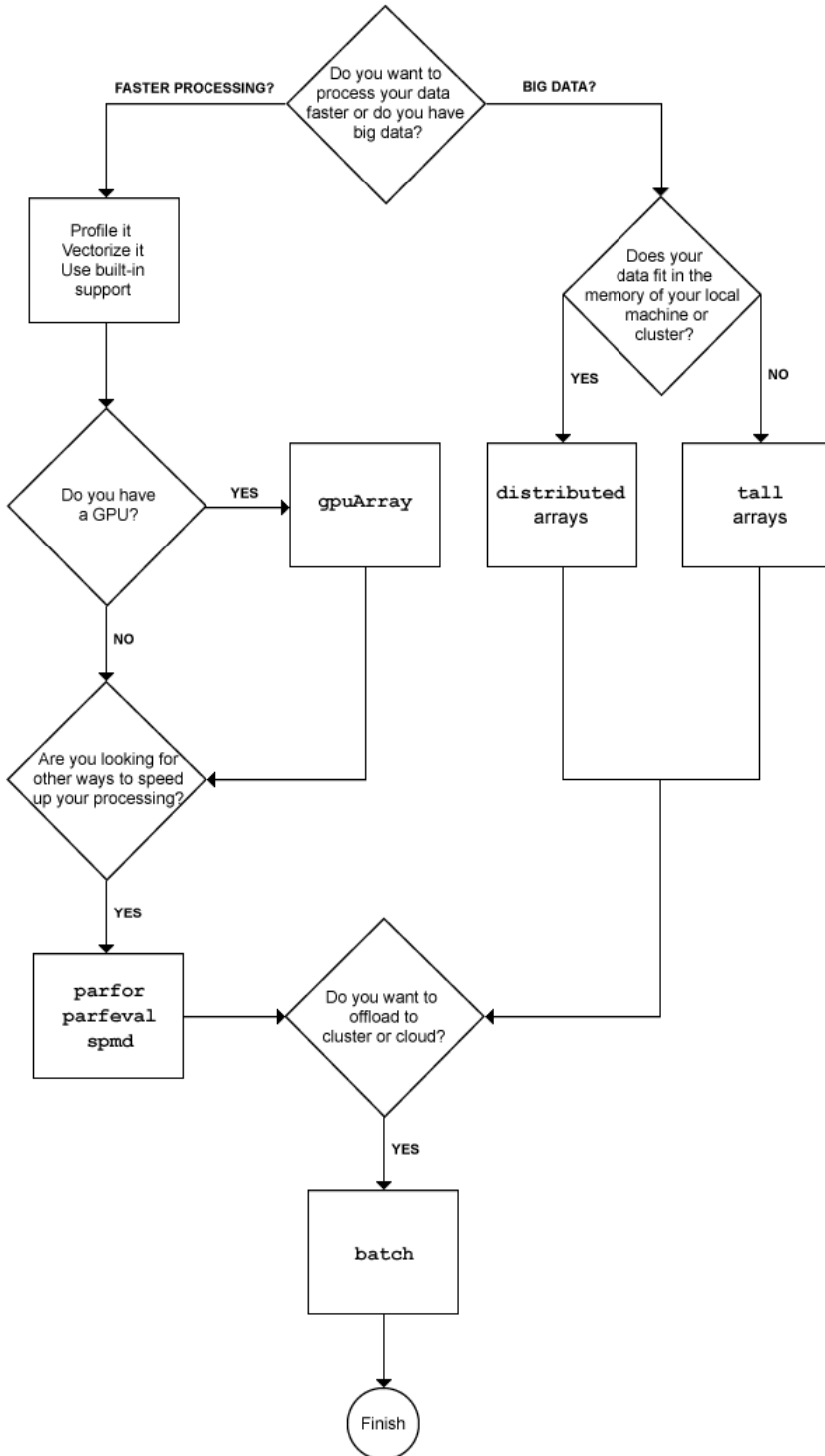
See Also

Related Examples

- "Profile Your Code to Improve Performance"
- "Vectorization"
- Built-in Parallel Computing Support
- "Identify and Select a GPU Device" on page 8-19
- "Interactively Run a Loop in Parallel Using parfor" on page 1-7
- "Evaluate Functions in the Background Using parfeval" on page 1-22
- "Run Single Programs on Multiple Data Sets" on page 3-2
- "Choose Between spmd, parfor, and parfeval" on page 3-15
- "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46
- "Use Tall Arrays on a Parallel Pool" on page 5-49
- "Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52
- "Distributing Arrays to Parallel Workers" on page 3-10
- "Run Batch Parallel Jobs" on page 1-9

Run MATLAB Functions with Automatic Parallel Support

Several MATLAB and Simulink products have a growing number of functions and features that help you take advantage of parallel computing resources without requiring any extra coding. You can enable this support by simply setting a flag or preference.

To take advantage of this functionality on your desktop, you need Parallel Computing Toolbox. Run calculations in parallel using local workers to speed up large calculations. To scale the parallel computing to larger resources such as computer clusters, you also need MATLAB Parallel Server.

- Some functions run automatically in parallel by default. For example, `parfor`, `parsim`, and `tall`.
- Many other functions run automatically in parallel if you set an option to use parallel.

When you run a function with parallel enabled, MATLAB automatically opens a parallel pool of workers. MATLAB runs the computation across the available workers.

Automatic parallel support starts a parallel pool of workers using the default cluster profile. If you have not touched your parallel preferences, the default profile is local. Control parallel behavior with the parallel preferences, including scaling up to a cluster, automatic pool creation, and preferred number of workers.

Find Automatic Parallel Support

- On function pages, find information under **Extended Capabilities**.
- You can browse supported functions from all MathWorks® products at the following link: All Functions List (Automatic Parallel Support). Alternatively, you can filter by product. On the **Help** bar, click the **Functions** tab, select a product, and select the check box **Automatic Parallel Support**. For example, for a filtered list of all Statistics and Machine Learning Toolbox™ functions with automatic parallel support, see Function List (Automatic Parallel Support). If you select a product that does not have functions with automatic parallel support, then the **Automatic Parallel Support** filter is not available.

If a function you are interested in does not include automatic parallel support, there are the alternatives:

- If you have a GPU, many MATLAB functions run automatically on a GPU. See "Run MATLAB Functions on a GPU" on page 8-9.
- Any MATLAB code inside a for-loop can be made into a parallel for loop, provided the iterations are independent. See `parfor`.
- If you are you looking for other ways to speed up your processing or to scale up your big data calculation, see "Choose a Parallel Computing Solution" on page 1-16.

See Also

Related Examples

- "Specify Your Parallel Preferences" on page 5-9
- "Run Code on Parallel Pools" on page 2-55
- "Scale Up from Desktop to Cluster" on page 9-88

More About

Run Non-Blocking Code in Parallel Using parfeval

You can execute a function on one or all parallel pool workers, without waiting for it to complete, using `parfeval` or `parfevalOnAll`. This can be useful if you want to be able to plot intermediate results. In addition, `parfeval` allows you to break out of a loop early, if you have established that your results are good enough. This may be convenient in e.g. optimization procedures. Note that this is different from using `parfor`, where you have to wait for the loop to complete.

Evaluate Functions in the Background Using parfeval

This example shows how you can use `parfeval` to evaluate a function in the background and to collect results as they become available. In this example, you submit a vector of multiple future requests in a for-loop and retrieve the individual future outputs as they become available.

```matlab
p = gcp();
% To request multiple evaluations, use a loop.
for idx = 1:10
  f(idx) = parfeval(p,@magic,1,idx); % Square size determined by idx
end
% Collect the results as they become available.
magicResults = cell(1,10);
for idx = 1:10
  % fetchNext blocks until next results are available.
  [completedIdx,value] = fetchNext(f);
  magicResults{completedIdx} = value;
  fprintf('Got result with index: %d.\n', completedIdx);
end
```

```
Got result with index: 1.
Got result with index: 2.
Got result with index: 3.
Got result with index: 4.
Got result with index: 5.
Got result with index: 6.
Got result with index: 7.
Got result with index: 8.
Got result with index: 9.
Got result with index: 10.
```

See Also

Related Examples

- "Parfeval Blackjack" on page 9-229

Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online

You can run parallel code in MATLAB Online. To access MATLAB Online, follow this link: `https://matlab.mathworks.com`.

To use Parallel Computing Toolbox functionality in MATLAB Online, you must have access to a Cloud Center cluster. You can:

- Create a cloud cluster. For more information, see "Create Cloud Cluster" on page 5-14.
- Discover an existing cluster. For more information, see "Discover Clusters" on page 5-12. You can only discover Cloud Center clusters in your MathWorks Account.
- Import a cloud cluster profile. For more information, see "Import and Export Cluster Profiles" on page 5-18. Note that if the profile is not in your MATLAB Drive™, you must upload it first. On the **Home** tab, in the **File** area, click **Upload**.

Note The local profile is not supported in MATLAB Online.

After you set up a cloud cluster, you can use parallel language functions, such as `parfor` or `parfeval`. Note that if you do not have any clusters set up, then parallel functions that require a parallel pool run in serial or throw an error.

Some differences with MATLAB Desktop include the following.

- The parallel status indicator is not visible by default. You must start a parallel pool first by using `parpool` or any function that automatically start a parallel pool.
- The Parallel Computing Toolbox Preferences options pane is not available in MATLAB Online.
- `mpiprofile viewer` is not supported in MATLAB Online.

See Also

Related Examples
- "Run Code on Parallel Pools" on page 2-55

Parallel for-Loops (parfor)

Decide When to Use parfor

parfor-Loops in MATLAB

A `parfor`-loop in MATLAB executes a series of statements in the loop body in parallel. The MATLAB client issues the `parfor` command and coordinates with MATLAB workers to execute the loop iterations in parallel on the workers in a *parallel pool*. The client sends the necessary data on which `parfor` operates to workers, where most of the computation is executed. The results are sent back to the client and assembled.

A `parfor`-loop can provide significantly better performance than its analogous `for`-loop, because several MATLAB workers can compute simultaneously on the same loop.

Each execution of the body of a `parfor`-loop is an iteration. MATLAB workers evaluate iterations in no particular order and independently of each other. Because each iteration is independent, there is no guarantee that the iterations are synchronized in any way, nor is there any need for this. If the number of workers is equal to the number of loop iterations, each worker performs one iteration of the loop. If there are more iterations than workers, some workers perform more than one loop iteration; in this case, a worker might receive multiple iterations at once to reduce communication time.

Deciding When to Use parfor

A `parfor`-loop can be useful if you have a slow `for`-loop. Consider `parfor` if you have:

- Some loop iterations that take a long time to execute. In this case, the workers can execute the long iterations simultaneously. Make sure that the number of iterations exceeds the number of workers. Otherwise, you will not use all workers available.

- Many loop iterations of a simple calculation, such as a Monte Carlo simulation or a parameter sweep. `parfor` divides the loop iterations into groups so that each worker executes some portion of the total number of iterations.

A `parfor`-loop might not be useful if you have:

- Code that has vectorized out the `for`-loops. Generally, if you want to make code run faster, first try to vectorize it. For details how to do this, see "Vectorization". Vectorizing code allows you to benefit from the built-in parallelism provided by the multithreaded nature of many of the underlying MATLAB libraries. However, if you have vectorized code and you have access only to *local* workers, then `parfor`-loops may run slower than `for`-loops. Do not devectorize code to allow for `parfor`; in general, this solution does not work well.

- Loop iterations that take a short time to execute. In this case, parallel overhead dominates your calculation.

You cannot use a `parfor`-loop when an iteration in your loop depends on the results of other iterations. Each iteration must be independent of all others. For help dealing with independent loops, see "Ensure That parfor-Loop Iterations are Independent" on page 2-9. The exception to this rule is to accumulate values in a loop using "Reduction Variables" on page 2-41.

In deciding when to use `parfor`, consider parallel overhead. Parallel overhead includes the time required for communication, coordination and data transfer — sending and receiving data — from client to workers and back. If iteration evaluations are fast, this overhead could be a significant part of the total time. Consider two different types of loop iterations:

- `for`-loops with a computationally demanding task. These loops are generally good candidates for conversion into a `parfor`-loop, because the time needed for computation dominates the time required for data transfer.
- `for`-loops with a simple computational task. These loops generally do not benefit from conversion into a `parfor`-loop, because the time needed for data transfer is significant compared with the time needed for computation.

Example of parfor With Low Parallel Overhead

In this example, you start with a computationally demanding task inside a `for`-loop. The `for`-loops are slow, and you speed up the calculation using `parfor`-loops instead. `parfor` splits the execution of `for`-loop iterations over the workers in a parallel pool.

This example calculates the spectral radius of a matrix and converts a `for`-loop into a `parfor`-loop. Find out how to measure the resulting speedup and how much data is transferred to and from the workers in the parallel pool.

1 In the MATLAB Editor, enter the following `for`-loop. Add `tic` and `toc` to measure the computation time.

```
tic
n = 200;
A = 500;
a = zeros(n);
for i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

2 Run the script, and note the elapsed time.

```
Elapsed time is 31.935373 seconds.
```

3 In the script, replace the `for`-loop with a `parfor`-loop. Add `ticBytes` and `tocBytes` to measure how much data is transferred to and from the workers in the parallel pool.

```
tic
ticBytes(gcp);
n = 200;
A = 500;
a = zeros(n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
```

```
tocBytes(gcp)
toc
```

4   Run the new script on four workers, and run it again. Note that the first run is slower than the second run, because the parallel pool takes some time to start and make the code available to the workers. Note the data transfer and elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
...
            BytesSentToWorkers     BytesReceivedFromWorkers

            _____    _____

      1      15340                   7024
      2      13328                   5712
      3      13328                   5704
      4      13328                   5728
    Total    55324                  24168
```

```
Elapsed time is 10.760068 seconds.
```

The `parfor` run on four workers is about three times faster than the corresponding `for`-loop calculation. The speed-up is smaller than the ideal speed-up of a factor of four on four workers. This is due to parallel overhead, including the time required to transfer data from the client to the workers and back. Use the `ticBytes` and `tocBytes` results to examine the amount of data transferred. Assume that the time required for data transfer is proportional to the size of the data. This approximation allows you to get an indication of the time required for data transfer, and to compare your parallel overhead with other `parfor`-loop iterations. In this example, the data transfer and parallel overhead are small in comparison with the next example.

The current example has a low parallel overhead and benefits from conversion into a `parfor`-loop. Compare this example with the simple loop iteration in the next example, see "Example of parfor With High Parallel Overhead" on page 2-4.

For another example of a `parfor`-loop with computationally demanding tasks, see "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12

Example of parfor With High Parallel Overhead

In this example, you write a loop to create a simple sine wave. Replacing the `for`-loop with a `parfor`-loop does *not* speed up your calculation. This loop does not have a lot of iterations, it does not take long to execute and you do not notice an increase in execution speed. This example has a high parallel overhead and does not benefit from conversion into a `parfor`-loop.

1   Write a loop to create a sine wave. Use `tic` and `toc` to measure the time elapsed.

```
tic
n = 1024;
A = zeros(n);
for i = 1:n
    A(i,:) = (1:n) .* sin(i*2*pi/1024);
end
toc
```

```
Elapsed time is 0.012501 seconds.
```

2 Replace the `for`-loop with a `parfor`-loop. Add `ticBytes` and `tocBytes` to measure how much data is transferred to and from the workers in the parallel pool.

```
tic
ticBytes(gcp);
n = 1024;
A = zeros(n);
parfor (i = 1:n)
    A(i,:) = (1:n) .* sin(i*2*pi/1024);
end
tocBytes(gcp)
toc
```

3 Run the script on four workers and run the code again. Note that the first run is slower than the second run, because the parallel pool takes some time to start and make the code available to the workers. Note the data transfer and elapsed time for the second run.

	BytesSentToWorkers	BytesReceivedFromWorkers
1	13176	2.0615e+06
2	15188	2.0874e+06
3	13176	2.4056e+06
4	13176	1.8567e+06
Total	54716	8.4112e+06

```
Elapsed time is 0.743855 seconds.
```

Note that the elapsed time is much smaller for the serial `for`-loop than for the `parfor`-loop on four workers. In this case, you do not benefit from turning your `for`-loop into a `parfor`-loop. The reason is that the transfer of data is much greater than in the previous example, see "Example of parfor With Low Parallel Overhead" on page 2-3. In the current example, the parallel overhead dominates the computing time. Therefore the sine wave iteration does not benefit from conversion into a `parfor`-loop.

This example illustrates why high parallel overhead calculations do not benefit from conversion into a `parfor`-loop. To learn more about speeding up your code, see "Convert for-Loops Into parfor-Loops" on page 2-6

See Also
`parfor` | `ticBytes` | `tocBytes`

Related Examples
- "Interactively Run a Loop in Parallel Using parfor" on page 1-7
- "Convert for-Loops Into parfor-Loops" on page 2-6
- "Ensure That parfor-Loop Iterations are Independent" on page 2-9
- "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12
- "Scale Up parfor-Loops to Cluster and Cloud" on page 2-20

Convert for-Loops Into parfor-Loops

In some cases, you must modify the code to convert `for`-loops to `parfor`-loops. This example shows how to diagnose and fix `parfor`-loop problems using a simple nested `for`-loop. Run this code in MATLAB and examine the results.

```
for x = 0:0.1:1
    for k = 2:10
        x(k) = x(k-1) + k;
    end
    x
end
```

To speed up the code, try to convert the `for`-loops to `parfor`-loops. Observe that this code produces errors.

```
parfor x = 0:0.1:1
    parfor k = 2:10
        x(k) = x(k-1) + k;
    end
    x
end
```

In this case you cannot simply convert the `for`-loops to `parfor`-loops without modification. To make this work, you must change the code in several places. To diagnose the problems, look for Code Analyzer messages in the MATLAB Editor.
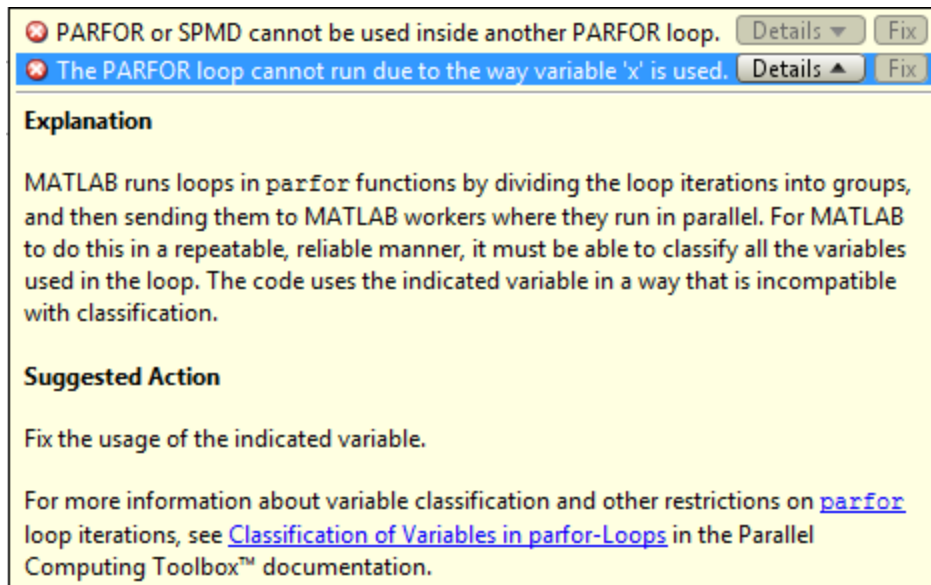
This code shows common problems when you try to convert `for`-loops to `parfor`-loops.

Noninteger loop variable

```
parfor x = 0:0.1:1
```

Nested parallel loops ⊗ `parfor y = 2:10`

```
            A(y) = A(y-1) + ...
```
Dependent loop body
```
        end
```
```
    end
```

To solve these problems, you must modify the code to use `parfor`. The body of the `parfor`-loop is executed in a parallel pool using multiple MATLAB workers in a nondeterministic order. Therefore, you have to meet these requirements for the body of the `parfor`-loop:

1 The body of the `parfor`-loop must be independent. One loop iteration cannot depend on a previous iteration, because the iterations are executed in parallel in a nondeterministic order. In the example,

```
x(k) = x(k-1) + k;
```

is not independent, and therefore you cannot use `parfor`. For next steps in dealing with independence issues, see "Ensure That parfor-Loop Iterations are Independent" on page 2-9.

2 You cannot nest a `parfor`-loop inside another `parfor`-loop. The example has two nested `for`-loops, and therefore you can replace only one `for`-loop with a `parfor`-loop. Instead, you can call a function that uses a `parfor`-loop inside the body of the other `parfor`-loop. However, such nested `parfor`-loops give you no computational benefit, because all workers are used to parallelize the outermost loop. For help dealing with nested loops, see "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12.

3 `parfor`-loop variables must be consecutive increasing integers. In the example,

```
parfor x = 0:0.1:1
```

has non-integer loop variables, and therefore you cannot use `parfor` here. You can solve this problem by changing the value of the loop variable to integer values required by the algorithm. For next steps in troubleshooting `parfor`-loop variables, see "Ensure That parfor-Loop Variables Are Consecutive Increasing Integers" on page 2-28.

4 You cannot break out of a `parfor`-loop early, as you can in a `for`-loop. Do not include a return or break statement in the body of your `parfor`-loop. Without communication, the other MATLAB instances running the loop do not know when to stop. As an alternative, consider `parfeval`.

If you still have problems converting `for`-loops to `parfor`-loops, see "Troubleshoot Variables in parfor-Loops" on page 2-28.

Tip You can profile a `parfor`-loops using `tic` and `toc` to measure the speedup compared to the corresponding `for`-loop. Use `ticBytes` and `tocBytes` to measure how much data is transferred to and from the workers in the parallel pool. For more information and examples, see "Profiling parfor-loops" on page 2-52.

See Also
parfor | ticBytes | tocBytes

Related Examples

Ensure That parfor-Loop Iterations are Independent

If you get an error when you convert `for`-loops to `parfor`-loops, ensure that your `parfor`-loop iterations are independent. `parfor`-loop iterations have *no guaranteed order*, while the iteration order in `for`-loops is *sequential*. Also `parfor`-loop iterations are performed on different MATLAB workers in the parallel pool, so that there is no sharing of information between iterations. Therefore one `parfor`-loop iteration must not depend on the result of a previous iteration. The only exception to this rule is to accumulate values in a loop using "Reduction Variables" on page 2-41.

The following example produces equivalent results, using a `for`-loop on the left and a `parfor`-loop on the right. Try the example in your MATLAB Command Window:

```
clear A                          clear A
for i = 1:8                      parfor i = 1:8
    A(i) = i;                        A(i) = i;
end                              end
A                                A

A =                              A =

    1   2   3   4   5   6   7    81   2   3   4   5   6   7   8
```

Each element of `A` is equal to its index. The `parfor`-loop works because each element is determined by the indexed loop variable only and does not depend on other variables. `for`-loops with independent tasks are ideal candidates for `parfor`-loops.

Note By default, `parfor` automatically starts a parallel pool of workers, if you have not started one already. `parfor` creates a pool using your default cluster profile, if you have set your parallel preferences accordingly.

In the example, the array elements are available in the client workspace after the `parfor`-loop, exactly as with a `for`-loop.

Now use a nonindexed variable inside the loop, or a variable whose indexing does not depend on the loop variable `i`. Try these examples, and note the values of `d` and `i` afterward:

```
clear A
d = 0; i = 0;
for i = 1:4
    d = i*2;
    A(i) = d;
end
A
d
i

A =

    2    4    6    8

d =

    8

i =

    4
```

```
clear A
d = 0; i = 0;
parfor i = 1:4
    d = i*2;
    A(i) = d;
end
A
d
i

A =

    2    4    6    8

d =

    0

i =

    0
```

Although the elements of A are the same in both examples, the value of d is not. In the for-loop, the iterations are executed sequentially, so afterward d has the value it held in the last iteration of the loop. In the parfor-loop, however, the iterations execute in parallel, so it is impossible to assign d a defined value at the end of the loop. This situation also applies to the loop variable i. Therefore, parfor-loop behavior is defined so that it does not affect the values d and i outside the loop. Their values remain the same before and after the loop. If the variables in your parfor-loop are not independent, then you might get different answers from those in the for-loop. In summary, a parfor-loop requires that each iteration be independent of the other iterations. All code that follows the parfor statement should not depend on the loop iteration sequence.

Code Analyzer can help diagnose whether the loop iterations are dependent. The code in the example shows iterations defined in terms of the previous iteration:

```
parfor k = 2:10
    x(k) = x(k-1) + k;
end
```

Look for Code Analyzer messages in the MATLAB Editor. In this case, Code Analyzer reports the dependency problem.

> ⊗ Line 2: In a PARFOR loop, variable 'x' is indexed in different ways, potentially causing dependencies between iterations.

In other cases, however, Code Analyzer is unable to mark dependencies.

For help with other common parfor problems, see "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12.

See Also
parfor

Related Examples

More About

Nested parfor and for-Loops and Other parfor Requirements

Nested parfor-Loops

You cannot use a `parfor`-loop inside another `parfor`-loop. As an example, the following nesting of `parfor`-loops is not allowed:

```
parfor i = 1:10
    parfor j = 1:5
        ...
    end
end
```

Tip You cannot nest `parfor` directly within another `parfor`-loop. A `parfor`-loop can call a function that contains a `parfor`-loop, but you do not get any additional parallelism.

Code Analyzer in the MATLAB Editor flags the use of `parfor` inside another `parfor`-loop:

⊗ PARFOR or SPMD cannot be used inside another PARFOR loop.

You cannot nest `parfor`-loops because parallelization can be performed at only one level. Therefore, choose which loop to run in parallel, and convert the other loop to a `for`-loop.

Consider the following performance issues when dealing with nested loops:

- Parallel processing incurs overhead. Generally, you should run the outer loop in parallel, because overhead only occurs once. If you run the inner loop in parallel, then each of the multiple `parfor` executions incurs an overhead. See "Convert Nested for-Loops to parfor-Loops" on page 2-13 for an example how to measure parallel overhead.
- Make sure that the number of iterations exceeds the number of workers. Otherwise, you do not use all available workers.
- Try to balance the `parfor`-loop iteration times. `parfor` tries to compensate for some load imbalance.

Tip Always run the outermost loop in parallel, because you reduce parallel overhead.

You can also use a function that uses `parfor` and embed it in a `parfor`-loop. Parallelization occurs only at the outer level. In the following example, call a function `MyFun.m` inside the outer `parfor`-loop. The inner `parfor`-loop embedded in `MyFun.m` runs sequentially, not in parallel.

```
parfor i = 1:10
    MyFun(i)
```

```
    end

function MyFun(i)
    parfor j = 1:5
        ...
    end
end
```

Tip Nested `parfor`-loops generally give you no computational benefit.

Convert Nested for-Loops to parfor-Loops

A typical use of nested loops is to step through an array using a one-loop variable to index one dimension, and a nested-loop variable to index another dimension. The basic form is:

```
X = zeros(n,m);
for a = 1:n
    for b = 1:m
        X(a,b) = fun(a,b)
    end
end
```

The following code shows a simple example. Use `tic` and `toc` to measure the computing time needed.

```
A = 100;
tic
for i = 1:100
    for j = 1:100
        a(i,j) = max(abs(eig(rand(A))));
    end
end
toc
```

```
Elapsed time is 49.376732 seconds.
```

You can parallelize either of the nested loops, but you cannot run both in parallel. The reason is that the workers in a parallel pool cannot start or access further parallel pools.

If the loop counted by `i` is converted to a `parfor`-loop, then each worker in the pool executes the nested loops using the `j` loop counter. The `j` loops themselves cannot run as a `parfor` on each worker.

Because parallel processing incurs overhead, you must choose carefully whether you want to convert either the inner or the outer `for`-loop to a `parfor`-loop. The following example shows how to measure the parallel overhead.

First convert only the *outer* `for`-loop to a `parfor`-loop. Use `tic` and `toc` to measure the computing time needed. Use `ticBytes` and `tocBytes` to measure how much data is transferred to and from the workers in the parallel pool.

Run the new code, and run it again. The first run is slower than subsequent runs, because the parallel pool takes some time to start and make the code available to the workers.

```
A = 100;
tic
```

```
ticBytes(gcp);
parfor i = 1:100
    for j = 1:100
        a(i,j) = max(abs(eig(rand(A))));
    end
end
tocBytes(gcp)
toc
```

	BytesSentToWorkers	BytesReceivedFromWorkers
1	32984	24512
2	33784	25312
3	33784	25312
4	34584	26112
Total	1.3514e+05	1.0125e+05

```
Elapsed time is 14.130674 seconds.
```

Next convert only the *inner* loop to a `parfor`-loop. Measure the time needed and data transferred as in the previous case.

```
A = 100;
tic
ticBytes(gcp);
for i = 1:100
    parfor j = 1:100
        a(i,j) = max(abs(eig(rand(A))));
    end
end
tocBytes(gcp)
toc
```

	BytesSentToWorkers	BytesReceivedFromWorkers
1	1.3496e+06	5.487e+05
2	1.3496e+06	5.4858e+05
3	1.3677e+06	5.6034e+05
4	1.3476e+06	5.4717e+05
Total	5.4144e+06	2.2048e+06

```
Elapsed time is 48.631737 seconds.
```

If you convert the *inner* loop to a `parfor`-loop, both the time and amount of data transferred are much greater than in the parallel outer loop. In this case, the elapsed time is almost the same as in the nested `for`-loop example. The speedup is smaller than running the outer loop in parallel, because you have more data transfer and thus more parallel overhead. Therefore if you execute the *inner* loop in parallel, you get no computational benefit compared to running the serial `for`-loop.

If you want to reduce parallel overhead and speed up your computation, run the outer loop in parallel.

If you convert the *inner* loop instead, then each iteration of the outer loop initiates a separate `parfor`-loop. That is, the inner loop conversion creates 100 `parfor`-loops. Each of the multiple `parfor` executions incurs overhead. If you want to reduce parallel overhead, you should run the outer loop in parallel instead, because overhead only occurs once.

Tip If you want to speed up your code, always run the outer loop in parallel, because you reduce parallel overhead.

Nested for-Loops: Requirements and Limitations

If you want to convert a nested `for`-loop to a `parfor`-loop, you must ensure that your loop variables are properly classified, see "Troubleshoot Variables in parfor-Loops" on page 2-28. If your code does not adhere to the guidelines and restrictions labeled as **Required**, you get an error. MATLAB catches some of these errors at the time it reads the code, and others when it executes the code. These errors are labeled as **Required (static)** or **Required (dynamic)** respectively.

Required (static): You must define the range of a `for`-loop nested in a `parfor`-loop by constant numbers or broadcast variables.

In the following example, the code on the left does not work because you define the upper limit of the `for`-loop by a function call. The code on the right provides a workaround by first defining a broadcast or constant variable outside the `parfor`-loop:

Invalid	Valid
``A = zeros(100, 200);
parfor i = 1:size(A, 1)
for j = 1:size(A, 2)
A(i, j) = i + j;
end
end```	```A = zeros(100, 200);
n = size(A, 2);
parfor i = 1:size(A,1)
for j = 1:n
A(i, j) = i + j;
end
end```

Required (static): The index variable for the nested `for`-loop must never be explicitly assigned other than by its `for` statement.

This restriction is required, because changing the nested `for`-loop variable in the loop body cannot guarantee that the region indexed by the `for`-loop variable is available at each worker.

The code on the left is not valid because it tries to modify the value of the nested `for`-loop variable j in the body of the loop. The code on the right provides a workaround by assigning the nested `for`-loop variable to a temporary variable t, and then updating t.

Invalid	Valid
``A = zeros(10);
parfor i = 1:10
for j = 1:10
A(i, j) = 1;
j = j+1;
end
end```	```A = zeros(10);
parfor i = 1:10
for j = 1:10
A(i, j) = 1;
t = j;
t = t + 1;
end
end```

Required (static): You cannot index or subscript a nested for-loop variable.

This restriction is required, because indexing a loop variable cannot guarantee the independence of iterations.

The example on the left is invalid because it attempts to index the nested `for`-loop variable `j`. The example on the right removes this indexing.

Invalid	Valid
``A = zeros(10);
parfor i = 1:10
for j = 1:10
j(1);
end
end```	```A = zeros(10);
parfor i = 1:10
for j = 1:10
j;
end
end```

> **Required (static)**: When using the nested `for`-loop variable for indexing a sliced array, you must use the variable in plain form, not as part of an expression.

For example, the following code on the left does not work, but the code on the right does:

Invalid	Valid
``A = zeros(4, 11);
parfor i = 1:4
for j = 1:10
A(i, j + 1) = i + j;
end
end```	```A = zeros(4, 11);
parfor i = 1:4
for j = 2:11
A(i, j) = i + j - 1;
end
end```

> **Required (static)**: If you use a nested `for`-loop to index into a sliced array, you cannot use that array elsewhere in the `parfor`-loop.

In the following example, the code on the left does not work because `A` is sliced and indexed inside the nested `for`-loop. The code on the right works because `v` is assigned to `A` outside of the nested loop:

Invalid	Valid
``A = zeros(4, 10);
parfor i = 1:4
for j = 1:10
A(i, j) = i + j;
end
disp(A(i, j))
end```	```A = zeros(4, 10);
parfor i = 1:4
v = zeros(1, 10);
for j = 1:10
v(j) = i + j;
end
disp(v(j))
A(i, :) = v;
end```

parfor-Loop Limitations

Nested Functions

The body of a `parfor`-loop cannot reference a nested function. However, it can call a nested function by a function handle. Try the following example. Note that `A(idx) = nfcn(idx)` in the `parfor`-loop does not work. You must use `feval` to invoke the `fcn` handle in the `parfor`-loop body.

```
function A = pfeg
    function out = nfcn(in)
        out = 1 + in;
```

```
        end

    fcn = @nfcn;

    parfor idx = 1:10
        A(idx) = feval(fcn, idx);
    end
end
>> pfeg
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

ans =

    2    3    4    5    6    7    8    9   10   11
```

Tip If you use function handles that refer to nested functions inside a `parfor`-loop, then the values of externally scoped variables are not synchronized among the workers.

Nested parfor-Loops

The body of a `parfor`-loop cannot contain a `parfor`-loop. For more information, see "Nested parfor-Loops" on page 2-12.

Nested spmd Statements

The body of a `parfor`-loop cannot contain an `spmd` statement, and an `spmd` statement cannot contain a `parfor`-loop. The reason is that workers cannot start or access further parallel pools.

break and return Statements

The body of a `parfor`-loop cannot contain `break` or `return` statements. Consider `parfeval` or `parfevalOnAll` instead, because you can use `cancel` on them.

Global and Persistent Variables

The body of a `parfor`-loop cannot contain `global` or `persistent` variable declarations. The reason is that these variables are not synchronized between workers. You can use `global` or `persistent` variables within functions, but their value is visible only to the worker that creates them. Instead of `global` variables, it is a better practice to use function arguments to share values.

To learn more about variable requirements, see "Troubleshoot Variables in parfor-Loops" on page 2-28.

Scripts

If a script introduces a variable, you cannot call this script from within a `parfor`-loop or `spmd` statement. The reason is that this script would cause a transparency violation. For more details, see "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49.

Anonymous Functions

You can define an anonymous function inside the body of a `parfor`-loop. However, sliced output variables inside anonymous functions are not supported. You can work around this by using a temporary variable for the sliced variable, as shown in the following example.

```
x = 1:10;
parfor i=1:10
    temp = x(i);
    anonymousFunction = @() 2*temp;
    x(i) = anonymousFunction() + i;
end
disp(x);
```

For more information on sliced variables, see "Sliced Variables" on page 2-36.

inputname Functions

Using `inputname` to return the workspace variable name corresponding to an argument number is not supported inside `parfor`-loops. The reason is that `parfor` workers do not have access to the workspace of the MATLAB desktop. To work around this, call `inputname` before `parfor`, as shown in the following example.

```
a = 'a';
myFunction(a)

function X = myFunction(a)
    name = inputname(1);

    parfor i=1:2
        X(i).(name) = i;
    end
end
```

load Functions

The syntaxes of `load` that do not assign to an output structure are not supported inside `parfor`-loops. Inside `parfor`, always assign the output of `load` to a structure.

nargin or nargout Functions

The following uses are not supported inside `parfor`-loops:

- Using `nargin` or `nargout` without a function argument
- Using `narginchk` or `nargoutchk` to validate the number of input or output arguments in a call to the function that is currently executing

The reason is that workers do not have access to the workspace of the MATLAB desktop. To work around this, call these functions before `parfor`, as shown in the following example.

```
myFunction('a','b')

function X = myFunction(a,b)
    nin = nargin;
    parfor i=1:2
        X(i) = i*nin;
    end
end
```

P-Code Scripts

You can call P-code script files from within a `parfor`-loop, but P-code scripts cannot contain a `parfor`-loop. To work around this, use a P-code function instead of a P-code script.

See Also

`parfeval` | `parfevalOnAll` | `parfor`

Related Examples

- "Decide When to Use parfor" on page 2-2
- "Convert for-Loops Into parfor-Loops" on page 2-6
- "Ensure That parfor-Loop Iterations are Independent" on page 2-9
- "Troubleshoot Variables in parfor-Loops" on page 2-28

Scale Up parfor-Loops to Cluster and Cloud

In this example, you start on your local multicore desktop and measure the time required to run a calculation, as a function of increasing numbers of workers. The test is called a *strong scaling* test. It enables you to measure the decrease in time required for the calculation if you add more workers. This dependence is known as *speedup*, and allows you to estimate the *parallel scalability* of your code. You can then decide whether it is useful to increase the number of workers in your parallel pool, and scale up to cluster and cloud computing.

1   Create the function.

```
edit MyCode
```

2   In the MATLAB Editor, enter the new parfor-loop and add tic and toc to measure the time elapsed.

```
function a = MyCode(A)
    tic
    parfor i = 1:200
        a(i) = max(abs(eig(rand(A))));
    end
    toc
end
```

3   Save the file, and close the Editor.

4   On the **Parallel > Parallel Preferences** menu, check that your **Default Cluster** is **local** (your desktop machine).

5   In the MATLAB Command Window, define a parallel pool of size 1, and run your function on one worker to calculate the elapsed time. Note the elapsed time for a single worker and shut down your parallel pool.

```
parpool(1);
a = MyCode(1000);

Elapsed time is 172.529228 seconds.

delete(gcp);
```

6   Open a new parallel pool of two workers, and run the function again.

```
parpool(2);
a = MyCode(1000);
```

Note the elapsed time; you should see that this now has decreased compared to the single worker case.

7   Try 4, 8, 12 and 16 workers. Measure the parallel scalability by plotting the elapsed time for each number of workers on a log-log scale.

The figure shows the scalability for a typical multicore desktop PC (blue circle data points). The strong scaling test shows almost linear speedup and significant parallel scalability for up to eight workers. Observe from the figure that, in this case, we do not achieve further speedup for more than eight workers. This result means that, on a local desktop machine, all cores are fully used for 8 workers. You can get a different result on your local desktop, depending on your hardware. To further speed up your parallel application, consider scaling up to cloud or cluster computing.

8 If you have exhausted your local workers, as in the previous example, you can scale up your calculation to cloud computing. Check your access to cloud computing from the **Parallel > Discover Clusters** menu.

Open a parallel pool in the cloud and run your application without changing your code.

```
parpool(16);
a = MyCode(1000);
```

Note the elapsed time for increasing numbers of cluster workers. Measure the parallel scalability by plotting the elapsed time as a function of number of workers on a log-log scale.
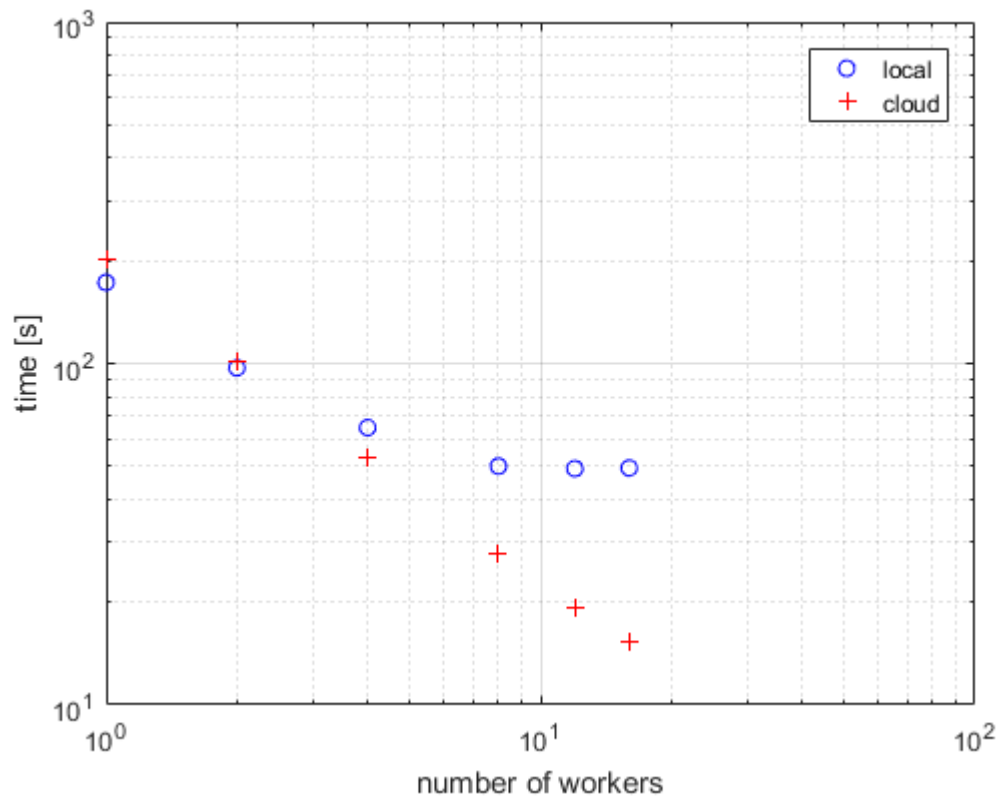
The figure shows typical performance for workers in the cloud (red plus data points). This strong scaling test shows linear speedup and 100% parallel scalability up to 16 workers in the cloud. Consider further scaling up of your calculation by increasing the number of workers in the cloud or on a compute cluster. Note that the parallel scalability can be different, depending on your hardware, for a larger number of workers and other applications.

9 If you have direct access to a cluster, you can scale up your calculation using workers on the cluster. Check your access to clusters from the **Parallel > Discover Clusters** menu. If you have an account, select **cluster**, open a parallel pool and run your application without changing your code.

```
parpool(64);
a = MyCode(1000);
```
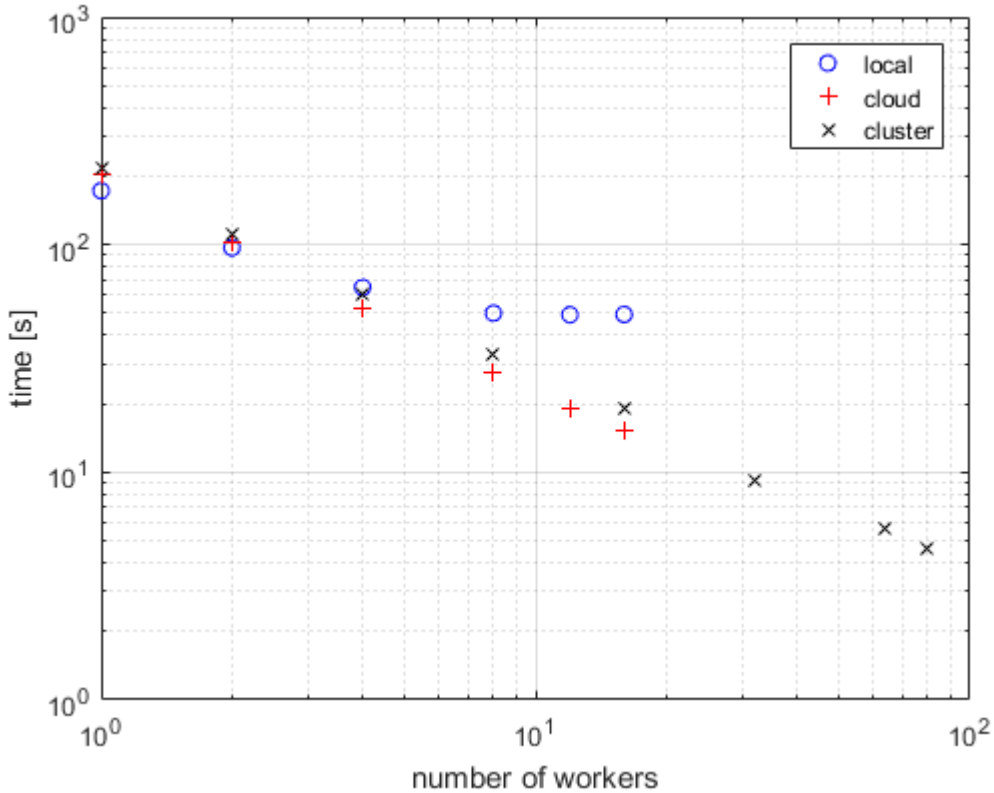
The figure shows typical strong scaling performance for workers on a cluster (black x data points). Observe that you achieve 100% parallel scalability, persisting up to at least 80 workers on the cluster. Note that this application scales linearly - the speedup is equal to the number of workers used.

This example shows a speedup equal to the number of workers. Not every task can achieve a similar speedup, see for example "Interactively Run a Loop in Parallel Using parfor" on page 1-7.

You might need different approaches for your particular tasks. To learn more about alternative approaches, see "Choose a Parallel Computing Solution" on page 1-16.

Tip You can further profile a `parfor`-loop by measuring how much data is transferred to and from the workers in the parallel pool by using `ticBytes` and `tocBytes`. For more information and examples, see "Profiling parfor-loops" on page 2-52.

See Also

Related Examples

- "Discover Clusters" on page 5-12
- "Discover Clusters and Use Cluster Profiles" on page 5-11
- "Profiling parfor-loops" on page 2-52

- "Interactively Run a Loop in Parallel Using parfor" on page 1-7
- "Choose a Parallel Computing Solution" on page 1-16

Use parfor-Loops for Reduction Assignments

These two examples show `parfor`-loops using reduction assignments. A reduction is an accumulation across iterations of a loop. The example on the left uses `x` to accumulate a sum across 10 iterations of the loop. The example on the right generates a concatenated array, `1:10`. In both of these examples, the execution order of the iterations on the workers does not matter: while the workers calculate individual results for each iteration, the client properly accumulates and assembles the final loop result.

```
x = 0;
parfor i = 1:10
    x = x + i;
end
x

x =

    55
```

```
x2 = [];
n = 10;
parfor i = 1:n
    x2 = [x2, i];
end
x2

x2 =

    1    2    3    4    5    6    7    8
```

If the loop iterations operate in a nondeterministic sequence, you might expect the concatenation sequence in the example on the right to be nonconsecutive. However, MATLAB recognizes the concatenation operation and yields deterministic results.

The next example, which attempts to compute Fibonacci numbers, is not a valid `parfor`-loop because the value of an element of `f` in one iteration depends on the values of other elements of `f` calculated in other iterations.

```
f = zeros(1,50);
f(1) = 1;
f(2) = 2;
parfor n = 3:50
    f(n) = f(n-1) + f(n-2);
end
```

When you are finished with your loop examples, clear your workspace and delete your parallel pool of workers:

```
clear
delete(gcp)
```

See Also

More About

- "Reduction Variables" on page 2-41
- "Ensure That parfor-Loop Iterations are Independent" on page 2-9

Use Objects and Handles in parfor-Loops

In this section...
"Using Objects in parfor-Loops" on page 2-26
"Handle Classes" on page 2-26
"Sliced Variables Referencing Function Handles" on page 2-26

Using Objects in parfor-Loops

If you are passing objects into or out of a `parfor`-loop, the objects must properly facilitate being saved and loaded. For more information, see "Save and Load Process for Objects".

You cannot slice the fields of objects because of first-level indexing constraints. For details, see "Sliced Variables" on page 2-36.

For example, in the code on the left, both lines in the loop generate a classification error because of the indexing. In the code on the right, as a workaround for sliced output, you employ separate sliced arrays in the loop. Then you assign the structure fields after the loop is complete.

Invalid	Valid
```parfor i = 1:4 outputData.outArray1(i) = 1/i; outputData.outArray2(i) = i^2; end```	```parfor i = 1:4 outArray1(i) = 1/i; outArray2(i) = i^2; end outputData = struct('outArray1',outArray1,'outArray2'```

Handle Classes

You can send handle objects as inputs to the body of a `parfor`-loop. However, any changes made to handle objects on the workers during loop iterations are not automatically propagated back to the client. That is, changes made inside the loop are not automatically reflected after the loop.

To get the changes back to the client after the loop, explicitly assign the modified handle objects to output variables of the `parfor`-loop. In the following example, `maps` is a sliced input/output variable.

```
maps = {containers.Map(),containers.Map(),containers.Map()};
parfor ii = 1:numel(maps)
    mymap = maps{ii};    % input slice assigned to local copy
    for jj = 1:1000
        mymap(num2str(jj)) = rand;
    end
    maps{ii} = mymap;    % modified local copy assigned to output slice
end
```

Sliced Variables Referencing Function Handles

You cannot directly call a function handle with the loop index as an input argument, because this variable cannot be distinguished from a sliced input variable. If you must call a function handle with the loop index variable as an argument, use `feval`.

The following example uses a function handle and a `for`-loop.

```
B = @sin;
for ii = 1:100
    A(ii) = B(ii);
end
```

A corresponding `parfor`-loop does not allow `B` to reference a function handle. You can work around the problem using `feval`.

```
B = @sin;
parfor ii = 1:100
    A(ii) = feval(B,ii);
end
```

See Also

More About

Troubleshoot Variables in parfor-Loops

In this section...
"Ensure That parfor-Loop Variables Are Consecutive Increasing Integers" on page 2-28
"Avoid Overflows in parfor-Loops" on page 2-28
"Solve Variable Classification Issues in parfor-Loops" on page 2-29
"Structure Arrays in parfor-Loops" on page 2-30
"Converting the Body of a parfor-Loop into a Function" on page 2-31
"Unambiguous Variable Names" on page 2-32
"Transparent parfor-loops" on page 2-32
"Global and Persistent Variables" on page 2-32

Ensure That parfor-Loop Variables Are Consecutive Increasing Integers

Loop variables in a `parfor`-loop must be consecutive increasing integers. For this reason, the following examples return errors:

```
parfor i = 0:0.2:1      % not integers
parfor j = 1:2:11       % not consecutive
parfor k = 12:-1:1      % not increasing
```

You can fix these errors by converting the loop variables into a valid range. For example, you can fix the noninteger example as follows:

```
iValues = 0:0.2:1;
parfor idx = 1:numel(iValues)
    i = iValues(idx);
    ...
end
```

Avoid Overflows in parfor-Loops

If MATLAB detects that the `parfor`-loop variable can overflow, it reports an error.

Overflow condition	Example	Solution

The length of the `parfor`-loop range exceeds the maximum value of the loop variable type.	Here, MATLAB reports an error because `length(-128:127)>maxint('int8')`:
	
	`parfor idx=int8(-128:127)`
	`    idx;`
	`end`
		Use a larger data type for the `parfor`-loop variable. If you want to keep the original data type in your calculations, convert the `parfor`-loop variable inside the `parfor` loop.
		
		`parfor idx=-128:127`
		`    int8(idx);`
		`end`

Overflow condition	Example	Solution
The initial value of the `parfor`-loop range equals the minimum value of the loop variable type.	Here, MATLAB reports an error because `0=intmin('uint32')`: ```parfor idx=uint32(0:1) idx;end```	• Use a larger data type with a lower minimum value, as in the previous solution. • Increment the range of values. For example: ```parfor idx=uint32(0:1)+1 idx-1;end```

Solve Variable Classification Issues in parfor-Loops

When MATLAB recognizes a name in a `parfor`-loop as a variable, the variable is classified in one of several categories, shown in the following table. Make sure that your variables are uniquely classified and meet the category requirements. `parfor`-loops that violate the requirement return an error.

Classification	Description
"Loop Variables" on page 2-34	Loop indices
"Sliced Variables" on page 2-36	Arrays whose segments are operated on by different iterations of the loop
"Broadcast Variables" on page 2-40	Variables defined before the loop whose value is required inside the loop, but never assigned inside the loop
"Reduction Variables" on page 2-41	Variables that accumulates a value across iterations of the loop, regardless of iteration order
"Temporary Variables" on page 2-47	Variables created inside the loop, and not accessed outside the loop

To find out which variables you have, examine the code fragment. All variable classifications in the table are represented in this code:

If you run into variable classification problems, consider these approaches before you resort to the more difficult method of converting the body of a `parfor`-loop into a function.

• If you use a nested `for`-loop to index into a sliced array, you cannot use that array elsewhere in the `parfor`-loop. The code on the left does not work because A is sliced and indexed inside the nested `for`-loop. The code on the right works because v is assigned to A outside the nested loop. You can compute an entire row, and then perform a single assignment into the sliced output.

Invalid	Valid
``A = zeros(4, 10);
parfor i = 1:4
for j = 1:10
A(i, j) = i + j;
end
disp(A(i, 1))
end```	```A = zeros(4, 10);
parfor i = 1:4
v = zeros(1, 10);
for j = 1:10
v(j) = i + j;
end
disp(v(1))
A(i, :) = v;
end```

- The code on the left does not work because the variable x in `parfor` cannot be classified. This variable cannot be classified because there are multiple assignments to different parts of x. Therefore `parfor` cannot determine whether there is a dependency between iterations of the loop. The code on the right works because you completely overwrite the value of x. `parfor` can now determine unambiguously that x is a temporary variable.

Invalid	Valid
``parfor idx = 1:10
x(1) = 7;
x(2) = 8;
out(idx) = sum(x);
end```	```parfor idx = 1:10
x = [7, 8];
out(idx) = sum(x);
end```

- This example shows how to slice the field of a structured array. See `struct` for details. The code on the left does not work because the variable a in `parfor` cannot be classified. This variable cannot be classified because the form of indexing is not valid for a sliced variable. The first level of indexing is not the sliced indexing operation, even though the field x of a appears to be sliced correctly. The code on the right works because you extract the field of the `struct` into a separate variable tmpx. `parfor` can now determine correctly that this variable is sliced. In general, you cannot use fields of `struct`s or properties of objects as sliced input or output variables in `parfor`.

Invalid	Valid
``a.x = [];
parfor idx = 1:10
a.x(idx) = 7;
end```	```tmpx = [];
parfor idx = 1:10
tmpx(idx) = 7;
end
a.x = tmpx;```

Structure Arrays in parfor-Loops

Creating Structures as Temporaries

You cannot create a structure in a `parfor`-loop using dot notation assignment. In the code on the left, both lines inside the loop generate a classification error. In the code on the right, as a workaround you can use the `struct` function to create the structure in the loop or in the first field.

Invalid	Valid
```parfor i = 1:4 temp.myfield1 = rand(); temp.myfield2 = i; end ```	```parfor i = 1:4 temp = struct(); temp.myfield1 = rand(); temp.myfield2 = i; end parfor i = 1:4 temp = struct('myfield1',rand(),'myfield2',i); end ```

Slicing Structure Fields

You cannot use structure fields as sliced *input or output* arrays in a `parfor`-loop. In other words, you cannot use the loop variable to index the elements of a structure field. In the code on the left, both lines in the loop generate a classification error because of the indexing. In the code on the right, as a workaround for sliced output, you employ separate sliced arrays in the loop. Then you assign the structure fields after the loop is complete.

Invalid	Valid
```parfor i = 1:4 outputData.outArray1(i) = 1/i; outputData.outArray2(i) = i^2; end ```	```parfor i = 1:4 outArray1(i) = 1/i; outArray2(i) = i^2; end outputData = struct('outArray1',outArray1,'outArray2' ```

The workaround for sliced input is to assign the structure field to a separate array before the loop. You can use that new array for the sliced input.

```
inArray1 = inputData.inArray1;
inArray2 = inputData.inArray2;
parfor i = 1:4
    temp1 = inArray1(i);
    temp2 = inArray2(i);
end
```

Converting the Body of a parfor-Loop into a Function

If all else fails, you can usually solve variable classification problems in `parfor`-loops by converting the body of the `parfor`-loop into a function. In the code on the left, Code Analyzer flags a problem with variable *y*, but cannot resolve it. In the code on the right, you solve this problem by converting the body of the `parfor`-loop into a function.

Invalid	Valid

```matlab
function parfor_loop_body_bad
    data = rand(5,5);
    means = zeros(1,5);
    parfor i = 1:5
        % Code Analyzer flags problem
        % with variable y below
        y.mean = mean(data(:,i));
        means(i) = y.mean;
    end
    disp(means);
end
```

```matlab
function parfor_loop_body_good
    data = rand(5,5);
    means = zeros(1,5);
    parfor i = 1:5
        % Call a function instead
        means(i) = computeMeans(data(:,i));
    end
    disp(means);
end

% This function now contains the body
% of the parfor-loop
function means = computeMeans(data)
    y.mean = mean(data);
    means = y.mean;
end

Starting parallel pool (parpool) using the 'local' pr
    0.6786    0.5691    0.6742    0.6462    0.6307
```

Unambiguous Variable Names

If you use a name that MATLAB cannot unambiguously distinguish as a variable inside a `parfor`-loop, at parse time MATLAB assumes you are referencing a function. Then at run-time, if the function cannot be found, MATLAB generates an error. See "Variable Names". For example, in the following code `f(5)` could refer either to the fifth element of an array named `f`, or to a function named `f` with an argument of `5`. If `f` is not clearly defined as a variable in the code, MATLAB looks for the function `f` on the path when the code runs.

```matlab
parfor i = 1:n
    ...
    a = f(5);
    ...
end
```

Transparent parfor-loops

The body of a `parfor`-loop must be *transparent*: all references to variables must be "visible" in the text of the code. For more details about transparency, see "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49.

Global and Persistent Variables

The body of a `parfor`-loop cannot contain `global` or `persistent` variable declarations.

See Also

More About

- "Decide When to Use parfor" on page 2-2
- "Convert for-Loops Into parfor-Loops" on page 2-6

- "Ensure That parfor-Loop Iterations are Independent" on page 2-9
- "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12
- "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49
- "Use parfor-Loops for Reduction Assignments" on page 2-25
- "Run Parallel Simulations" (Simulink)

Loop Variables

The loop variable defines the loop index value for each iteration. You set it in the first line of a `parfor` statement.

```
parfor p=1:12
```

For values across all iterations, the loop variable must evaluate to ascending consecutive integers. Each iteration is independent of all others, and each has its own loop index value.

Required (static): Assignments to the loop variable are not allowed.

This restriction is required, because changing `p` in the `parfor` body cannot guarantee the independence of iterations.

This example attempts to modify the value of the loop variable `p` in the body of the loop, and thus is invalid.

```
parfor p = 1:n
    p = p + 1;
    a(p) = i;
end
```

Required (static): You cannot index or subscript the loop variable in any way.

This restriction is required, because referencing a field of a loop variable cannot guarantee the independence of iterations.

The following code attempts to reference a field (`b`) of the loop variable (`p`) as if it were a structure. Both lines within the loop are invalid.

```
parfor p = 1:n
    p.b = 3
    x(p) = fun(p.b)
end
```

Similarly, the following code is invalid because it attempts to index the loop variable as a 1-by-1 matrix:

```
parfor p = 1:n
    x = p(1)
end
```

Required (static): You cannot use a range increment in `for`-loops nested inside a `parfor`-loop.

Consider the following example:

```
N = 10;
T = 3;
A = zeros(N,T);
B = zeros(N,T);
```

The following code is invalid.

```
parfor i = 1:1:N
    for t = 1:1:T
```

```
        A(i,t) = t;
    end
end
```

The following code is valid.

```
parfor i = 1:1:N
    for t = 1:T
        B(i,t) = t;
    end
end
```

See Also

```
parfor
```

More About

- "Troubleshoot Variables in parfor-Loops" on page 2-28

Sliced Variables

A *sliced variable* is one whose value can be broken up into segments, or *slices*, which are then operated on separately by different workers. Each iteration of the loop works on a different slice of the array. Using sliced variables can reduce communication between the client and workers.

In this example, the workers apply f to the elements of A separately.

```
parfor i = 1:length(A)
    B(i) = f(A(i));
end
```

Characteristics of a Sliced Variable

If a variable in a `parfor`-loop has all the following characteristics, then the variable is sliced:

- Type of First-Level Indexing — The first level of indexing is either parentheses, (), or braces, {}.
- Fixed Index Listing — Within the first-level parentheses or braces, the list of indices is the same for all occurrences of a given variable.
- Form of Indexing — Within the list of indices for the variable, exactly one index involves the loop variable.
- Shape of Array — The array maintains a constant shape. In assigning to a sliced variable, the right side of the assignment cannot be [] or ' ', because these operators attempt to delete elements.

Type of First-Level Indexing. For a sliced variable, the first level of indexing is enclosed in either parentheses, (), or braces, {}.

Here are the forms for the first level of indexing for arrays that are sliced and not sliced.

Not Sliced	Sliced
A.x	A(...)
A.(...)	A{...}

After the first level, you can use any type of valid MATLAB indexing in the second and subsequent levels.

The variable A shown here on the left is not sliced; that shown on the right is sliced.

A.q{i,12}   A{i,12}.q

Fixed Index Listing. Within the first-level indexing of a sliced variable, the list of indices is the same for all occurrences of a given variable.

The variable A on the left is not sliced because A is indexed by i and i+1 in different places. In the code on the right, variable A is sliced correctly.

Not sliced	Sliced
```parfor i = 1:k B(:) = h(A(i), A(i+1)); end```	```parfor i = 1:k B(:) = f(A(i)); C(:) = g(A{i}); end```

The example on the right shows occurrences of first-level indexing using both parentheses and braces in the same loop, which is acceptable.

The following example on the left does not slice A because the indexing of A is not the same in all places. The example on the right slices both A and B. The indexing of A is not the same as the indexing of B. However, the indexing of both A and B are individually consistent.

Not sliced	Sliced
``parfor i=1:10
b = A(1,i) + A(2,i)
end```	```A = [1 2 3 4 5 6 7 8 9 10;
10 20 30 40 50 60 70 80 90 100];
B = zeros(1,10);
parfor i=1:10
for n=1:2
B(i) = B(i)+A(n,i)
end
end```

Form of Indexing. Within the first-level of indexing for a sliced variable, exactly one indexing expression is of the form i, $i+k$, $i-k$, or $k+i$. The index i is the loop variable and k is a scalar integer constant or a simple (non-indexed) broadcast variable. Every other indexing expression is a positive integer constant, a simple (non-indexed) broadcast variable, a nested `for`-loop index variable, colon, or `end`.

With i as the loop variable, the A variables shown on the left are not sliced, while the A variables on the right are sliced.

Not sliced	Sliced
``A(i+f(k),j,:,3) % f(k) invalid for slicing
A(i,20:30,end) % 20:30 not scalar
A(i,:,s.field1) % s.field1 not simple broadcast var```	```A(i+k,j,:,3)
A(i,:,end)
A(i,:,k)```

When you use other variables along with the loop variable to index an array, you cannot set these variables inside the loop. In effect, such variables are constant over the execution of the entire `parfor` statement. You cannot combine the loop variable with itself to form an index expression.

Shape of Array. A sliced variable must maintain a constant shape. The variable A shown here is not sliced:

```
A(i,:) = [];
```

A is not sliced because changing the shape of a sliced array would violate assumptions governing communication between the client and workers.

Sliced Input and Output Variables

A sliced variable can be an input variable, an output variable, or both. MATLAB transmits sliced input variables from the client to the workers, and sliced output variables from workers back to the client. If a variable is both input and output, it is transmitted in both directions.

In this `parfor`-loop, A is a sliced input variable and B is a sliced output variable.

```
A = rand(1,10);
parfor ii = 1:10
    B(ii) = A(ii);
end
```

However, if MATLAB determines that, in each iteration, the sliced variable elements are set before any use, then MATLAB does not transmit the variable to the workers. In this example, all elements of A are set before any use.

```
parfor ii = 1:n
    if someCondition
        A(ii) = 32;
    else
        A(ii) = 17;
    end
    % loop code that uses A(ii)
end
```

Sliced-output variables can grow dynamically through indexed assignments with default values inserted at intermediate indices. In this example, you can see that the default value of 0 has been inserted at several places in A.

```
A = [];
parfor idx = 1:10
    if rand < 0.5
        A(idx) = idx;
    end
end

disp(A);
    0    2    0    4    5    0    0    8    9    10
```

Even if a sliced variable is not explicitly referenced as an input, implicit usage can make it so. In the following example, not all elements of A are necessarily set inside the parfor-loop. Therefore the original values of the array are received, held, and then returned from the loop.

```
A = 1:10;
parfor ii = 1:10
    if rand < 0.5
        A(ii) = 0;
    end
end
```

Under some circumstances, parfor-loops must assume that a worker may need all segments of a sliced variable. In this example, it is not possible to determine which elements of the sliced variable will be read before execution, so parfor sends all possible segments.

```
A = 1:10;
parfor ii=1:11
    if ii <= randi([10 11])
        A(ii) = A(ii) + 1;
    end
end
```

Note that in these circumstances, the code can attempt to index a sliced variable outside of the array bounds and generate an error.

Nested for-Loops with Sliced Variables

When you index a sliced variable with a nested for-loop variable, keep these requirements in mind:

- The sliced variable must be enclosed within the corresponding for-loop.

  In this example, the code on the left does not work because it indexes the sliced variable A outside the nested for-loop that defines j.

Not Sliced	Sliced
| ```
A = zeros(10);
parfor i=1:10
    for j=1:10
    end
    A(i,j)=1;
end
``` | ```
A = zeros(10);
parfor i=1:10
    for j=1:10
        A(i,j) = 1;
    end
end
``` |

- The range of the for-loop variable must be a row vector of positive constant numbers or variables.

  In this example, the code on the left does not work because it defines the upper limit of the nested for-loop with a function call. The code on the right provides a workaround by defining the upper limit in a constant variable outside the parfor-loop.

Not Sliced	Sliced
| ```
A = zeros(10);

parfor i=1:10
    for j=1:size(A,2)
        A(i,j)=1;
    end
end
``` | ```
A = zeros(10);
L = size(A,2);
parfor=1:10
    for j=1:L
        A(i,j)=1;
    end
end
``` |

- The for-loop variable must not be assigned other than by its for statement.

  In this example, the code on the left does not work because it reassigns the for-loop variable inside the for-loop. The code on the right provides a workaround by assigning i to the temporary variable t.

Not Sliced	Sliced
| ```
A = zeros(10);
parfor i=1:10
    for j=1:10
        if i == j
            j = i;
            A(i,j) = j;
        end
    end
end
``` | ```
A = zeros(10);
parfor i=1:10
    for j=1:10
        if i == j
            t = i;
            A(i,j) = t;
        end
    end
end
``` |

See Also

More About

- "Troubleshoot Variables in parfor-Loops" on page 2-28

Broadcast Variables

A *broadcast variable* is any variable, other than the loop variable or a sliced variable, that does not change inside the loop. At the start of a `parfor`-loop, the values of any broadcast variables are sent to all workers. This type of variable can be useful or even essential for particular tasks. However, large broadcast variables can cause significant communication between client and workers and increase parallel overhead. Sometimes it is more efficient to use temporary variables for this purpose, creating and assigning them inside the loop.

For more details, see "Temporary Variables" on page 2-47 and "Deciding When to Use parfor" on page 2-2.

See Also

More About

- "Troubleshoot Variables in parfor-Loops" on page 2-28
- "Deciding When to Use parfor" on page 2-2
- "Temporary Variables" on page 2-47

Reduction Variables

MATLAB supports an important exception, called reduction, to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order. MATLAB allows reduction variables in `parfor`-loops.

Reduction variables appear on both sides of an assignment statement, such as any of the following, where `expr` is a MATLAB expression.

X = X + expr	X = expr + X
X = X - expr	See Associativity in Reduction Assignments in "Requirements for Reduction Assignments" on page 2-43
X = X .* expr	X = expr .* X
X = X * expr	X = expr * X
X = X & expr	X = expr & X
X = X | expr	X = expr | X
X = [X, expr]	X = [expr, X]
X = [X; expr]	X = [expr; X]
X = min(X, expr)	X = min(expr, X)
X = max(X, expr)	X = max(expr, X)
X = union(X, expr)	X = union(expr, X)
X = intersect(X, expr)	X = intersect(expr, X)

Each of the allowed statements listed in this table is referred to as a *reduction assignment*. By definition, a reduction variable can appear only in assignments of this type.

The general form of a reduction assignment is

X = f(X, expr)	X = f(expr, X)

The following example shows a typical usage of a reduction variable X.

```
X = 0;              % Do some initialization of X
parfor i = 1:n
    X = X + d(i);
end
```

This loop is equivalent to the following, where you calculate each `d(i)` by a different iteration.

```
X = X + d(1) + ... + d(n)
```

In a regular `for`-loop, the variable X would get its value either before entering the loop or from the previous iteration of the loop. However, this concept does not apply to `parfor`-loops.

In a `parfor`-loop, the value of X is never transmitted from client to workers or from worker to worker. Rather, additions of `d(i)` are done in each worker, with `i` ranging over the subset of `1:n` being performed on that worker. The results are then transmitted back to the client, which adds the partial sums of the workers into X. Thus, workers do some of the additions, and the client does the rest.

Notes About Required and Recommended Guidelines

If your `parfor` code does not adhere to the guidelines and restrictions labeled as **Required**, you get an error. MATLAB catches some of these errors at the time it reads the code, and others when it executes the code. These errors are labeled as **Required (static)** or **Required (dynamic)** respectively. Guidelines that do not cause errors are labeled as **Recommended**. You can use MATLAB Code Analyzer to help `parfor`-loops comply with the guidelines.

Basic Rules for Reduction Variables

The following requirements further define the reduction assignments associated with a given variable.

> **Required (static)**: For any reduction variable, the same reduction function or operation must be used in all reduction assignments for that variable.

The `parfor`-loop on the left is not valid because the reduction assignment uses + in one instance, and [,] in another. The `parfor`-loop on the right is valid.

Invalid	Valid
<pre>parfor i = 1:n
if testLevel(k)
A = A + i;
else
A = [A, 4+i];
end
% loop body continued
end</pre>	<pre>parfor i = 1:n
if testLevel(k)
A = A + i;
else
A = A + i + 5*k;
end
% loop body continued
end</pre>

> **Required (static)**: If the reduction assignment uses *, [,], or [;], then X must be consistently specified as the first or second argument in every reduction assignment.

The `parfor`-loop on the left is not valid because the order of items in the concatenation is not consistent throughout the loop. The `parfor`-loop on the right is valid.

Invalid	Valid
<pre>parfor i = 1:n
if testLevel(k)
A = [A, 4+i];
else
A = [r(i), A];
end
% loop body continued
end</pre>	<pre>parfor i = 1:n
if testLevel(k)
A = [A, 4+i];
else
A = [A, r(i)];
end
% loop body continued
end</pre>

> **Required (static)**: You cannot index or subscript a reduction variable.

The code on the left is not valid because it tries to index `a`, and so MATLAB cannot classify it as a reduction variable. To fix it, the code on the right uses a non-indexed variable.

Invalid	Valid
``a.x = 0
parfor i = 1:10
a.x = a.x + 1;
end```	```tmpx = 0
parfor i = 1:10
tmpx = tmpx + 1;
end
a.x = tmpx;```

Requirements for Reduction Assignments

Reduction Assignments. In addition to the specific forms of reduction assignment listed in the table in "Reduction Variables" on page 2-41, the only other (and more general) form of a reduction assignment is

X = f(X, expr)	X = f(expr, X)

Required (static): f can be a function or a variable. If f is a variable, then you cannot change f in the parfor body (in other words, it is a broadcast variable).

If f is a variable, then for all practical purposes its value at run time is a function handle. However, as long as the right side can be evaluated, the resulting value is stored in X.

The parfor-loop on the left does not execute correctly because the statement f = @times causes f to be classified as a temporary variable. Therefore f is cleared at the beginning of each iteration. The parfor-loop on the right is correct, because it does not assign f inside the loop.

Invalid	Valid
``f = @(x,k)x * k;
parfor i = 1:n
a = f(a,i);
% loop body continued
f = @times; % Affects f
end```	```f = @(x,k)x * k;
parfor i = 1:n
a = f(a,i);
% loop body continued
end```

The operators && and || are not listed in the table in "Reduction Variables" on page 2-41. Except for && and ||, all the matrix operations of MATLAB have a corresponding function f, such that u op v is equivalent to f(u,v). For && and ||, such a function cannot be written because u&&v and u||v might or might not evaluate v. However, f(u,v) *always* evaluates v before calling f. Therefore && and || are excluded from the table of allowed reduction assignments for a parfor-loop.

Every reduction assignment has an associated function f. The properties of f that ensure deterministic behavior of a parfor statement are discussed in the following sections.

Associativity in Reduction Assignments. The following practice is recommended for the function f, as used in the definition of a reduction variable. However, this rule does not generate an error if not adhered to. Therefore, it is up to you to ensure that your code meets this recommendation.

Recommended: To get deterministic behavior of parfor-loops, the reduction function f must be associative.

To be associative, the function f must satisfy the following for all a, b, and c.

```
f(a,f(b,c)) = f(f(a,b),c)
```

The classification rules for variables, including reduction variables, are purely syntactic. They cannot determine whether the `f` you have supplied is truly associative or not. Associativity is assumed, but if you violate this rule, each execution of the loop might result in different answers.

Note The addition of mathematical real numbers is associative. However, the addition of floating-point numbers is only approximately associative. Different executions of this `parfor` statement might produce values of X with different round-off errors. You cannot avoid this cost of parallelism.

For example, the statement on the left yields 1, while the statement on the right returns `1 + eps`:

`(1 + eps/2) + eps/2`   `1 + (eps/2 + eps/2)`

Except for the minus operator (`-`), all special cases listed in the table in "Reduction Variables" on page 2-41 have a corresponding (approximately) associative function. MATLAB calculates the assignment `X = X - expr` by using `X = X + (-expr)`. (So, technically, the function for calculating this reduction assignment is `plus`, not `minus`.) However, the assignment `X = expr - X` cannot be written using an associative function, which explains its exclusion from the table.

Commutativity in Reduction Assignments. Some associative functions, including `+`, `.*`, `min`, and `max`, `intersect`, and `union`, are also commutative. That is, they satisfy the following for all `a` and `b`.

`f(a,b) = f(b,a)`

Noncommutative functions include `*` (because matrix multiplication is not commutative for matrices in which both dimensions have size greater than one), `[,]`, and `[;]`. Noncommutativity is the reason that consistency in the order of arguments to these functions is required. As a practical matter, a more efficient algorithm is possible when a function is commutative as well as associative, and `parfor` is optimized to exploit commutativity.

> **Recommended:** Except in the cases of `*`, `[,]`, and `[;]`, the function `f` of a reduction assignment must be commutative. If `f` is not commutative, different executions of the loop might result in different answers.

Violating the restriction on commutativity in a function used for reduction could result in unexpected behavior, even if it does not generate an error.

Unless `f` is a known noncommutative built-in function, it is assumed to be commutative. There is currently no way to specify a user-defined, noncommutative function in `parfor`.

> **Recommended:** An overload of `+`, `*`, `.*`, `[,]`, or `[;]` must be associative if it is used in a reduction assignment in a `parfor`-loop.

> **Recommended:** An overload of `+`, `.*`, `union`, or `intersect` must be commutative.

Similarly, because of the special treatment of `X = X - expr`, the following is recommended.

> **Recommended:** An overload of the minus operator (`-`) must obey the mathematical law that `X - (y + z)` is equivalent to `(X - y) - z`.

Using a Custom Reduction Function

Suppose that each iteration of a loop performs some calculation, and you are interested in finding which iteration of a loop produces the maximum value. This reduction exercise makes an

accumulation across multiple iterations of a loop. Your reduction function must compare iteration results, until the maximum value can be determined after all iterations are compared.

First consider the reduction function itself. To compare one iteration result against another, the function requires as input the current result and the known maximum from other iterations so far. Each of the two inputs is a vector containing iteration results and iteration number.

```matlab
function mc = comparemax(A, B)
    % Custom reduction function for 2-element vector input

    if A(1) >= B(1) % Compare the two input data values
        mc = A;      % Return the vector with the larger result
    else
        mc = B;
    end
end
```

Inside the loop, each iteration calls the reduction function (`comparemax`), passing in a pair of two-element vectors:

- The accumulated maximum and its iteration index, which is the reduction variable `cummax`
- The iteration value and index

If the data value of the current iteration is greater than the maximum in `cummmax`, the function returns a vector of the new value and its iteration number. Otherwise, the function returns the existing maximum and its iteration number.

Each iteration calls the reduction function `comparemax` to compare its own data `[dat i]` to data already accumulated in `cummax`. Try the following code for this loop.

```matlab
% First element of cummax is maximum data value
% Second element of cummax is where (iteration) maximum occurs
cummax = [0 0];  % Initialize reduction variable
parfor ii = 1:100
    dat = rand(); % Simulate some actual computation
    cummax = comparemax(cummax, [dat ii]);
end
disp(cummax);
```

Chaining Reduction Operators

MATLAB classifies assignments of the form $X = expr\ op\ X$ or $X = X\ op\ expr$ as reduction statements when they are equivalent to the parenthesized assignments $X = (expr)\ op\ X$ or $X = X\ op\ (expr)$ respectively. X is a variable, `op` is a reduction operator, and `expr` is an expression with one or more binary reduction operators. Consequently, due to the MATLAB operator precedence rules, MATLAB might not classify some assignments of the form $X = expr\ op1\ X\ op2\ expr2\ ...,$ that chain operators, as reduction statements in `parfor`-loops.

In this example, MATLAB classifies X as a reduction variable because the assignment is equivalent to $X = X + (1\ *\ 2)$.

```matlab
X = 0;
parfor i=1:10
    X = X + 1 * 2;
end
```

In this example, MATLAB classifies X as a temporary variable because the assignment, equivalent to X = (X * 1) + 2, is not of the form X = (expr) op X or X = X op (expr).

```
X = 0;
parfor i=1:10
    X = X * 1 + 2;
end
```

As a best practice, use parentheses to explicitly specify operator precedence for chained reduction assignments.

See Also

More About

- "Troubleshoot Variables in parfor-Loops" on page 2-28
- "Use parfor-Loops for Reduction Assignments" on page 2-25

Temporary Variables

A *temporary variable* is any variable that is the target of a direct, nonindexed assignment, but is not a reduction variable. In the following `parfor`-loop, `a` and `d` are temporary variables:

```
a = 0;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;              % Variable a is temporary
    z = z + i;
    if i <= 5
        d = 2*a;        % Variable d is temporary
    end
end
```

In contrast to the behavior of a `for`-loop, MATLAB clears any temporary variables before each iteration of a `parfor`-loop. To help ensure the independence of iterations, the values of temporary variables cannot be passed from one iteration of the loop to another. Therefore, temporary variables must be set inside the body of a `parfor`-loop, so that their values are defined separately for each iteration.

MATLAB does not send temporary variables back to the client. A temporary variable in a `parfor`-loop has no effect on a variable with the same name that exists outside the loop. This behavior is different from ordinary `for`-loops.

Uninitialized Temporaries

Temporary variables in a `parfor`-loop are cleared at the beginning of every iteration. MATLAB can sometimes detect cases in which loop iterations use a temporary variable before it is set in that iteration. In this case, MATLAB issues a static error rather than a run-time error. There is little point in allowing execution to proceed if a run-time error is guaranteed to occur. This kind of error often arises because of confusion between `for` and `parfor`, especially regarding the rules of classification of variables. For example:

```
b = true;
parfor i = 1:n
    if b && some_condition(i)
        do_something(i);
        b = false;
    end
    ...
end
```

This loop is acceptable as an ordinary `for`-loop. However, as a `parfor`-loop, `b` is a temporary variable because it occurs directly as the target of an assignment inside the loop. Therefore it is cleared at the start of each iteration, so its use in the condition of the `if` is guaranteed to be uninitialized. If you change `parfor` to `for`, the value of `b` assumes sequential execution of the loop. In that case, `do_something(i)` is executed only for the lower values of `i` until `b` is set `false`.

Temporary Variables Intended as Reduction Variables

Another common cause of uninitialized temporaries can arise when you have a variable that you intended to be a reduction variable. However, if you use it elsewhere in the loop, then it is classified as a temporary variable. For example:

```
s = 0;
parfor i = 1:n
    s = s + f(i);
    ...
    if (s > whatever)
        ...
    end
end
```

If the only occurrences of `s` are the two in the first statement of the body, `s` would be classified as a reduction variable. But in this example, `s` is not a reduction variable because it has a use outside of reduction assignments in the line `s > whatever`. Because `s` is the target of an assignment (in the first statement), it is a temporary. Therefore MATLAB issues an error, but points out the possible connection with reduction.

If you change `parfor` to `for`, the use of `s` outside the reduction assignment relies on the iterations being performed in a particular order. In a `parfor`-loop, it matters that the loop "does not care" about the value of a reduction variable as it goes along. It is only after the loop that the reduction value becomes usable.

ans Variable

Inside the body of a `parfor`-loop, the `ans` variable is classified as a temporary variable. All considerations and restrictions for temporary variables apply to `ans`. For example, assignments to `ans` inside a `parfor`-loop have no effect on `ans` outside the loop.

See Also

More About

- "Troubleshoot Variables in parfor-Loops" on page 2-28
- "Reduction Variables" on page 2-41
- "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49

Ensure Transparency in parfor-Loops or spmd Statements

The body of a `parfor`-loop or `spmd` block must be *transparent*. Transparency means that all references to variables must be visible in the text of the code.

In the following examples, the variable X is not transferred to the workers. Only the character vector `'X'` is passed to `eval`, and X is not visible as an input variable in the loop or block body. As a result, MATLAB issues an error at run time.

```
X = 5;                                    X = 5;
parfor ii = 1:4                           spmd
    eval('X');                                eval('X');
end                                       end
```

Similarly, you cannot clear variables from a workspace by executing `clear` inside a `parfor` or `spmd` statement:

```
parfor ii = 1:4                           spmd; clear('X'); end
    <statements...>
    clear('X')  % cannot clear: transparency violation
    <statements...>
end
```

Alternatively, you can free up memory used by a variable by setting its value to empty when it is no longer needed.

```
parfor ii = 1:4
    <statements...>
    X = [];
    <statements...>
end
```

In the case of `spmd` blocks, you can clear its Composite from the client workspace.

In general, the requirement for transparency restricts all dynamic access to variables, because the entire variable might not be present in any given worker. In a transparent workspace, you cannot create, delete, modify, access, or query variables if you do not explicitly specify these variables in the code.

Examples of other actions or functions that violate transparency in a `parfor`-loop include:

- `who` and `whos`
- `evalc`, `evalin`, and `assignin` with the `workspace` argument specified as `'caller'`
- `save` and `load`, unless the output of `load` is assigned to a variable
- If a script attempts to read or write variables of the parent workspace, then running this script can cause a transparency violation. To avoid this issue, convert the script to a function, and call it with the necessary variables as input or output arguments.

Note Transparency applies only to the direct body of the `parfor` or `spmd` construct, and not to any functions called from there. The workaround for `save` and `load` is to hide the calls to `save` and `load` inside a function.

MATLAB *does* successfully execute `eval` and `evalc` statements that appear in functions called from the `parfor` body.

Parallel Simulink Simulations

You can run Simulink models in parallel with the `parsim` command instead of using `parfor`-loops. For more information and examples of using Simulink in parallel, see "Run Multiple Simulations" (Simulink).

- If your Simulink model requires access to variables contained in a `.mat` file, you must load these parameters in the workspace of each worker. You must do this before the `parfor`-loop, and after opening `parpool`. To achieve this, you can use `spmd` or `parfevalOnAll`, as shown in the examples.

  ```
  spmd
      evalin('base', 'load(''path/to/file'')')
  end

  parfevalOnAll(@evalin, 0, 'base', 'load(''path/to/file'')')
  ```

- If your model also requires variables defined in the body of your MATLAB script, you must use `assignin` or `evalin` to move these variables to the base workspace of each worker, in every `parfor` iteration.

See Also
`parfor` | `spmd`

More About
- "Troubleshoot Variables in parfor-Loops" on page 2-28
- "Run Single Programs on Multiple Data Sets" on page 3-2
- "Run Parallel Simulations" (Simulink)

Improve parfor Performance

You can improve the performance of `parfor`-loops in various ways. This includes parallel creation of arrays inside the loop; profiling `parfor`-loops; slicing arrays; and optimizing your code on local workers before running on a cluster.

Where to Create Arrays

When you create a large array in the client before your `parfor`-loop, and access it within the loop, you might observe slow execution of your code. To improve performance, tell each MATLAB worker to create its own arrays, or portions of them, in parallel. You can save the time of transferring data from client to workers by asking each worker to create its own copy of these arrays, in parallel, inside the loop. Consider changing your usual practice of initializing variables before a `for`-loop, avoiding needless repetition inside the loop. You might find that parallel creation of arrays inside the loop improves performance.

Performance improvement depends on different factors, including

*   size of the arrays
*   time needed to create arrays
*   worker access to all or part of the arrays
*   number of loop iterations that each worker performs

Consider all factors in this list when you are considering to convert `for`-loops to `parfor`-loops. For more details, see "Convert for-Loops Into parfor-Loops" on page 2-6.

As an alternative, consider the `parallel.pool.Constant` function to establish variables on the pool workers before the loop. These variables remain on the workers after the loop finishes, and remain available for multiple `parfor`-loops. You might improve performance using `parallel.pool.Constant`, because the data is transferred only once to the workers.

In this example, you first create a big data set D and execute a `parfor`-loop accessing D. Then you use D to build a `parallel.pool.Constant` object, which allows you to reuse the data by copying D to each worker. Measure the elapsed time using `tic` and `toc` for each case and note the difference.

```
function constantDemo
    D = rand(1e7, 1);
    tic
    for i = 1:20
        a = 0;
        parfor j = 1:60
            a = a + sum(D);
        end
    end
    toc

    tic
    D = parallel.pool.Constant(D);
    for i = 1:20
        b = 0;
        parfor j = 1:60
            b = b + sum(D.Value);
        end
```

```
        end
    toc
end
```

```
>> constantDemo
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Elapsed time is 63.839702 seconds.
Elapsed time is 10.194815 seconds.
```

In the second case, you send the data only once. You can enhance the performance of the `parfor`-loop by using the `parallel.pool.Constant` object.

Profiling parfor-loops

You can profile a `parfor`-loop by measuring the time elapsed using `tic` and `toc`. You can also measure how much data is transferred to and from the workers in the parallel pool by using `ticBytes` and `tocBytes`. Note that this is different from profiling MATLAB code in the usual sense using the MATLAB profiler, see "Profile Your Code to Improve Performance".

This example calculates the spectral radius of a matrix and converts a `for`-loop into a `parfor`-loop. Measure the resulting speedup and the amount of transferred data.

1   In the MATLAB Editor, enter the following `for`-loop. Add `tic` and `toc` to measure the time elapsed. Save the file as `MyForLoop.m`.

```
function a = MyForLoop(A)
    tic
    for i = 1:200
        a(i) = max(abs(eig(rand(A))));
    end
    toc
end
```

2   Run the code, and note the elapsed time.

```
a = MyForLoop(500);
```

```
Elapsed time is 31.935373 seconds.
```

3   In `MyForLoop.m`, replace the `for`-loop with a `parfor`-loop. Add `ticBytes` and `tocBytes` to measure how much data is transferred to and from the workers in the parallel pool. Save the file as `MyParforLoop.m`.

```
ticBytes(gcp);
parfor i = 1:200
    a(i) = max(abs(eig(rand(A))));
end
tocBytes(gcp)
```

4   Run the new code, and run it again. Note that the first run is slower than the second run, because the parallel pool has to be started and you have to make the code available to the workers. Note the elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

```
a = MyParforLoop(500);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
...
```

	BytesSentToWorkers	BytesReceivedFromWorkers
1	15340	7024
2	13328	5712
3	13328	5704
4	13328	5728
Total	55324	24168

```
Elapsed time is 10.760068 seconds.
```

The elapsed time is 31.9 seconds in serial and 10.8 seconds in parallel, and shows that this code benefits from converting to a `parfor`-loop.

Slicing Arrays

If a variable is initialized before a `parfor`-loop, then used inside the `parfor`-loop, it has to be passed to each MATLAB worker evaluating the loop iterations. Only those variables used inside the loop are passed from the client workspace. However, if all occurrences of the variable are indexed by the loop variable, each worker receives only the part of the array it needs.

As an example, you first run a `parfor`-loop using a sliced variable and measure the elapsed time.

```
% Sliced version

M = 100;
N = 1e6;
data = rand(M, N);

tic
parfor idx = 1:M
    out2(idx) = sum(data(idx, :)) ./ N;
end
toc
```

```
Elapsed time is 2.261504 seconds.
```

Now suppose that you accidentally use a reference to the variable `data` instead of `N` inside the `parfor`-loop. The problem here is that the call to `size(data, 2)` converts the sliced variable into a broadcast (non-sliced) variable.

```
% Accidentally non-sliced version

clear

M = 100;
N = 1e6;
data = rand(M, N);

tic
parfor idx = 1:M
    out2(idx) = sum(data(idx, :)) ./ size(data, 2);
end
toc
```

```
Elapsed time is 8.369071 seconds.
```

Note that the elapsed time is greater for the accidentally broadcast variable.

In this case, you can easily avoid the non-sliced usage of `data`, because the result is a constant, and can be computed outside the loop. In general, you can perform computations that depend only on broadcast data before the loop starts, since the broadcast data cannot be modified inside the loop. In this case, the computation is trivial, and results in a scalar result, so you benefit from taking the computation out of the loop.

Optimizing on Local vs. Cluster Workers

Running your code on local workers might offer the convenience of testing your application without requiring the use of cluster resources. However, there are certain drawbacks or limitations with using local workers. Because the transfer of data does not occur over the network, transfer behavior on local workers might not be indicative of how it will typically occur over a network.

With local workers, because all the MATLAB worker sessions are running on the same machine, you might not see any performance improvement from a `parfor`-loop regarding execution time. This can depend on many factors, including how many processors and cores your machine has. The key point here is that a cluster might have more cores available than your local machine. If your code can be multithreaded by MATLAB, then the only way to go faster is to use more cores to work on the problem, using a cluster.

You might experiment to see if it is faster to create the arrays before the loop (as shown on the left below), rather than have each worker create its own arrays inside the loop (as shown on the right).

Try the following examples running a parallel pool locally, and notice the difference in time execution for each loop. First open a local parallel pool:

```
parpool('local')
```

Run the following examples, and execute again. Note that the first run for each case is slower than the second run, because the parallel pool has to be started and you have to make the code available to the workers. Note the elapsed time, for each case, for the second run.

```
tic;                                 tic;
n = 200;                             n = 200;
M = magic(n);                        parfor i = 1:n
R = rand(n);                             M = magic(n);
parfor i = 1:n                           R = rand(n);
    A(i) = sum(M(i,:).*R(n+1-i,:));       A(i) = sum(M(i,:).*R(n+1-i,:));
end                                  end
toc                                  toc
```

Running on a remote cluster, you might find different behavior, as workers can simultaneously create their arrays, saving transfer time. Therefore, code that is optimized for local workers might not be optimized for cluster workers, and vice versa.

See Also
`parallel.pool.Constant`

More About
- "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49
- "Use parfor-Loops for Reduction Assignments" on page 2-25

Run Code on Parallel Pools

In this section...
"What Is a Parallel Pool?" on page 2-55
"Automatically Start and Stop a Parallel Pool" on page 2-55
"Alternative Ways to Start and Stop Pools" on page 2-56
"Pool Size and Cluster Selection" on page 2-58

What Is a Parallel Pool?

A parallel pool is a set of MATLAB workers on a compute cluster or desktop. By default, a parallel pool starts automatically when needed by parallel language features such as `parfor`. You can specify the default pool size and cluster in your parallel preferences. The preferences panel displays your pool size and cluster when you select **Parallel Preferences** in the **Parallel** menu. You can change pool size and cluster in the **Parallel** menu. Alternatively, you can choose cluster and pool size using `parcluster` and `parpool` respectively, on the MATLAB command line. See the image for more detail.

The workers in a parallel pool can be used interactively and communicate with each other during the lifetime of the job. You can view your `parpool` jobs in the "Job Monitor" on page 5-24. While these pool workers are reserved for your interactive use, they are not available to other users. You can have only one parallel pool at a time from a MATLAB client session. In MATLAB, the current parallel pool is represented by a parallel.Pool object.
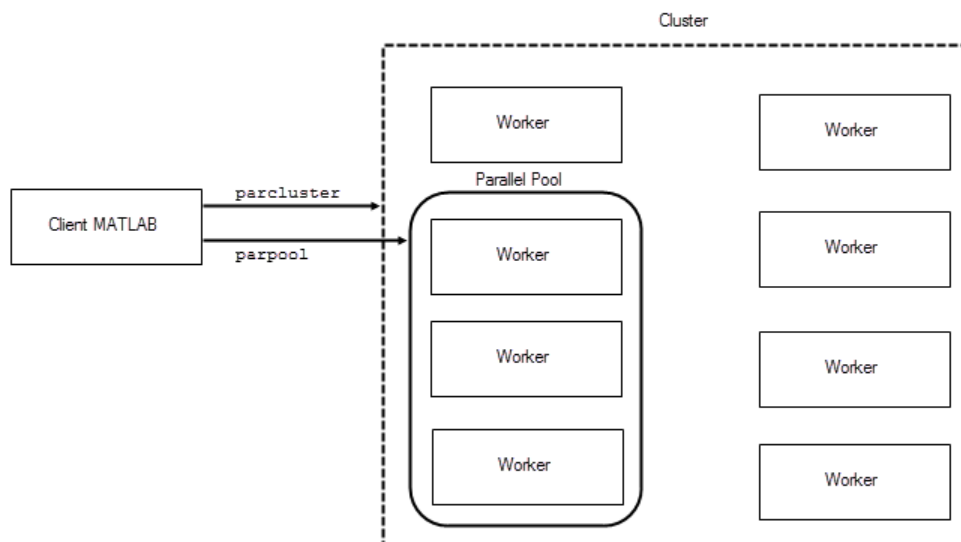
Automatically Start and Stop a Parallel Pool

By default, a parallel pool starts automatically when needed by certain parallel language features. Many functions can automatically start a parallel pool, including:

* `parfor`

- spmd
- distributed
- Composite
- parfeval
- parfevalOnAll
- gcp
- mapreduce
- mapreducer
- tall
- ticBytes and tocBytes

Your parallel preferences specify which cluster the pool runs on, and the preferred number of workers in the pool. To access your preferences, on the **Home** tab, in the **Environment** section, click **Parallel > Parallel Preferences**.

Alternative Ways to Start and Stop Pools

In your parallel preferences, you can turn off the option for the pool to open or close automatically. If you choose not to have the pool open automatically, you can control the pool with the following techniques.

Control the Parallel Pool from the MATLAB Desktop

You can use the parallel status indicator in the lower left corner of the MATLAB desktop to start a parallel pool manually.
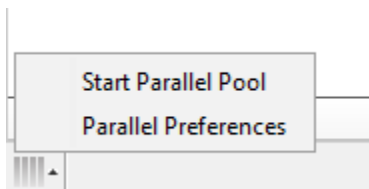
In MATLAB Online, the parallel status indicator is not visible by default. You must start a parallel pool first by using parpool or any of the functions that automatically start a parallel pool.

Click the indicator icon, and select **Start Parallel Pool**. The pool size and cluster are specified by your parallel preferences and default cluster. Your default cluster is indicated by a check mark on the **Parallel > Default Cluster** menu.

The menu options are different when a pool is running. You can:

- View the number of workers and cluster name
- Change the time until automatic shut-down
- Shut down the parallel pool
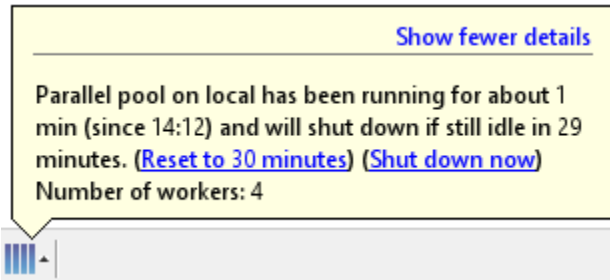
To stop a pool, you can also select **Shut Down Parallel Pool**.
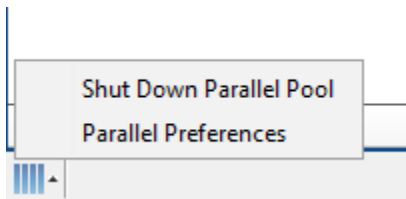
Programming Interface

Start a Parallel Pool

You can start and stop a parallel pool programmatically by using default settings or specifying alternatives.

To open a parallel pool based on your preference settings:

```
parpool
```

To open a pool of a specific size:

```
parpool(4)
```

To use a cluster other than your default and specify where the pool runs:

```
parpool('MyCluster',4)
```

You can run a parallel pool on different parallel environments. For more information, see "Choose Between Thread-Based and Process-Based Environments" on page 2-60.

Shut Down a Parallel Pool

To get the current parallel pool and use that object when you want to shut down the pool:

```
p = gcp;
delete(p)
```

Ensure That No Parallel Pool Is Running

When you issue the command `gcp` without arguments, you might inadvertently open a pool. To avoid this problem:

```
delete(gcp('nocreate'))
```

Pool Size and Cluster Selection

There are several places to specify pool size. Several factors might limit the size of a pool. The actual size of your parallel pool is determined by the combination of the following:

1 Licensing or cluster size

   The maximum limit on the number of workers in a pool is restricted by the number of workers in your cluster. This limit might be determined by the number of MATLAB Parallel Server licenses available. In the case of MATLAB Job Scheduler, the limit might be determined by the number of workers running in the cluster. A local cluster running on the client machine requires no licensing beyond the one for Parallel Computing Toolbox. The limit on the number of workers is high enough to support the range of known desktop hardware.

2 Cluster profile number of workers (`NumWorkers`)

   A cluster object can set a hard limit on the number of workers, which you specify in the cluster profile. Even if you request more workers at the command line or in your preferences, you cannot exceed the limit set in the applicable profile. Attempting to exceed this number generates an error.

3 Command-line argument

   If you specify a pool size at the command line, you override the setting of your preferences. This value must fall within the limits of the applicable cluster profile.

4 Parallel preferences

   If you do not specify a pool size at the command line, MATLAB attempts to start a pool with size determined by your parallel preferences. This value is a *preference*, not a requirement or a request for a specific number of workers. So if a pool cannot start with as many workers as called for in your preferences, you get a smaller pool without any errors. You can set the value of the **Preferred number of workers** to a large number, so that it never limits the size of the pool that is created. If you need an exact number of workers, specify the number at the command line.

For selection of the cluster on which the pool runs, precedence is determined by the following.

1 The command-line cluster object argument overrides the default profile setting and uses the cluster identified by the profile `'MyProfile'`.

```
c = parcluster('MyProfile');
p = parpool(c);
```

2 The cluster is specified in the default profile.

```
p = parpool;
```

See Also
`delete` | `distributed` | `gcp` | `parcluster` | `parfeval` | `parfor` | `parpool` | `spmd`

Related Examples
- "Run MATLAB Functions with Automatic Parallel Support" on page 1-19
- "Scale Up from Desktop to Cluster" on page 9-88

More About

Choose Between Thread-Based and Process-Based Environments

With Parallel Computing Toolbox, you can run your parallel code in different parallel environments, such as thread-based or process-based environments. These environments offer different advantages.

Note that thread-based environments support only a subset of the MATLAB functions available for process workers. If you are interested in a function that is not supported, let the MathWorks Technical Support team know. For more information on support, see "Check Support for Thread-Based Environment" on page 2-67.

Select Parallel Environment

Depending on the type of parallel environment you select, features run on either process workers or thread workers. To decide which environment is right for you, consult the following diagram and table.

Do you want to run your code on multiple machines?

Yes → Prototype your code using process-based local environment → Use remote cluster environment (process-based)

No → Can you run your code in a thread-based pool without changes?

Yes → Use thread-based local pool

No → Are you willing to make code changes for potential benefits?

Yes → Measure data transfer on a process-based local pool with ticBytes and tocBytes → Does your code transfer large amounts of data?

Yes → Use thread-based local pool

No → Use process-based local pool

No → Use process-based local pool

- To use parallel pool features, such as `parfor` or `parfeval`, create a parallel pool in the chosen environment by using the `parpool` function.

Environment	Recommendation	Example
Thread-based environment on local machine	Use this setup for reduced memory usage, faster scheduling, and lower data transfer costs.	`parpool('threads')` **Note** If you choose `'threads'`, check that your code is supported. For more information, see "Check Support for Thread-Based Environment" on page 2-67. To find out if you can get sufficient benefit from a thread-based pool, measure data transfer in a process-based pool with `ticBytes` and `tocBytes`. If the data transfer is large, such as above 100 MB, then use `'threads'`.
Process-based environment on local machine	Use this setup for most use cases and for prototyping before scaling to clusters or clouds.	`parpool('local')`
Process-based environment on remote cluster	Use this setup to scale up your computations.	`parpool('MyCluster')` where `MyCluster` is the name of a cluster profile.

- To use cluster features, such as `batch`, create a cluster object in the chosen environment by using the `parcluster` function. Note that cluster features are supported only in process-based environments.

Environment	Recommendation	Example
Process-based environment on local machine	Use this setup if you have sufficient local resources, or to prototype before scaling to clusters or clouds.	`parcluster('local')`
Process-based environment on remote cluster	Use this setup to scale up your computations.	`parcluster('MyCluster')` where `MyCluster` is the name of a cluster profile.

Recommendation Defaulting to process-based environments is recommended.

- They support the full parallel language.
- They are backwards compatible with previous releases.

- They are more robust in the event of crashes.
- External libraries do not need to be thread-safe.

Choose thread-based environments when:

- Your parallel code is supported by thread-based environments.
- You want reduced memory usage, faster scheduling and lower data transfer costs.

Compare Process Workers and Thread Workers

The following shows a performance comparison between process workers and thread workers for an example that leverages the efficiency of thread workers.

Create some data.

```
X = rand(10000, 10000);
```

Create a parallel pool of process workers.

```
pool = parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Time the execution and measure data transfer of some parallel code. For this example, use a `parfeval` execution.

```
ticBytes(pool);
tProcesses = timeit(@() fetchOutputs(parfeval(@sum,1,X,'all')))
tocBytes(pool)
```

```
tProcesses = 3.9060
```

	BytesSentToWorkers	BytesReceivedFromWorkers
1	0	0
2	0	0
3	0	0
4	0	0
5	5.6e+09	16254
6	0	0
Total	5.6e+09	16254

Note that the data transfer is significant. To avoid incurring data transfer costs, you can use thread workers. Delete the current parallel pool and create a thread-based parallel pool.

```
delete(pool);
pool = parpool('threads');
```

Time how long the same code takes to run.

```
tThreads = timeit(@() fetchOutputs(parfeval(@sum,1,X,'all')))
```

```
tThreads = 0.0232
```

2-63

Compare the times.

```
fprintf('Without data transfer, this example is %.2fx faster.\n', tProcesses/tThreads)
```

```
Without data transfer, this example is 168.27x faster.
```

Thread workers outperform process workers because thread workers can use the data X without copying it, and they have less scheduling overhead.

Solve Optimization Problem in Parallel on Process-Based and Thread-Based Pool

This example shows how to use a process-based and thread-based pool to solve an optimization problem in parallel. Thread-based pools are optimized for less data transfer, faster scheduling, and reduced memory usage, so they can result in a performance gain in your applications.

Problem Description

The problem is to change the position and angle of a cannon to fire a projectile as far as possible beyond a wall. The cannon has a muzzle velocity of 300 m/s. The wall is 20 m high. If the cannon is too close to the wall, it fires at too steep an angle, and the projectile does not travel far enough. If the cannon is too far from the wall, the projectile does not travel far enough. For full problem details, see "Optimize an ODE in Parallel" (Global Optimization Toolbox) or the latter part of the video Surrogate Optimization.

MATLAB Problem Formulation

To solve the problem, call the `patternsearch` solver from Global Optimization Toolbox. The objective function is in the `cannonobjective` helper function, which calculates the distance the projectile lands beyond the wall for a given position and angle. The constraint is in the `cannonconstraint` helper function, which calculates whether the projectile hits the wall, or even reaches the wall before hitting the ground. The helper functions are in separate files that you can view when you run this example.

Set the following inputs for the `patternsearch` solver. Note that, to use Parallel Computing Toolbox, you must set `'UseParallel'` to `true` in the optimization options.

```
lb = [-200;0.05];
ub = [-1;pi/2-.05];
x0 = [-30,pi/3];
opts = optimoptions('patternsearch',...
    'UseCompletePoll', true, ...
    'Display','off',...
    'UseParallel',true);
% No linear constraints, so set these inputs to empty:
A = [];
b = [];
Aeq = [];
beq = [];
```

Solve on Process-Based Pool

For comparison, solve the problem on a process-based parallel pool first.

Start a parallel pool of process workers.

```
p = parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

To reproduce the same computations later, seed the random generator with the default value.

```
rng default;
```

Use a loop to solve the problem several times and average the results.

```
tProcesses = zeros(5,1);
for repetition = 1:numel(tProcesses)
    tic
    [xsolution,distance,eflag,outpt] = patternsearch(@cannonobjective,x0, ...
        A,b,Aeq,beq,lb,ub,@cannonconstraint,opts);
    tProcesses(repetition) = toc;
end
tProcesses = mean(tProcesses)
```

```
tProcesses = 2.7677
```

To prepare for the comparison with a thread-based pool, delete the current parallel pool.

```
delete(p);
```

Solve on Thread-Based Pool

Start a parallel pool of thread workers.

```
p = parpool('threads');
```

```
Starting parallel pool (parpool) ...
Connected to the parallel pool (number of workers: 6).
```

Restore the random number generator to default settings and run the same code as before.

```
rng default
tThreads = zeros(5,1);
for repetition = 1:numel(tThreads)
    tic
    [xsolution,distance,eflag,outpt] = patternsearch(@cannonobjective,x0, ...
        A,b,Aeq,beq,lb,ub,@cannonconstraint,opts);
    tThreads(repetition) = toc;
end
tThreads = mean(tThreads)
```

```
tThreads = 1.5790
```

Compare the performance of thread workers and process workers.

```
fprintf('In this example, thread workers are %.2fx faster than process workers.\n', tProcesses/t
```

```
In this example, thread workers are 1.75x faster than process workers.
```

Notice the performance gain due to the optimizations of the thread-based pool.

When you are done with computations, delete the parallel pool.

```
delete(p);
```

What Are Thread-Based Environments?

In thread-based environments, parallel language features run on workers that are backed by computing threads, which run code on cores on a machine. They differ from computing processes in that they coexist within the same process and can share memory.
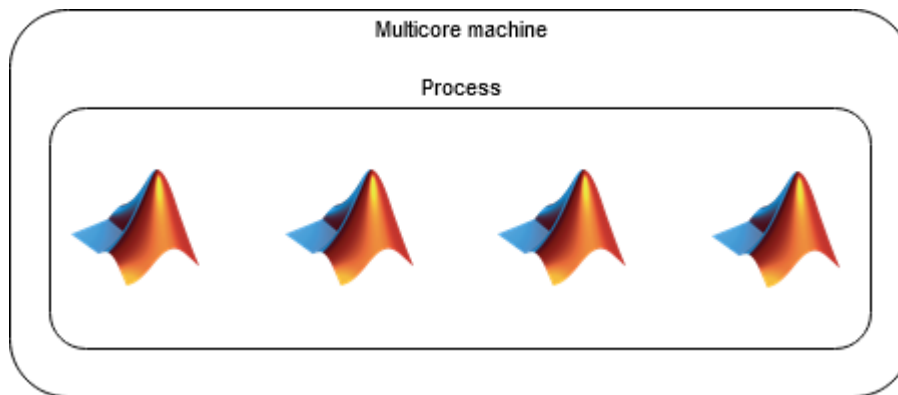
Thread-based environments have the following advantages over process-based environments.

- Because thread workers can share memory, they can access numeric data without copying, so they are more memory efficient.
- Communication between threads is less time consuming. Therefore, the overhead of scheduling a task or inter-worker communication is smaller.

When you use thread-based environments, keep the following considerations in mind.

- Check that your code is supported for a thread-based environment. For more information, see "Check Support for Thread-Based Environment" on page 2-67.
- If you are using external libraries from workers, then you must ensure that the library functions are thread-safe.

What are Process-Based Environments?

In process-based environments, parallel language features run on workers that are backed by computing processes, which run code on cores on a machine. They differ from computing threads in that they are independent of each other.
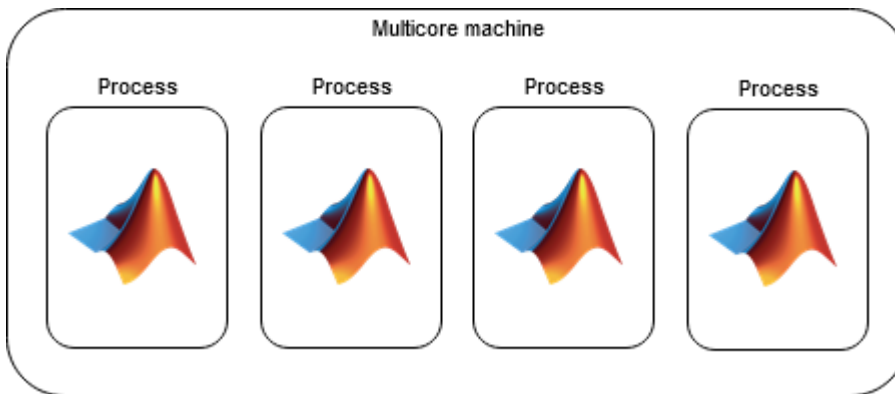
Process-based environments have the following advantages over thread-based environments.

- They support all language features and are backwards compatible with previous releases.
- They are more robust in the event of crashes. If a process worker crashes, then the MATLAB client does not crash. If a process worker crashes and your code does not use `spmd` or distributed arrays, then the rest of the workers can continue running.
- If you use external libraries from workers, then you do not need to pay attention to thread-safety.
- You can use cluster features, such as `batch`.

When you use a process-based environment, keep the following consideration in mind.

- If your code accesses files from workers, then you must use additional options, such as `'AttachedFiles'` or `'AdditionalPaths'`, to make the data accessible.

Check Support for Thread-Based Environment

Thread workers support only a subset of the MATLAB functions available for process workers. If you are interested in a function that is not supported, let the MathWorks Technical Support team know.

`parpool`, `parfor`, `parfeval`, `parfevalOnAll`, `tall`, and `parallel.pool.Constant` are supported, subject to the following limitations.

- A thread-based parallel pool does not have an associated cluster object.
- `afterEach` and `afterAll` are not supported.
- `FevalQueue` is not supported.
- Tall arrays do not support `write` and support only tabular text and in-memory inputs.

Other parallel language functionality, including `spmd`, `distributed`, and `parallel.pool.DataQueue`, is not supported.

The following core MATLAB functionality is supported on a thread worker.

- Functionality in Language Fundamentals except for functionality in Entering Commands.
- Functionality in Mathematics except any functions which plot to a figure.
- Functionality in Functions, Classes, and all functionality related to try, catch and `MException`.

In general, functionality that modifies or accesses things outside of the thread worker are not supported, including the following core MATLAB functionality.

- Functionality in Data Import and Analysis except for functionality in Tall Arrays, Preprocessing Data, and Descriptive Statistics.
- Functionality in Graphics and App Building.
- Functionality in External Language Interfaces.
- Functionality that interacts directly with the file system or MATLAB environment.

`gpuArray` is supported on a thread worker.

See Also

`parcluster` | `parpool`

Related Examples

- "Run Code on Parallel Pools" on page 2-55

Repeat Random Numbers in parfor-Loops

As described in "Control Random Number Streams on Workers" on page 5-29, each worker in a cluster working on the same job has an independent random number generator stream. By default, therefore, each worker in a pool, and each iteration in a `parfor`-loop has a unique, independent set of random numbers. Subsequent runs of the `parfor`-loop generate different numbers.

In a `parfor`-loop, you cannot control what sequence the iterations execute in, nor can you control which worker runs which iterations. So even if you reset the random number generators, the `parfor`-loop can generate the same values in a different sequence.

To reproduce the same set of random numbers in a `parfor`-loop each time the loop runs, you must control random generation by assigning a particular substream for each iteration.

First, create the stream you want to use, using a generator that supports substreams. Creating the stream as a `parallel.pool.Constant` allows all workers to access the stream.

```
sc = parallel.pool.Constant(RandStream('Threefry'))
```

Inside the `parfor`-loop, you can set the substream index by the loop index. This ensures that each iteration uses its particular set of random numbers, regardless of which worker runs that iteration or what sequence iterations run in.

```
r = zeros(1,16);
parfor i = 1:16
    stream = sc.Value;          % Extract the stream from the Constant
    stream.Substream = i;
    r(i) = rand(stream);
end
r

r =

  Columns 1 through 8

    0.3640    0.8645    0.0440    0.7564    0.5323    0.8075    0.2145    0.9128

  Columns 9 through 16

    0.4057    0.0581    0.5515    0.4347    0.3531    0.4677    0.8287    0.2312
```

See Also
`RandStream` | `rng`

More About
- "Control Random Number Streams on Workers" on page 5-29
- "Creating and Controlling a Random Number Stream"

Recommended System Limits for Macintosh and Linux

If you use a UNIX® system (Linux® or Macintosh), it is recommended that you adjust your operating system limits. Check and set limits with the `ulimit` or `limit` command, depending on your installation. Note that these commands might require root access.

System Limit	Recommended Value	Option (`ulimit`)	Option (`limit`)
Maximum number of user processes	23741	`-u`	`maxproc`
Maximum number of open file descriptors	4096	`-n`	`descriptors`

For example, these commands set the maximum number of user processes.

```
ulimit - u 23741
limit maxproc 23741
```

Changing a limit inside a shell affects only that shell and any subsequent MATLAB sessions you start there. To make this setting persistent system-wide, you must modify the relevant file.

* Linux – Modify the `limits.conf` file.
* Macintosh – Modify plist files, such as `limit.maxfiles.plist` and `limit.maxproc.plist`.

For assistance, check with your system administrator.

For more information on `ulimit`, `limit`, or `limits.conf`, see their man pages.

Without these settings, large parallel pools can error, hang, or lose workers during creation. These problems occur when MATLAB attempts to create more user processes or file handles than your operating system allows.

If you use a cluster of machines, you must set the maximum number of user processes for each machine.

See Also

More About

* "What Is a Parallel Pool?" on page 2-55

Single Program Multiple Data (spmd)

Run Single Programs on Multiple Data Sets

In this section...
"Introduction" on page 3-2
"When to Use spmd" on page 3-2
"Define an spmd Statement" on page 3-2
"Display Output" on page 3-4
"MATLAB Path" on page 3-4
"Error Handling" on page 3-4
"spmd Limitations" on page 3-4

Introduction

The single program multiple data (spmd) language construct allows seamless interleaving of serial and parallel programming. The `spmd` statement lets you define a block of code to run simultaneously on multiple workers. Variables assigned inside the `spmd` statement on the workers allow direct access to their values from the client by reference via *Composite* objects.

This chapter explains some of the characteristics of `spmd` statements and `Composite` objects.

When to Use spmd

The "single program" aspect of spmd means that the identical code runs on multiple workers. You run one program in the MATLAB client, and those parts of it labeled as `spmd` blocks run on the workers. When the spmd block is complete, your program continues running in the client.

The "multiple data" aspect means that even though the `spmd` statement runs identical code on all workers, each worker can have different, unique data for that code. So multiple data sets can be accommodated by multiple workers.

Typical applications appropriate for `spmd` are those that require running simultaneous execution of a program on multiple data sets, when communication or synchronization is required between the workers. Some common cases are:

- Programs that take a long time to execute — `spmd` lets several workers compute solutions simultaneously.
- Programs operating on large data sets — `spmd` lets the data be distributed to multiple workers.

For more information, see "Choose Between spmd, parfor, and parfeval" on page 3-15.

Define an spmd Statement

The general form of an `spmd` statement is:

```
spmd
    <statements>
end
```

Note If a parallel pool is not running, `spmd` creates a pool using your default cluster profile, if your parallel preferences are set accordingly.

The block of code represented by `<statements>` executes in parallel simultaneously on all workers in the parallel pool. If you want to limit the execution to only a portion of these workers, specify exactly how many workers to run on:

```
spmd (n)
    <statements>
end
```

This statement requires that `n` workers run the `spmd` code. `n` must be less than or equal to the number of workers in the open parallel pool. If the pool is large enough, but `n` workers are not available, the statement waits until enough workers are available. If `n` is 0, the `spmd` statement uses no workers, and runs locally on the client, the same as if there were not a pool currently running.

You can specify a range for the number of workers:

```
spmd (m,n)
    <statements>
end
```

In this case, the `spmd` statement requires a minimum of `m` workers, and it uses a maximum of `n` workers.

If it is important to control the number of workers that execute your `spmd` statement, set the exact number in the cluster profile or with the `spmd` statement, rather than using a range.

For example, create a random matrix on three workers:

```
spmd (3)
    R = rand(4,4);
end
```

Note All subsequent examples in this chapter assume that a parallel pool is open and remains open between sequences of `spmd` statements.

Unlike a `parfor`-loop, the workers used for an `spmd` statement each have a unique value for `labindex`. This lets you specify code to be run on only certain workers, or to customize execution, usually for the purpose of accessing unique data.

For example, create different sized arrays depending on `labindex`:

```
spmd (3)
    if labindex==1
        R = rand(9,9);
    else
        R = rand(4,4);
    end
end
```

Load unique data on each worker according to `labindex`, and use the same function on each worker to compute a result from the data:

```
spmd (3)
    labdata = load(['datafile_' num2str(labindex) '.ascii'])
    result = MyFunction(labdata)
end
```

The workers executing an `spmd` statement operate simultaneously and are aware of each other. As with a communicating job, you are allowed to directly control communications between the workers, transfer data between them, and use codistributed arrays among them.

For example, use a codistributed array in an `spmd` statement:

```
spmd (3)
    RR = rand(30, codistributor());
end
```

Each worker has a 30-by-10 segment of the codistributed array RR. For more information about codistributed arrays, see "Working with Codistributed Arrays" on page 4-4.

Display Output

When running an `spmd` statement on a parallel pool, all command-line output from the workers displays in the client Command Window. Because the workers are MATLAB sessions without displays, any graphical output (for example, figure windows) from the pool does not display at all.

MATLAB Path

All workers executing an `spmd` statement must have the same MATLAB search path as the client, so that they can execute any functions called in their common block of code. Therefore, whenever you use `cd`, `addpath`, or `rmpath` on the client, it also executes on all the workers, if possible. For more information, see the `parpool` reference page. When the workers are running on a different platform than the client, use the function `pctRunOnAll` to properly set the MATLAB path on all workers.

Error Handling

When an error occurs on a worker during the execution of an `spmd` statement, the error is reported to the client. The client tries to interrupt execution on all workers, and throws an error to the user.

Errors and warnings produced on workers are annotated with the worker ID (`labindex`) and displayed in the client's Command Window in the order in which they are received by the MATLAB client.

The behavior of `lastwarn` is unspecified at the end of an `spmd` if used within its body.

spmd Limitations

Nested Functions

Inside a function, the body of an `spmd` statement cannot reference a nested function. However, it can call a nested function by means of a variable defined as a function handle to the nested function.

Because the `spmd` body executes on workers, variables that are updated by nested functions called inside an `spmd` statement are not updated in the workspace of the outer function.

Nested spmd Statements

The body of an `spmd` statement cannot directly contain another `spmd`. However, it can call a function that contains another `spmd` statement. The inner `spmd` statement does not run in parallel in another parallel pool, but runs serially in a single thread on the worker running its containing function.

Nested parfor-Loops

An `spmd` statement cannot contain a `parfor`-loop, and the body of a `parfor`-loop cannot contain an `spmd` statement. The reason is that workers cannot start or access further parallel pools.

break, continue, and return Statements

The body of an `spmd` statement cannot contain `break`, `continue`, or `return` statements. Consider `parfeval` or `parfevalOnAll` instead of `spmd`, because you can use `cancel` on them.

Global and Persistent Variables

The body of an `spmd` statement cannot contain `global` or `persistent` variable declarations. The reason is that these variables are not synchronized between workers. You can use `global` or `persistent` variables within functions, but their value is only visible to the worker that creates them. Instead of `global` variables, it is a better practice to use function arguments to share values.

Anonymous Functions

The body of an `spmd` statement cannot define an anonymous function. However, it can reference an anonymous function by means of a function handle.

inputname Functions

Using `inputname` to return the workspace variable name corresponding to an argument number is not supported inside `spmd`. The reason is that `spmd` workers do not have access to the workspace of the MATLAB desktop. To work around this, call `inputname` before `spmd`, as shown in the following example.

```
a = 'a';
myFunction(a)

function X = myFunction(a)
name = inputname(1);
spmd
    X.(name) = labindex;
end
X = [X{:}];
end
```

load Functions

The syntaxes of `load` that do not assign to an output structure are not supported inside `spmd` statements. Inside `spmd`, always assign the output of `load` to a structure.

nargin or nargout Functions

The following uses are not supported inside `spmd` statements:

- Using `nargin` or `nargout` without a function argument
- Using `narginchk` or `nargoutchk` to validate the number of input or output arguments in a call to the function that is currently executing

The reason is that workers do not have access to the workspace of the MATLAB desktop. To work around this, call these functions before `spmd`.

```
myFunction('a','b')

function myFunction(a,b)
nin = nargin;
spmd
    X = labindex*nin;
end
end
```

P-Code Scripts

You can call P-code script files from within an `spmd` statement, but P-code scripts cannot contain an `spmd` statement. To work around this, use a P-code function instead of a P-code script.

ans Variable

References to the `ans` variable defined outside an `spmd` statement are not supported inside the `spmd` statement. Inside the body of an `spmd` statement, you must assign the `ans` variable before you use it.

See Also
spmd | Composite | parfeval | parfevalOnAll | parfor

More About
- "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49
- "Choose Between spmd, parfor, and parfeval" on page 3-15

Access Worker Variables with Composites

In this section...
"Introduction to Composites" on page 3-7
"Create Composites in spmd Statements" on page 3-7
"Variable Persistence and Sequences of spmd" on page 3-8
"Create Composites Outside spmd Statements" on page 3-9

Introduction to Composites

Composite objects in the MATLAB client session let you directly access data values on the workers. Most often you assigned these variables within `spmd` statements. In their display and usage, Composites resemble cell arrays. There are two ways to create Composites:

- Use the `Composite` function on the client. Values assigned to the Composite elements are stored on the workers.

- Define variables on workers inside an `spmd` statement. After the `spmd` statement, the stored values are accessible on the client as Composites.

Create Composites in spmd Statements

When you define or assign values to variables inside an `spmd` statement, the data values are stored on the workers.

After the `spmd` statement, those data values are accessible on the client as Composites. Composite objects resemble cell arrays, and behave similarly. On the client, a Composite has one element per worker. For example, suppose you create a parallel pool of three local workers and run an `spmd` statement on that pool:

```
parpool('local',3)

spmd  % Uses all 3 workers
    MM = magic(labindex+2); % MM is a variable on each worker
end
MM{1} % In the client, MM is a Composite with one element per worker

    8    1    6
    3    5    7
    4    9    2

MM{2}

   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
```

A variable might not be defined on every worker. For the workers on which a variable is not defined, the corresponding Composite element has no value. Trying to read that element throws an error.

```
spmd
    if labindex > 1
```

```
        HH = rand(4);
    end
end
HH

    Lab 1: No data
    Lab 2: class = double, size = [4  4]
    Lab 3: class = double, size = [4  4]
```

You can also set values of Composite elements from the client. This causes a transfer of data, storing the value on the appropriate worker even though it is not executed within an spmd statement:

```
MM{3} = eye(4);
```

In this case, MM must already exist as a Composite, otherwise MATLAB interprets it as a cell array.

Now when you do enter an spmd statement, the value of the variable MM on worker 3 is as set:

```
spmd
    if labindex == 3, MM, end
end

Lab 3:
    MM =
        1    0    0    0
        0    1    0    0
        0    0    1    0
        0    0    0    1
```

Data transfers from worker to client when you explicitly assign a variable in the client workspace using a Composite element:

```
M = MM{1} % Transfer data from worker 1 to variable M on the client

        8    1    6
        3    5    7
        4    9    2
```

Assigning an entire Composite to another Composite does not cause a data transfer. Instead, the client merely duplicates the Composite as a reference to the appropriate data stored on the workers:

```
NN = MM % Set entire Composite equal to another, without transfer
```

However, accessing a Composite's elements to assign values to other Composites *does* result in a transfer of data from the workers to the client, even if the assignment then goes to the same worker. In this case, NN must already exist as a Composite:

```
NN{1} = MM{1} % Transfer data to the client and then to worker
```

When finished, you can delete the pool:

```
delete(gcp)
```

Variable Persistence and Sequences of spmd

The values stored on the workers are retained between spmd statements. This allows you to use multiple spmd statements in sequence, and continue to use the same variables defined in previous spmd blocks.

The values are retained on the workers until the corresponding Composites are cleared on the client, or until the parallel pool is deleted. The following example illustrates data value lifespan with `spmd` blocks, using a pool of four workers:

```matlab
parpool('local',4)

spmd
    AA = labindex; % Initial setting
end
AA(:)  % Composite

    [1]
    [2]
    [3]
    [4]

spmd
    AA = AA * 2; % Multiply existing value
end
AA(:)  % Composite

    [2]
    [4]
    [6]
    [8]

clear AA % Clearing in client also clears on workers

spmd; AA = AA * 2; end   % Generates error

delete(gcp)
```

Create Composites Outside spmd Statements

The `Composite` function creates Composite objects without using an `spmd` statement. This might be useful to prepopulate values of variables on workers before an `spmd` statement begins executing on those workers. Assume a parallel pool is already running:

```matlab
PP = Composite()
```

By default, this creates a Composite with an element for each worker in the parallel pool. You can also create Composites on only a subset of the workers in the pool. See the Composite reference page for more details. The elements of the Composite can now be set as usual on the client, or as variables inside an `spmd` statement. When you set an element of a Composite, the data is immediately transferred to the appropriate worker:

```matlab
for ii = 1:numel(PP)
    PP{ii} = ii;
end
```

Distributing Arrays to Parallel Workers

In this section...
"Using Distributed Arrays to Partition Data Across Workers" on page 3-10
"Load Distributed Arrays in Parallel Using datastore" on page 3-10
"Alternative Methods for Creating Distributed and Codistributed Arrays" on page 3-12

Using Distributed Arrays to Partition Data Across Workers

Depending on how your data fits in memory, choose one of the following methods:

- If your data is currently in the memory of your local machine, you can use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool. This option can be useful for testing or before performing operations which significantly increase the size of your arrays, such as `repmat`.
- If your data does not fit in the memory of your local machine, but does fit in the memory of your cluster, you can use `datastore` with the `distributed` function to read data into the memory of the workers of a parallel pool.
- If your data does not fit in the memory of your cluster, you can use `datastore` with `tall` arrays to partition and process your data in chunks. See also "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46.

Load Distributed Arrays in Parallel Using datastore

If your data does not fit in the memory of your local machine, but does fit in the memory of your cluster, you can use `datastore` with the `distributed` function to create distributed arrays and partition the data among your workers.

This example shows how to create and load distributed arrays using `datastore`. Create a datastore using a tabular file of airline flight data. This data set is too small to show equal partitioning of the data over the workers. To simulate a large data set, artificially increase the size of the datastore using `repmat`.

```
files = repmat({'airlinesmall.csv'}, 10, 1);
ds = tabularTextDatastore(files);
```

Select the example variables.

```
ds.SelectedVariableNames = {'DepTime','DepDelay'};
ds.TreatAsMissing = 'NA';
```

Create a distributed table by reading the datastore in parallel. Partition the datastore with one partition per worker. Each worker then reads all data from the corresponding partition. The files must be in a shared location that is accessible by the workers.

```
dt = distributed(ds);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

Display summary information about the distributed table.

```
summary(dt)
```

```
Variables:

    DepTime: 1,235,230×1 double
        Values:

            min              1
            max           2505
            NaNs        23,510

    DepDelay: 1,235,230×1 double
        Values:

            min          -1036
            max           1438
            NaNs        23,510
```

Determine the size of the tall table.

```
size(dt)
```

```
ans =

    1235230            2
```

Return the first few rows of dt.

```
head(dt)
```

```
ans =

    DepTime     DepDelay
    _____    _____

      642          12
     1021           1
     2055          20
     1332          12
      629          -1
     1446          63
      928          -2
      859          -1
     1833           3
     1041           1
```

Finally, check how much data each worker has loaded.

```
spmd, dt, end
```

```
Lab 1:

  This worker stores dt2(1:370569,:).

        LocalPart: [370569×2 table]
     Codistributor: [1×1 codistributor1d]

Lab 2:

  This worker stores dt2(370570:617615,:).
```

```
            LocalPart: [247046×2 table]
        Codistributor: [1×1 codistributor1d]

Lab 3:

  This worker stores dt2(617616:988184,:).

            LocalPart: [370569×2 table]
        Codistributor: [1×1 codistributor1d]

Lab 4:

  This worker stores dt2(988185:1235230,:).

            LocalPart: [247046×2 table]
        Codistributor: [1×1 codistributor1d]
```

Note that the data is partitioned equally over the workers. For more details on `datastore`, see "What Is a Datastore?"

For more details about workflows for big data, see "Choose a Parallel Computing Solution" on page 1-16.

Alternative Methods for Creating Distributed and Codistributed Arrays

If your data fits in the memory of your local machine, you can use distributed arrays to partition the data among your workers. Use the `distributed` function to create a distributed array in the MATLAB client, and store its data on the workers of the open parallel pool. A distributed array is distributed in one dimension, and as evenly as possible along that dimension among the workers. You cannot control the details of distribution when creating a distributed array.

You can create a distributed array in several ways:

- Use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool.

- Use any of the `distributed` functions to directly construct a distributed array on the workers. This technique does not require that the array already exists in the client, thereby reducing client workspace memory requirements. Functions include `eye(___,'distributed')` and `rand(___,'distributed')`. For a full list, see the `distributed` object reference page.

- Create a codistributed array inside an `spmd` statement, and then access it as a distributed array outside the `spmd` statement. This technique lets you use distribution schemes other than the default.

The first two techniques do not involve `spmd` in creating the array, but you can use `spmd` to manipulate arrays created this way. For example:

Create an array in the client workspace, and then make it a distributed array.

```
parpool('local',2) % Create pool
W = ones(6,6);
W = distributed(W); % Distribute to the workers
spmd
    T = W*2; % Calculation performed on workers, in parallel.
             % T and W are both codistributed arrays here.
end
```

```
T              % View results in client.
whos           % T and W are both distributed arrays here.
delete(gcp) % Stop pool
```

Alternatively, you can use the `codistributed` function, which allows you to control more options such as dimensions and partitions, but is often more complicated. You can create a `codistributed` array by executing on the workers themselves, either inside an `spmd` statement or inside a communicating job. When creating a `codistributed` array, you can control all aspects of distribution, including dimensions and partitions.

The relationship between distributed and codistributed arrays is one of perspective. Codistributed arrays are partitioned among the workers from which you execute code to create or manipulate them. When you create a distributed array in the client, you can access it as a codistributed array inside an `spmd` statement. When you create a codistributed array in an `spmd` statement, you can access it as a distributed array in the client. Only `spmd` statements let you access the same array data from two different perspectives.

You can create a `codistributed` array in several ways:

- Use the `codistributed` function inside an `spmd` statement or a communicating job to codistribute data already existing on the workers running that job.

- Use any of the codistributed functions to directly construct a codistributed array on the workers. This technique does not require that the array already exists in the workers. Functions include `eye(___,'codistributed')` and `rand(___,'codistributed')`. For a full list, see the `codistributed` object reference page.

- Create a distributed array outside an `spmd` statement, then access it as a codistributed array inside the `spmd` statement running on the same parallel pool.

Create a codistributed array inside an `spmd` statement using a nondefault distribution scheme. First, define 1-D distribution along the third dimension, with 4 parts on worker 1, and 12 parts on worker 2. Then create a 3-by-3-by-16 array of zeros.

```
parpool('local',2) % Create pool
spmd
    codist = codistributor1d(3,[4,12]);
    Z = zeros(3,3,16,codist);
    Z = Z + labindex;
end
Z  % View results in client.
   % Z is a distributed array here.
delete(gcp) % Stop pool
```

For more details on codistributed arrays, see "Working with Codistributed Arrays" on page 4-4.

See Also
`codistributed` | `datastore` | `distributed` | `eye` | `rand` | `repmat` | `spmd` | `tall`

Related Examples
- "Run MATLAB Functions with Distributed Arrays" on page 4-18
- "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46
- "What Is a Datastore?"

More About

- "Datastore"
- "Tall Arrays for Out-of-Memory Data"

Choose Between spmd, parfor, and parfeval

Communicating Parallel Code

To run computations in parallel, you can use `parfor`, `parfeval`, `parfevalOnAll`, or `spmd`. Each construct relies on different parallel programming concepts. If you require workers to communicate throughout a computation, use `parfeval`, `parfevalOnAll`, or `spmd`.

- Use `parfeval` or `parfevalOnAll` if your code can be split into a set of tasks, where each task can depend on the output of other tasks.
- Use `spmd` if you require communication between workers during a computation.

Computations with `parfeval` are best represented as a graph, similar to a Kanban board with blocking. Generally, results are collected from workers after a computation is complete. You can collect results from execution of a `parfeval` operation by using `afterEach` or `afterAll`. You typically use the results in further calculations.

Computations with `spmd` are best represented by a flowchart, similar to a waterfall workflow. A pool worker executing `spmd` statements is called a lab. Results can be collected from labs during a computation. Sometimes, labs must communicate with other labs before they can finish their computation.

If you are unsure, ask yourself the following: **within my communicating parallel code, can each computation be completed without any communication between workers?** If yes, use `parfeval`. Otherwise, use `spmd`.

Synchronous and Asynchronous Work

When choosing between `parfor`, `parfeval`, and `spmd`, consider whether your calculation requires synchronization with the client.

`parfor` and `spmd` require synchronization, and therefore block you from running any new computations on the MATLAB client. `parfeval` does not require synchronization, so the client is free to pursue other work.

Compare Performance of Multithreading and ProcessPool

In this example, you compare how fast functions run on the client and on a `ProcessPool`. Some MATLAB functions make use of multithreading. Tasks that use these functions perform better on multiple threads than a single thread. Therefore, if you use these functions on a machine with many cores, a local cluster can perform worse than multithreading on the client.

The supporting function `clientFasterThanPool`, listed at the end of this example, returns `true` if multiple executions are performed faster on the client than a `parfor`-loop. The syntax is the same as `parfeval`: use a function handle as the first argument, the number of outputs as the second argument, and then give all required arguments for the function.

First, create a local `ProcessPool`.

```
p = parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Check how fast the `eig` function runs by using the `clientFasterThanPool` supporting function. Create an anonymous function with `eig` to represent your function call.

```
[~, t_client, t_pool] = clientFasterThanPool(@(N) eig(randn(N)), 0, 500)

t_client = 22.6243

t_pool = 4.9334
```

The parallel pool computes the answer faster than the client. Divide `t_client` by `maxNumCompThreads` to find the time taken per thread on the client.

```
t_client/maxNumCompThreads

ans = 3.7707
```

Workers are single threaded by default. The result indicates that the time taken per thread is similar on both the client and the pool, as the value of `t_pool` is roughly 1.5 times the value of `t_client/maxNumCompThreads`. The `eig` function does not benefit from multithreading.

Next, check how fast the `lu` function runs by using the `clientFasterThanPool` supporting function.

```
[~, t_client, t_pool] = clientFasterThanPool(@(N) lu(randn(N)), 0, 500)

t_client = 1.0225

t_pool = 0.4693
```

The parallel pool typically computes the answer faster than the client if your local machine has four or more cores. Divide `t_client` by `maxNumCompThreads` to find the time taken per thread.

```
t_client/maxNumCompThreads

ans = 0.1704
```

This result indicates that the time taken per thread is much less on the client than the pool, as the value of `t_pool` is roughly 3 times the value of `t_client/maxNumCompThreads`. Each thread is used for less computational time, indicating that `lu` uses multithreading.

Define Helper Function

The supporting function `clientFasterThanPool` checks whether a computation is faster on the client than on a parallel pool. It takes as input a function handle `fcn` and a variable number of input arguments (`in1, in2, ...`). `clientFasterThanPool` executes `fcn(in1, in2, ...)` on both the client and the active parallel pool. As an example, if you wish to test `rand(500)`, your function handle must be in the following form:

```
fcn = @(N) rand(N);
```

Then, use `clientFasterThanPool(fcn,500)`.

```
function [result, t_multi, t_single] = clientFasterThanPool(fcn,numout,varargin)
    % Preallocate cell array for outputs
    outputs = cell(numout);

    % Client
    tic
```

```
    for i = 1:200
        if numout == 0
            fcn(varargin{:});
        else
            [outputs{1:numout}] = fcn(varargin{:});
        end
    end
    t_multi = toc;

    % Parallel pool
    vararginC = parallel.pool.Constant(varargin);
    tic
    parfor i = 1:200
        % Preallocate cell array for outputs
        outputs = cell(numout);

        if numout == 0
            fcn(vararginC.Value{:});
        else
            [outputs{1:numout}] = fcn(vararginC.Value{:});
        end
    end
    t_single = toc;

    % If multhreading is quicker, return true
    result = t_single > t_multi;
end
```

Compare Performance of parfor, parfeval, and spmd

Using `spmd` can be slower or faster than using `parfor`-loops or `parfeval`, depending on the type of computation. Overhead affects the relative performance of `parfor`-loops, `parfeval`, and `spmd`.

For a set of tasks, `parfor` and `parfeval` typically perform better than `spmd` under these conditions.

- The computational time taken per task is not deterministic.
- The computational time taken per task is not uniform.
- The data returned from each task is small.

Use `parfeval` when:

- You want to run computations in the background.
- Each task is dependent on other tasks.

In this example, you examine the speed at which matrix operations can be performed when using a `parfor`-loop, `parfeval`, and `spmd`.

First, create a local parallel pool `p`.

```
p = parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Compute Random Matrices

Examine the speed at which random matrices can be generated by using a `parfor`-loop, `parfeval`, and `spmd`. Set the number of trials (`n`) and the matrix size (for an `m`-by-`m` matrix). Increasing the number of trials improves the statistics used in later analysis, but does not affect the calculation itself.

```
m =    1000   ═══◇════════════    ;
n =    20     ═══◇════════════  ;
```

Then, use a `parfor`-loop to execute `rand(m)` once for each worker. Time each of the m trials.

```matlab
parforTime = zeros(n,1);
for i = 1:n
    tic;
    mats = cell(1,p.NumWorkers);
    parfor N = 1:p.NumWorkers
      mats{N} = rand(m);
    end
    parforTime(i) = toc;
end
```

Next, use `parfeval` to execute `rand(m)` once for each worker. Time each of the m trials.

```matlab
parfevalTime = zeros(n,1);
for i = 1:n
    tic;
    f(1:p.NumWorkers) = parallel.FevalFuture;
    for N = 1:p.NumWorkers
      f(N) = parfeval(@rand,1,m);
    end
    mats = fetchOutputs(f, "UniformOutput", false)';
    parfevalTime(i) = toc;
    clear f
end
```

Finally, use `spmd` to execute `rand(m)` once for each lab. For details on labs and how to execute commands on them with `spmd`, see "Run Single Programs on Multiple Data Sets" on page 3-2. Time each of the m trials.

```matlab
spmdTime = zeros(n,1);
for i = 1:n
    tic;
    spmd
        e = rand(m);
    end
    eigenvals = {e{:}};
    spmdTime(i) = toc;
end
```

Use `rmoutliers` to remove the outliers from each of the trials. Then, use `boxplot` to compare the times.

```matlab
% Hide outliers
boxData = rmoutliers([parforTime parfevalTime spmdTime]);
```

```matlab
% Plot data
boxplot(boxData, 'labels',{'parfor','parfeval','spmd'}, 'Symbol','')
ylabel('Time (seconds)')
title('Make n random matrices (m by m)')
```
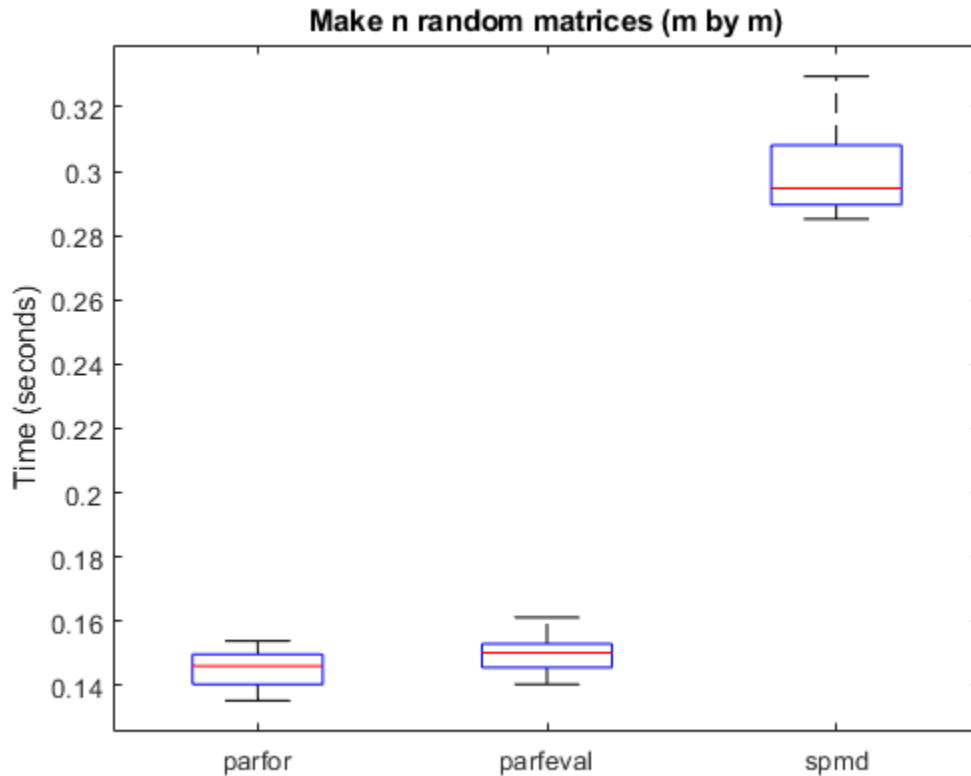

Typically, `spmd` requires more overhead per evaluation than `parfor` or `parfeval`. Therefore, in this case, using a `parfor`-loop or `parfeval` is more efficient.

Compute Sum of Random Matrices

Next, compute the sum of random matrices. You can do this by using a reduction variable with a `parfor-loop`, a sum after computations with `parfeval`, or `gplus` with `spmd`. Again, set the number of trials (n) and the matrix size (for an `m`-by-`m` matrix).

m = 1000   ⬦━━━━━━━━ ;

n = 20   ━━⬦━━━━━━ ;

Then, use a `parfor`-loop to execute `rand(m)` once for each worker. Compute the sum with a reduction variable. Time each of the `m` trials.

```matlab
parforTime = zeros(n,1);
for i = 1:n
    tic;
    result = 0;
    parfor N = 1:p.NumWorkers
        result = result + rand(m);
```

```
        end
        parforTime(i) = toc;
end
```

Next, use `parfeval` to execute `rand(m)` once for each worker. Use `fetchOutputs` on all of the matrices, then use `sum`. Time each of the m trials.

```
parfevalTime = zeros(n,1);
for i = 1:n
    tic;
    f(1:p.NumWorkers) = parallel.FevalFuture;
    for N = 1:p.NumWorkers
      f(N) = parfeval(@rand,1,m);
    end
    result = sum(fetchOutputs(f));
    parfevalTime(i) = toc;
    clear f
end
```

Finally, use `spmd` to execute `rand(m)` once for each lab. Use `gplus` to sum all of the matrices. To send the result only to the first lab, set the optional `targetlab` argument to `1`. Time each of the m trials.

```
spmdTime = zeros(n,1);
for i = 1:n
    tic;
    spmd
        r = gplus(rand(m), 1);
    end
    result = r{1};
    spmdTime(i) = toc;
end
```

Use `rmoutliers` to remove the outliers from each of the trials. Then, use `boxplot` to compare the times.

```
% Hide outliers
boxData = rmoutliers([parforTime parfevalTime spmdTime]);

% Plot data
boxplot(boxData, 'labels',{'parfor','parfeval','spmd'}, 'Symbol','')
ylabel('Time (seconds)')
title('Sum of n random matrices (m by m)')
```

Sum of n random matrices (m by m)
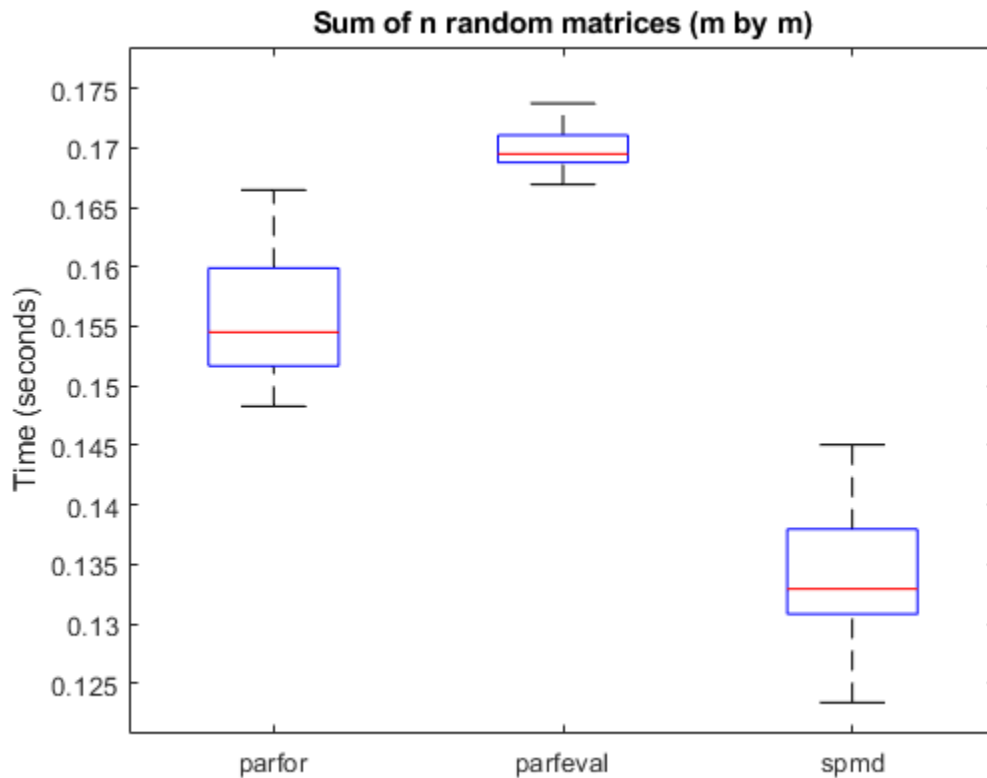
For this calculation, `spmd` is significantly faster than a `parfor`-loop or `parfeval`. When you use reduction variables in a `parfor`-loop, you broadcast the result of each iteration of the `parfor`-loop to all of the workers. By contrast, `spmd` calls `gplus` only once to do a global reduction operation, requiring less overhead. As such, the overhead for the reduction part of the calculation is $O(n^2)$ for `spmd`, and $O(mn^2)$ for `parfor`.

See Also
parfeval | parfor | spmd

More About
*   "Run Single Programs on Multiple Data Sets" on page 3-2
*   "Distribute Arrays and Run SPMD" on page 1-12

Math with Codistributed Arrays

This chapter describes the distribution or partition of data across several workers, and the functionality provided for operations on that data in `spmd` statements, communicating jobs, and pmode. The sections are as follows.

Nondistributed Versus Distributed Arrays

In this section...
"Introduction" on page 4-2
"Nondistributed Arrays" on page 4-2
"Codistributed Arrays" on page 4-3

Introduction

Many built-in data types and data structures supported by MATLAB software are also supported in the MATLAB parallel computing environment. This includes arrays of any number of dimensions containing numeric, character, logical values, cells, or structures. In addition to these basic building blocks, the MATLAB parallel computing environment also offers different *types* of arrays.

Nondistributed Arrays

When you create a nondistributed array, MATLAB constructs a separate array in the workspace of each worker, using the same variable name on all workers. Any operation performed on that variable affects all individual arrays assigned to it. If you display from worker 1 the value assigned to this variable, all workers respond by showing the array of that name that resides in their workspace.

The state of a nondistributed array depends on the value of that array in the workspace of each worker:

- "Replicated Arrays" on page 4-2
- "Variant Arrays" on page 4-2
- "Private Arrays" on page 4-3

Replicated Arrays

A *replicated array* resides in the workspaces of all workers, and its size and content are identical on all workers. When you create the array, MATLAB assigns it to the same variable on all workers. If you display in spmd the value assigned to this variable, all workers respond by showing the same array.

```
spmd, A = magic(3), end

  WORKER 1        WORKER 2        WORKER 3        WORKER 4
                |               |               |
8   1   6   |   8   1   6   |   8   1   6   |   8   1   6
3   5   7   |   3   5   7   |   3   5   7   |   3   5   7
4   9   2   |   4   9   2   |   4   9   2   |   4   9   2
```

Variant Arrays

A *variant array* also resides in the workspaces of all workers, but its content differs on one or more workers. When you create the array, MATLAB assigns a different value to the same variable on all workers. If you display the value assigned to this variable, all workers respond by showing their version of the array.

```
spmd, A = magic(3) + labindex - 1, end
```

```
 WORKER 1         WORKER 2         WORKER 3         WORKER 4
           |                |                |
 8   1   6 |  9   2   7 | 10   3   8 | 11   4   9
 3   5   7 |  4   6   9 |  5   7   9 |  6   8  10
 4   9   2 |  5  10   3 |  6  11   4 |  7  12   5
```

A replicated array can become a variant array when its value becomes unique on each worker.

```
spmd
    B = magic(3);       %replicated on all workers
    B = B + labindex;   %now a variant array, different on each worker
end
```

Private Arrays

A *private array* is defined on one or more, but not all workers. You could create this array by using `labindex` in a conditional statement, as shown here:

```
spmd
    if labindex >= 3, A = magic(3) + labindex - 1, end
end
```

```
 WORKER 1         WORKER 2          WORKER 3          WORKER 4
            |                |                 |
   A is     |     A is       | 10   3   8 | 11   4   9
 undefined  |  undefined     |  5   7   9 |  6   8  10
            |                |  6  11   4 |  7  12   5
```

Codistributed Arrays

With replicated and variant arrays, the full content of the array is stored in the workspace of each worker. *Codistributed arrays*, on the other hand, are partitioned into segments, with each segment residing in the workspace of a different worker. Each worker has its own array segment to work with. Reducing the size of the array that each worker has to store and process means a more efficient use of memory and faster processing, especially for large data sets.

This example distributes a 3-by-10 replicated array A across four workers. The resulting array D is also 3-by-10 in size, but only a segment of the full array resides on each worker.

```
spmd
    A = [11:20; 21:30; 31:40];
    D = codistributed(A);
    getLocalPart(D)
end
```

```
  WORKER 1         WORKER 2      WORKER 3     WORKER 4
            |                |            |
 11  12  13 | 14  15  16 | 17  18 | 19  20
 21  22  23 | 24  25  26 | 27  28 | 29  30
 31  32  33 | 34  35  36 | 37  38 | 39  40
```

For more details on using codistributed arrays, see "Working with Codistributed Arrays" on page 4-4.

Working with Codistributed Arrays

How MATLAB Software Distributes Arrays

When you distribute an array to a number of workers, MATLAB software partitions the array into segments and assigns one segment of the array to each worker. You can partition a two-dimensional array horizontally, assigning columns of the original array to the different workers, or vertically, by assigning rows. An array with N dimensions can be partitioned along any of its N dimensions. You choose which dimension of the array is to be partitioned by specifying it in the array constructor command.

For example, to distribute an 80-by-1000 array to four workers, you can partition it either by columns, giving each worker an 80-by-250 segment, or by rows, with each worker getting a 20-by-1000 segment. If the array dimension does not divide evenly over the number of workers, MATLAB partitions it as evenly as possible.

The following example creates an 80-by-1000 replicated array and assigns it to variable A. In doing so, each worker creates an identical array in its own workspace and assigns it to variable A, where A is local to that worker. The second command distributes A, creating a single 80-by-1000 array D that spans all four workers. Worker 1 stores columns 1 through 250, worker 2 stores columns 251 through 500, and so on. The default distribution is by the last nonsingleton dimension, thus, columns in this case of a 2-dimensional array.

```
spmd
  A = zeros(80, 1000);
  D = codistributed(A)
end

    Lab 1: This lab stores D(:,1:250).
    Lab 2: This lab stores D(:,251:500).
    Lab 3: This lab stores D(:,501:750).
    Lab 4: This lab stores D(:,751:1000).
```

Each worker has access to all segments of the array. Access to the local segment is faster than to a remote segment, because the latter requires sending and receiving data between workers and thus takes more time.

How MATLAB Displays a Codistributed Array

For each worker, the MATLAB Parallel Command Window displays information about the codistributed array, the local portion, and the codistributor. For example, an 8-by-8 identity matrix codistributed among four workers, with two columns on each worker, displays like this:

```
>> spmd
II = eye(8,'codistributed')
end
Lab 1:
  This lab stores II(:,1:2).
         LocalPart: [8x2 double]
     Codistributor: [1x1 codistributor1d]
Lab 2:
  This lab stores II(:,3:4).
         LocalPart: [8x2 double]
     Codistributor: [1x1 codistributor1d]
Lab 3:
  This lab stores II(:,5:6).
         LocalPart: [8x2 double]
     Codistributor: [1x1 codistributor1d]
Lab 4:
  This lab stores II(:,7:8).
         LocalPart: [8x2 double]
     Codistributor: [1x1 codistributor1d]
```

To see the actual data in the local segment of the array, use the `getLocalPart` function.

How Much Is Distributed to Each Worker

In distributing an array of `N` rows, if `N` is evenly divisible by the number of workers, MATLAB stores the same number of rows (`N/numlabs`) on each worker. When this number is not evenly divisible by the number of workers, MATLAB partitions the array as evenly as possible.

MATLAB provides codistributor object properties called `Dimension` and `Partition` that you can use to determine the exact distribution of an array. See "Indexing into a Codistributed Array" on page 4-11 for more information on indexing with codistributed arrays.

Distribution of Other Data Types

You can distribute arrays of any MATLAB built-in data type, and also numeric arrays that are complex or sparse, but not arrays of function handles or object types.

Creating a Codistributed Array

You can create a codistributed array in any of the following ways:

- "Partitioning a Larger Array" on page 4-6 — Start with a large array that is replicated on all workers, and partition it so that the pieces are distributed across the workers. This is most useful when you have sufficient memory to store the initial replicated array.
- "Building from Smaller Arrays" on page 4-6 — Start with smaller variant or replicated arrays stored on each worker, and combine them so that each array becomes a segment of a larger codistributed array. This method reduces memory requirements as it lets you build a codistributed array from smaller pieces.

- "Using MATLAB Constructor Functions" on page 4-7 — Use any of the MATLAB constructor functions like `rand` or `zeros` with a codistributor object argument. These functions offer a quick means of constructing a codistributed array of any size in just one step.

Partitioning a Larger Array

If you have a large array already in memory that you want MATLAB to process more quickly, you can partition it into smaller segments and distribute these segments to all of the workers using the `codistributed` function. Each worker then has an array that is a fraction the size of the original, thus reducing the time required to access the data that is local to each worker.

As a simple example, the following line of code creates a 4-by-8 replicated matrix on each worker assigned to the variable A:

```
spmd, A = [11:18; 21:28; 31:38; 41:48], end
A =
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48
```

The next line uses the `codistributed` function to construct a single 4-by-8 matrix D that is distributed along the second dimension of the array:

```
spmd
    D = codistributed(A);
    getLocalPart(D)
end

1: Local Part  | 2: Local Part  | 3: Local Part  | 4: Local Part
    11    12   |    13    14    |    15    16    |    17    18
    21    22   |    23    24    |    25    26    |    27    28
    31    32   |    33    34    |    35    36    |    37    38
    41    42   |    43    44    |    45    46    |    47    48
```

Arrays A and D are the same size (4-by-8). Array A exists in its full size on each worker, while only a segment of array D exists on each worker.

```
spmd, size(A), size(D), end
```

Examining the variables in the client workspace, an array that is codistributed among the workers inside an `spmd` statement, is a distributed array from the perspective of the client outside the `spmd` statement. Variables that are not codistributed inside the spmd, are Composites in the client outside the spmd.

```
whos
  Name      Size          Bytes  Class

  A         1x4             613  Composite
  D         4x8             649  distributed
```

See the `codistributed` function reference page for syntax and usage information.

Building from Smaller Arrays

The `codistributed` function is less useful for reducing the amount of memory required to store data when you first construct the full array in one workspace and then partition it into distributed

segments. To save on memory, you can construct the smaller pieces (local part) on each worker first, and then use `codistributed.build` to combine them into a single array that is distributed across the workers.

This example creates a 4-by-250 variant array A on each of four workers and then uses `codistributor` to distribute these segments across four workers, creating a 4-by-1000 codistributed array. Here is the variant array, A:

```
spmd
  A = [1:250; 251:500; 501:750; 751:1000] + 250 * (labindex - 1);
end

    WORKER 1             WORKER 2             WORKER 3
  1    2 ... 250 |   251   252 ... 500 |   501   502 ... 750 | etc.
251  252 ... 500 |   501   502 ... 750 |   751   752 ...1000 | etc.
501  502 ... 750 |   751   752 ...1000 |  1001  1002 ...1250 | etc.
751  752 ...1000 |  1001  1002 ...1250 |  1251  1252 ...1500 | etc.
                 |                     |                     |
```

Now combine these segments into an array that is distributed by the first dimension (rows). The array is now 16-by-250, with a 4-by-250 segment residing on each worker:

```
spmd
  D = codistributed.build(A, codistributor1d(1,[4 4 4 4],[16 250]))
end
Lab 1:
    This lab stores D(1:4,:).
          LocalPart: [4x250 double]
      Codistributor: [1x1 codistributor1d]

whos
  Name        Size            Bytes  Class

  A           1x4               613  Composite
  D           16x250            649  distributed
```

You could also use replicated arrays in the same fashion, if you wanted to create a codistributed array whose segments were all identical to start with. See the `codistributed` function reference page for syntax and usage information.

Using MATLAB Constructor Functions

MATLAB provides several array constructor functions that you can use to build codistributed arrays of specific values, sizes, and classes. These functions operate in the same way as their nondistributed counterparts in the MATLAB language, except that they distribute the resultant array across the workers using the specified codistributor object, `codist`.

Constructor Functions

The codistributed constructor functions are listed here. Use the `codist` argument (created by the `codistributor` function: `codist=codistributor()`) to specify over which dimension to distribute the array. See the individual reference pages for these functions for further syntax and usage information.

```
eye(___,codist)
false(___,codist)
Inf(___,codist)
NaN(___,codist)
ones(___,codist)
```

```
rand(___,codist)
randi(___,codist)
randn(___,codist)
true(___,codist)
zeros(___,codist)

codistributed.cell(m,n,...,codist)
codistributed.colon(a,d,b)
codistributed.linspace(m,n,...,codist)
codistributed.logspace(m,n,...,codist)
sparse(m,n,codist)
codistributed.speye(m,...,codist)
codistributed.sprand(m,n,density,codist)
codistributed.sprandn(m,n,density,codist)
```

Local Arrays

That part of a codistributed array that resides on each worker is a piece of a larger array. Each worker can work on its own segment of the common array, or it can make a copy of that segment in a variant or private array of its own. This local copy of a codistributed array segment is called a *local array*.

Creating Local Arrays from a Codistributed Array

The `getLocalPart` function copies the segments of a codistributed array to a separate variant array. This example makes a local copy L of each segment of codistributed array D. The size of L shows that it contains only the local part of D for each worker. Suppose you distribute an array across four workers:

```
spmd(4)
    A = [1:80; 81:160; 161:240];
    D = codistributed(A);
    size(D)
      L = getLocalPart(D);
    size(L)
end
```

returns on each worker:

```
3    80
3    20
```

Each worker recognizes that the codistributed array D is 3-by-80. However, notice that the size of the local part, L, is 3-by-20 on each worker, because the 80 columns of D are distributed over four workers.

Creating a Codistributed from Local Arrays

Use the `codistributed.build` function to perform the reverse operation. This function, described in "Building from Smaller Arrays" on page 4-6, combines the local variant arrays into a single array distributed along the specified dimension.

Continuing the previous example, take the local variant arrays L and put them together as segments to build a new codistributed array X.

```
spmd
  codist = codistributor1d(2,[20 20 20 20],[3 80]);
```

```
    X = codistributed.build(L,codist);
    size(X)
end
```

returns on each worker:

```
3    80
```

Obtaining information About the Array

MATLAB offers several functions that provide information on any particular array. In addition to these standard functions, there are also two functions that are useful solely with codistributed arrays.

Determining Whether an Array Is Codistributed

The `iscodistributed` function returns a logical `1` (`true`) if the input array is codistributed, and logical `0` (`false`) otherwise. The syntax is

```
spmd, TF = iscodistributed(D), end
```

where `D` is any MATLAB array.

Determining the Dimension of Distribution

The codistributor object determines how an array is partitioned and its dimension of distribution. To access the codistributor of an array, use the `getCodistributor` function. This returns two properties, `Dimension` and `Partition`:

```
spmd, getCodistributor(X), end

      Dimension: 2
      Partition: [20 20 20 20]
```

The `Dimension` value of `2` means the array `X` is distributed by columns (dimension 2); and the `Partition` value of `[20 20 20 20]` means that twenty columns reside on each of the four workers.

To get these properties programmatically, return the output of `getCodistributor` to a variable, then use dot notation to access each property:

```
spmd
    C = getCodistributor(X);
    part = C.Partition
    dim  = C.Dimension
end
```

Other Array Functions

Other functions that provide information about standard arrays also work on codistributed arrays and use the same syntax.

- `length` — Returns the length of a specific dimension.
- `ndims` — Returns the number of dimensions.
- `numel` — Returns the number of elements in the array.
- `size` — Returns the size of each dimension.

- `is*` — Many functions that have names beginning with `'is'`, such as `ischar` and `issparse`.

Changing the Dimension of Distribution

When constructing an array, you distribute the parts of the array along one of the array's dimensions. You can change the direction of this distribution on an existing array using the `redistribute` function with a different codistributor object.

Construct an 8-by-16 codistributed array D of random values distributed by columns on four workers:

```
spmd
    D = rand(8,16,codistributor());
    size(getLocalPart(D))
end
```

returns on each worker:

```
8    4
```

Create a new codistributed array distributed by rows from an existing one already distributed by columns:

```
spmd
    X = redistribute(D, codistributor1d(1));
    size(getLocalPart(X))
end
```

returns on each worker:

```
2    16
```

Restoring the Full Array

You can restore a codistributed array to its undistributed form using the `gather` function. `gather` takes the segments of an array that reside on different workers and combines them into a replicated array on all workers, or into a single array on one worker.

Distribute a 4-by-10 array to four workers along the second dimension:

```
spmd,  A = [11:20; 21:30; 31:40; 41:50],  end
A =
    11    12    13    14    15    16    17    18    19    20
    21    22    23    24    25    26    27    28    29    30
    31    32    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48    49    50

spmd,  D = codistributed(A),  end


        WORKER 1          WORKER 2        WORKER 3      WORKER 4
    11    12    13  | 14    15    16  | 17    18  | 19    20
    21    22    23  | 24    25    26  | 27    28  | 29    30
    31    32    33  | 34    35    36  | 37    38  | 39    40
    41    42    43  | 44    45    46  | 47    48  | 49    50
                    |                 |           |
                    |                 |           |
```

```
spmd,  size(getLocalPart(D)),  end
Lab 1:
    4     3
Lab 2:
    4     3
Lab 3:
    4     2
Lab 4:
    4     2
```

Restore the undistributed segments to the full array form by gathering the segments:

```
spmd,  X = gather(D),  end
X =
    11    12    13    14    15    16    17    18    19    20
    21    22    23    24    25    26    27    28    29    30
    31    32    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48    49    50

spmd,  size(X),  end
    4    10
```

Indexing into a Codistributed Array

While indexing into a nondistributed array is fairly straightforward, codistributed arrays require additional considerations. Each dimension of a nondistributed array is indexed within a range of 1 to the final subscript, which is represented in MATLAB by the `end` keyword. The length of any dimension can be easily determined using either the `size` or `length` function.

With codistributed arrays, these values are not so easily obtained. For example, the second segment of an array (that which resides in the workspace of worker 2) has a starting index that depends on the array distribution. For a 200-by-1000 array with a default distribution by columns over four workers, the starting index on worker 2 is 251. For a 1000-by-200 array also distributed by columns, that same index would be 51. As for the ending index, this is not given by using the `end` keyword, as `end` in this case refers to the end of the entire array; that is, the last subscript of the final segment. The length of each segment is also not given by using the `length` or `size` functions, as they only return the length of the entire array.

The MATLAB `colon` operator and `end` keyword are two of the basic tools for indexing into nondistributed arrays. For codistributed arrays, MATLAB provides a version of the `colon` operator, called `codistributed.colon`. This actually is a function, not a symbolic operator like `colon`.

Note When using arrays to index into codistributed arrays, you can use only replicated or codistributed arrays for indexing. The toolbox does not check to ensure that the index is replicated, as that would require global communications. Therefore, the use of unsupported variants (such as `labindex`) to index into codistributed arrays might create unexpected results.

Example: Find a Particular Element in a Codistributed Array

Suppose you have a row vector of 1 million elements, distributed among several workers, and you want to locate its element number 225,000. That is, you want to know what worker contains this element, and in what position in the local part of the vector on that worker. The `globalIndices` function provides a correlation between the local and global indexing of the codistributed array.

```
D = rand(1,1e6,'distributed'); %Distributed by columns
spmd
    globalInd = globalIndices(D,2);
    pos = find(globalInd == 225e3);
    if ~isempty(pos)
      fprintf(...
      'Element is in position %d on worker %d.\n', pos, labindex);
    end
end
```

If you run this code on a pool of four workers you get this result:

```
Lab 1:
  Element is in position 225000 on worker 1.
```

If you run this code on a pool of five workers you get this result:

```
Lab 2:
  Element is in position 25000 on worker 2.
```

Notice if you use a pool of a different size, the element ends up in a different location on a different worker, but the same code can be used to locate the element.

2-Dimensional Distribution

As an alternative to distributing by a single dimension of rows or columns, you can distribute a matrix by blocks using `'2dbc'` or two-dimensional block-cyclic distribution. Instead of segments that comprise a number of complete rows or columns of the matrix, the segments of the codistributed array are 2-dimensional square blocks.

For example, consider a simple 8-by-8 matrix with ascending element values. You can create this array in an `spmd` statement or communicating job.

```
spmd
    A = reshape(1:64, 8, 8)
end
```

The result is the replicated array:

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Suppose you want to distribute this array among four workers, with a 4-by-4 block as the local part on each worker. In this case, the lab grid is a 2-by-2 arrangement of the workers, and the block size is

a square of four elements on a side (i.e., each block is a 4-by-4 square). With this information, you can define the codistributor object:

```
spmd
    DIST = codistributor2dbc([2 2], 4);
end
```

Now you can use this codistributor object to distribute the original matrix:

```
spmd
    AA = codistributed(A, DIST)
end
```

This distributes the array among the workers according to this scheme:

If the lab grid does not perfectly overlay the dimensions of the codistributed array, you can still use '2dbc' distribution, which is block cyclic. In this case, you can imagine the lab grid being repeatedly overlaid in both dimensions until all the original matrix elements are included.

Using the same original 8-by-8 matrix and 2-by-2 lab grid, consider a block size of 3 instead of 4, so that 3-by-3 square blocks are distributed among the workers. The code looks like this:

```
spmd
    DIST = codistributor2dbc([2 2], 3)
    AA = codistributed(A, DIST)
end
```

The first "row" of the lab grid is distributed to worker 1 and worker 2, but that contains only six of the eight columns of the original matrix. Therefore, the next two columns are distributed to worker 1. This process continues until all columns in the first rows are distributed. Then a similar process applies to the rows as you proceed down the matrix, as shown in the following distribution scheme:

The diagram above shows a scheme that requires four overlays of the lab grid to accommodate the entire original matrix. The following code shows the resulting distribution of data to each of the workers.

```
spmd
    getLocalPart(AA)
end
```

```
Lab 1:

  ans =

        1      9     17     49     57
        2     10     18     50     58
        3     11     19     51     59
        7     15     23     55     63
        8     16     24     56     64

Lab 2:

  ans =
```

```
         25      33      41
         26      34      42
         27      35      43
         31      39      47
         32      40      48

Lab 3:

  ans =

          4      12      20      52      60
          5      13      21      53      61
          6      14      22      54      62

Lab 4:

  ans =

         28      36      44
         29      37      45
         30      38      46
```

The following points are worth noting:

- `'2dbc'` distribution might not offer any performance enhancement unless the block size is at least a few dozen. The default block size is 64.
- The lab grid should be as close to a square as possible.
- Not all functions that are enhanced to work on `'1d'` codistributed arrays work on `'2dbc'` codistributed arrays.

Looping Over a Distributed Range (for-drange)

In this section...
"Parallelizing a for-Loop" on page 4-15
"Codistributed Arrays in a for-drange Loop" on page 4-16

Note Using a `for`-loop over a distributed range (`drange`) is intended for explicit indexing of the distributed dimension of codistributed arrays (such as inside an `spmd` statement or a communicating job). For most applications involving parallel `for`-loops you should first try using `parfor` loops. See "Parallel for-Loops (parfor)".

Parallelizing a for-Loop

In some occasions you already have a coarse-grained application to perform, i.e. an application for which the run time is significantly greater than the communication time needed to start and stop the program. If you do not want to bother with the overhead of defining jobs and tasks, you can take advantage of the ease-of-use that `spmd` provides. Where an existing program might take hours or days to process all its independent data sets, you can shorten that time by distributing these independent computations over your cluster.

For example, suppose you have the following serial code:

```
results = zeros(1, numDataSets);
for i = 1:numDataSets
    load(['\\central\myData\dataSet' int2str(i) '.mat'])
    results(i) = processDataSet(i);
 end
plot(1:numDataSets, results);
save \\central\myResults\today.mat results
```

The following changes make this code operate in parallel, either interactively in `spmd` or in a communicating job:

```
results = zeros(1, numDataSets, codistributor());
for i = drange(1:numDataSets)
    load(['\\central\myData\dataSet' int2str(i) '.mat'])
    results(i) = processDataSet(i);
end
res = gather(results, 1);
if labindex == 1
    plot(1:numDataSets, res);
    print -dtiff -r300 fig.tiff;
    save \\central\myResults\today.mat res
end
```

Note that the length of the `for` iteration and the length of the codistributed array `results` need to match in order to index into `results` within a `for drange` loop. This way, no communication is required between the workers. If `results` was simply a replicated array, as it would have been when running the original code in parallel, each worker would have assigned into its part of `results`, leaving the remaining parts of `results` 0. At the end, `results` would have been a variant, and without explicitly calling `labSend` and `labReceive` or `gcat`, there would be no way to get the total results back to one (or all) workers.

When using the `load` function, you need to be careful that the data files are accessible to all workers if necessary. The best practice is to use explicit paths to files on a shared file system.

Correspondingly, when using the `save` function, you should be careful to only have one worker save to a particular file (on a shared file system) at a time. Thus, wrapping the code in `if labindex == 1` is recommended.

Because `results` is distributed across the workers, this example uses `gather` to collect the data onto worker 1.

A worker cannot plot a visible figure, so the `print` function creates a viewable file of the plot.

Codistributed Arrays in a for-drange Loop

When a `for`-loop over a distributed range is executed in a communicating job, each worker performs its portion of the loop, so that the workers are all working simultaneously. Because of this, no communication is allowed between the workers while executing a for-drange loop. In particular, a worker has access only to its partition of a codistributed array. Any calculations in such a loop that require a worker to access portions of a codistributed array from another worker will generate an error.

To illustrate this characteristic, you can try the following example, in which one for loop works, but the other does not.

With `spmd`, create two codistributed arrays, one an identity matrix, the other set to zeros, distributed across four workers.

```
D = eye(8, 8, codistributor())
E = zeros(8, 8, codistributor())
```

By default, these arrays are distributed by columns; that is, each of the four workers contains two columns of each array. If you use these arrays in a `for-drange` loop, any calculations must be self-contained within each worker. In other words, you can only perform calculations that are limited within each worker to the two columns of the arrays that the workers contain.

For example, suppose you want to set each column of array E to some multiple of the corresponding column of array D:

```
for j = drange(1:size(D,2)); E(:,j) = j*D(:,j); end
```

This statement sets the j-th column of E to j times the j-th column of D. In effect, while D is an identity matrix with 1s down the main diagonal, E has the sequence 1, 2, 3, etc., down its main diagonal.

This works because each worker has access to the entire column of D and the entire column of E necessary to perform the calculation, as each worker works independently and simultaneously on two of the eight columns.

Suppose, however, that you attempt to set the values of the columns of E according to different columns of D:

```
for j = drange(1:size(D,2)); E(:,j) = j*D(:,j+1); end
```

This method fails, because when j is 2, you are trying to set the second column of E using the third column of D. These columns are stored in different workers, so an error occurs, indicating that communication between the workers is not allowed.

Restrictions

To use `for-drange` on a codistributed array, the following conditions must exist:

- The codistributed array uses a 1-dimensional distribution scheme (not `2dbc`).
- The distribution complies with the default partition scheme.
- The variable over which the `for-drange` loop is indexing provides the array subscript for the distribution dimension.
- All other subscripts can be chosen freely (and can be taken from `for`-loops over the full range of each dimension).

To loop over all elements in the array, you can use `for-drange` on the dimension of distribution, and regular `for`-loops on all other dimensions. The following example executes in an `spmd` statement running on a parallel pool of 4 workers:

```
spmd
  PP = zeros(6,8,12,'codistributed');
  RR = rand(6,8,12,codistributor())
  % Default distribution:
  %   by third dimension, evenly across 4 workers.

  for ii = 1:6
    for jj = 1:8
      for kk = drange(1:12)
        PP(ii,jj,kk) = RR(ii,jj,kk) + labindex;
      end
    end
  end
end
```

To view the contents of the array, type:

```
PP
```

Run MATLAB Functions with Distributed Arrays

Hundreds of functions in MATLAB and other toolboxes are enhanced so that they operate on distributed arrays.

```
D = distributed(gallery('lehmer',n));
e = eig(D);
```

If any of the input arguments to these distributed-enabled functions is a distributed array, their output arrays are distributed, unless returning MATLAB data is more appropriate (for example, numel).

Distributed arrays are well suited for large mathematical computations, such as large problems of linear algebra. You can also use distributed arrays for big data processing. For more information on distributing arrays, see "Distributing Arrays to Parallel Workers" on page 3-10.

Check Distributed Array Support in Functions

If a MATLAB function has distributed array support, you can consult additional distributed array usage information on its function page. See **Distributed Arrays** in the **Extended Capabilities** section at the end of the function page.

Tip For a filtered list of all MATLAB functions that support distributed arrays, see Function List (Distributed Arrays).

You can browse functions that support distributed arrays from all MathWorks products at the following link: All Functions List (Distributed Arrays). Alternatively, you can filter by product. On the **Help** bar, click **Functions**. In the function list, browse the left pane to select a product, for example, MATLAB. At the bottom of the left pane, select **Distributed Arrays**. If you select a product that does not have distributed-enabled functions, then the **Distributed Arrays** filter is not available.

For information about updates to individual distributed-enabled functions, see the release notes.

To check support for sparse distributed arrays, consult the following section.

Support for Sparse Distributed Arrays

The following list shows functions that can help you work with sparse distributed arrays. In addition to this list, most element-wise functions in MATLAB also work for distributed arrays.

bandwidth	cumsum	isdiag	pcg	spones	vertcat([;])
bicg	diag	istril	power(.^)	subsasgn	
bicgstab	diff	istriu	plus(+)	subsref	
bicgstabl	find	ldivide(.\)	qmr	svds	
cat	flip	lsqr	rdivide(./)	tfqmr	
cgs	fliplr	minus(-)	rot90	transpose(.')	
ctranspose(')	flipud	mldivide(\)	sort	tril	
cummax	gmres	mrdivide(/)	sortrows	triu	
cummin	horzcat([])	mtimes(*)	sparse	uminus(-)	
cumprod	isbanded	normest	spfun	uplus(+)	

Programming Overview

This chapter provides information you need for programming with Parallel Computing Toolbox software. Further details of evaluating functions in a cluster, programming independent jobs, and programming communicating jobs are covered in later chapters. This chapter describes features common to programming all kinds of jobs. The sections are as follows.

How Parallel Computing Products Run a Job

In this section...
"Overview" on page 5-2
"Toolbox and Server Components" on page 5-3
"Life Cycle of a Job" on page 5-6

Overview

Parallel Computing Toolbox and MATLAB Parallel Server software let you solve computationally and data-intensive problems using MATLAB and Simulink on multicore and multiprocessor computers. Parallel processing constructs such as parallel for-loops and code blocks, distributed arrays, parallel numerical algorithms, and message-passing functions let you implement task-parallel and data-parallel algorithms at a high level in MATLAB without programming for specific hardware and network architectures.

A *job* is some large operation that you need to perform in your MATLAB session. A job is broken down into segments called *tasks*. You decide how best to divide your job into tasks. You could divide your job into identical tasks, but tasks do not have to be identical.

The MATLAB session in which the job and its tasks are defined is called the *client* session. Often, this is on the machine where you program MATLAB. The client uses Parallel Computing Toolbox software to perform the definition of jobs and tasks and to run them on a cluster local to your machine. MATLAB Parallel Server software is the product that performs the execution of your job on a cluster of machines.

The MATLAB Job Scheduler is the process that coordinates the execution of jobs and the evaluation of their tasks. The MATLAB Job Scheduler distributes the tasks for evaluation to the server's individual MATLAB sessions called *workers*. Use of the MATLAB Job Scheduler to access a cluster is optional; the distribution of tasks to cluster workers can also be performed by a third-party scheduler, such as Microsoft® Windows® HPC Server (including CCS) or Platform LSF®.
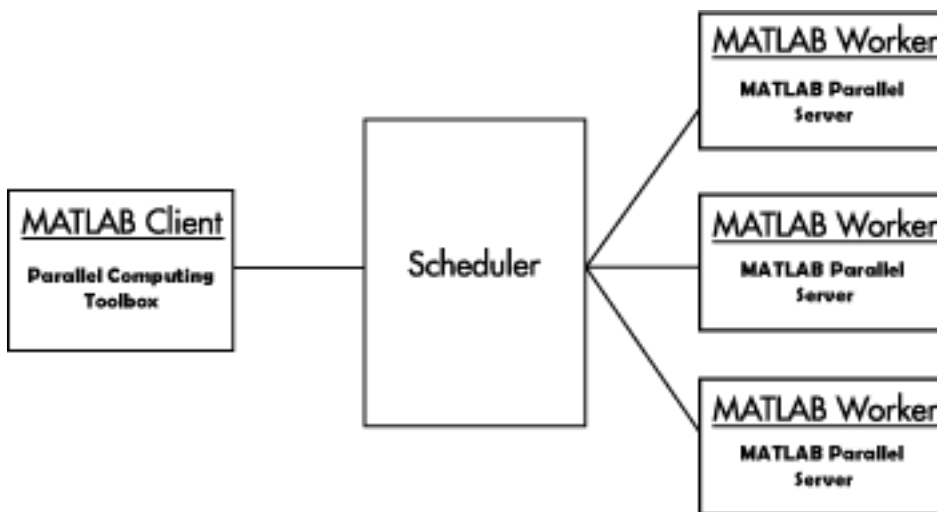
Basic Parallel Computing Setup

Toolbox and Server Components

- "MATLAB Job Scheduler, Workers, and Clients" on page 5-3
- "Local Cluster" on page 5-4
- "Third-Party Schedulers" on page 5-4
- "Components on Mixed Platforms or Heterogeneous Clusters" on page 5-5
- "mjs Service" on page 5-5
- "Components Represented in the Client" on page 5-5

MATLAB Job Scheduler, Workers, and Clients

The MATLAB Job Scheduler can be run on any machine on the network. The MATLAB Job Scheduler runs jobs in the order in which they are submitted, unless any jobs in its queue are promoted, demoted, canceled, or deleted.

Each worker is given a task from the running job by the MATLAB Job Scheduler, executes the task, returns the result to the MATLAB Job Scheduler, and then is given another task. When all tasks for a running job have been assigned to workers, the MATLAB Job Scheduler starts running the next job on the next available worker.

A MATLAB Parallel Server software setup usually includes many workers that can all execute tasks simultaneously, speeding up execution of large MATLAB jobs. It is generally not important which worker executes a specific task. In an independent job, the workers evaluate tasks one at a time as available, perhaps simultaneously, perhaps not, returning the results to the MATLAB Job Scheduler. In a communicating job, the workers evaluate tasks simultaneously. The MATLAB Job Scheduler then returns the results of all the tasks in the job to the client session.

Note For testing your application locally or other purposes, you can configure a single computer as client, worker, and MATLAB Job Scheduler host. You can also have more than one worker session or more than one MATLAB Job Scheduler session on a machine.
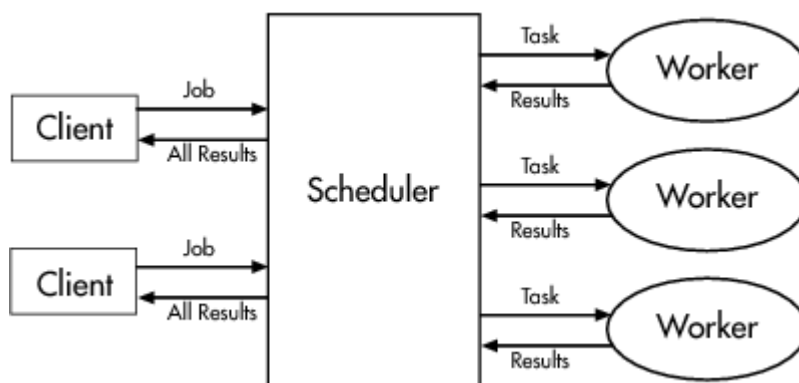
Interactions of Parallel Computing Sessions

A large network might include several MATLAB Job Schedulers as well as several client sessions. Any client session can create, run, and access jobs on any MATLAB Job Scheduler, but a worker session is registered with and dedicated to only one MATLAB Job Scheduler at a time. The following figure shows a configuration with multiple MATLAB Job Schedulers.
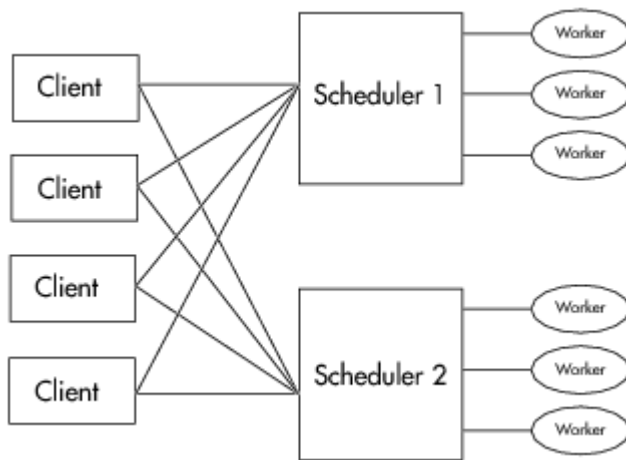
Cluster with Multiple Clients and MATLAB Job Schedulers

Local Cluster

A feature of Parallel Computing Toolbox software is the ability to run a local cluster of workers on the client machine, so that you can run jobs without requiring a remote cluster or MATLAB Parallel Server software. In this case, all the processing required for the client, scheduling, and task evaluation is performed on the same computer. This gives you the opportunity to develop, test, and debug your parallel applications before running them on your network cluster.

Third-Party Schedulers

As an alternative to using the MATLAB Job Scheduler, you can use a third-party scheduler. This could be a Microsoft Windows HPC Server (including CCS), Platform LSF scheduler, PBS Pro® scheduler, TORQUE scheduler, or a generic scheduler.

Choosing Between a Third-Party Scheduler and a MATLAB Job Scheduler

You should consider the following when deciding to use a third-party scheduler or the MATLAB Job Scheduler for distributing your tasks:

- Does your cluster already have a scheduler?

  If you already have a scheduler, you may be required to use it as a means of controlling access to the cluster. Your existing scheduler might be just as easy to use as a MATLAB Job Scheduler, so there might be no need for the extra administration involved.

- Is the handling of parallel computing jobs the only cluster scheduling management you need?

  The MATLAB Job Scheduler is designed specifically for MathWorks parallel computing applications. If other scheduling tasks are not needed, a third-party scheduler might not offer any advantages.

- Is there a file sharing configuration on your cluster already?

  The MATLAB Job Scheduler can handle all file and data sharing necessary for your parallel computing applications. This might be helpful in configurations where shared access is limited.

- Are you interested in batch mode or managed interactive processing?

  When you use a MATLAB Job Scheduler, worker processes usually remain running at all times, dedicated to their MATLAB Job Scheduler. With a third-party scheduler, workers are run as

applications that are started for the evaluation of tasks, and stopped when their tasks are complete. If tasks are small or take little time, starting a worker for each one might involve too much overhead time.

- Are there security concerns?

  Your own scheduler might be configured to accommodate your particular security requirements.

- How many nodes are on your cluster?

  If you have a large cluster, you probably already have a scheduler. Consult your MathWorks representative if you have questions about cluster size and the MATLAB Job Scheduler.

- Who administers your cluster?

  The person administering your cluster might have a preference for how jobs are scheduled.

- Do you need to monitor your job's progress or access intermediate data?

  A job run by the MATLAB Job Scheduler supports events and callbacks, so that particular functions can run as each job and task progresses from one state to another.

Components on Mixed Platforms or Heterogeneous Clusters

Parallel Computing Toolbox software and MATLAB Parallel Server software are supported on Windows, UNIX, and Macintosh operating systems. Mixed platforms are supported, so that the clients, MATLAB Job Scheduler, and workers do not have to be on the same platform. Other limitations are described at System Requirements.

In a mixed-platform environment, system administrators should be sure to follow the proper installation instructions for the local machine on which you are installing the software.

mjs Service

If you are using the MATLAB Job Scheduler, every machine that hosts a worker or MATLAB Job Scheduler session must also run the mjs service.

The mjs service controls the worker and MATLAB Job Scheduler sessions and recovers them when their host machines crash. If a worker or MATLAB Job Scheduler machine crashes, when the mjs service starts up again (usually configured to start at machine boot time), it automatically restarts the MATLAB Job Scheduler and worker sessions to resume their sessions from before the system crash. More information about the mjs service is available in the MATLAB Parallel Server documentation.

Components Represented in the Client

A client session communicates with the MATLAB Job Scheduler by calling methods and configuring properties of a *MATLAB Job Scheduler cluster object*. Though not often necessary, the client session can also access information about a worker session through a *worker object*.

When you create a job in the client session, the job actually exists in the MATLAB Job Scheduler job storage location. The client session has access to the job through a *job object*. Likewise, tasks that you define for a job in the client session exist in the MATLAB Job Scheduler data location, and you access them through *task objects*.

Life Cycle of a Job

When you create and run a job, it progresses through a number of stages. Each stage of a job is reflected in the value of the job object's `State` property, which can be `pending`, `queued`, `running`, or `finished`. Each of these stages is briefly described in this section.

The figure below illustrates the stages in the life cycle of a job. In the MATLAB Job Scheduler (or other scheduler), the jobs are shown categorized by their state. Some of the functions you use for managing a job are `createJob`, `submit`, and `fetchOutputs`.
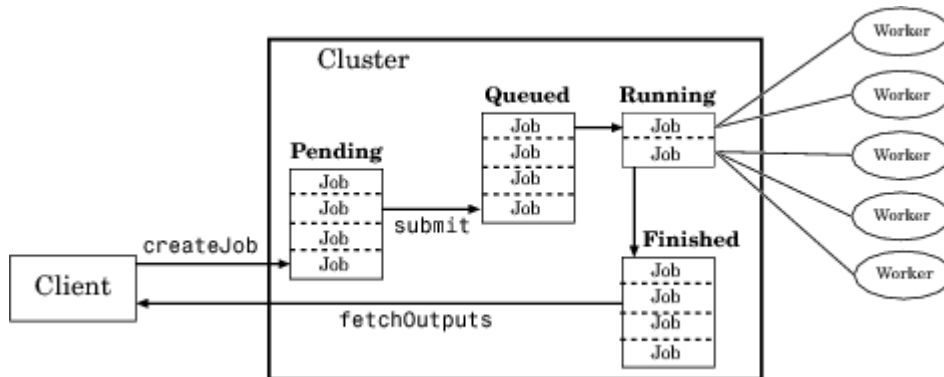
Stages of a Job

The following table describes each stage in the life cycle of a job.

Job Stage	Description
Pending	You create a job on the scheduler with the `createJob` function in your client session of Parallel Computing Toolbox software. The job's first state is `pending`. This is when you define the job by adding tasks to it.
Queued	When you execute the `submit` function on a job, the MATLAB Job Scheduler or scheduler places the job in the queue, and the job's state is `queued`. The scheduler executes jobs in the queue in the sequence in which they are submitted, all jobs moving up the queue as the jobs before them are finished. You can change the sequence of the jobs in the queue with the `promote` and `demote` functions.
Running	When a job reaches the top of the queue, the scheduler distributes the job's tasks to worker sessions for evaluation. The job's state is now `running`. If more workers are available than are required for a job's tasks, the scheduler begins executing the next job. In this way, there can be more than one job running at a time.
Finished	When all of a job's tasks have been evaluated, the job is moved to the `finished` state. At this time, you can retrieve the results from all the tasks in the job with the function `fetchOutputs`.
Failed	When using a third-party scheduler, a job might fail if the scheduler encounters an error when attempting to execute its commands or access necessary files.

Job Stage	Description
Deleted	When a job's data has been removed from its data location or from the MATLAB Job Scheduler with the `delete` function, the state of the job in the client is `deleted`. This state is available only as long as the job object remains in the client.

Note that when a job is finished, its data remains in the MATLAB Job Scheduler's `JobStorageLocation` folder, even if you clear all the objects from the client session. The MATLAB Job Scheduler or scheduler keeps all the jobs it has executed, until you restart the MATLAB Job Scheduler in a clean state. Therefore, you can retrieve information from a job later or in another client session, so long as the MATLAB Job Scheduler has not been restarted with the `-clean` option.

You can permanently remove completed jobs from the MATLAB Job Scheduler or scheduler's storage location using the Job Monitor GUI or the `delete` function.

Program a Job on a Local Cluster

In some situations, you might need to define the individual tasks of a job, perhaps because they might evaluate different functions or have uniquely structured arguments. To program a job like this, the typical Parallel Computing Toolbox client session includes the steps shown in the following example.

This example illustrates the basic steps in creating and running a job that contains a few simple tasks. Each task evaluates the `sum` function for an input array.

1 Identify a cluster. Use `parallel.defaultClusterProfile` to indicate that you are using the local cluster; and use `parcluster` to create the object `c` to represent this cluster. (For more information, see "Create a Cluster Object" on page 6-3.)

```
parallel.defaultClusterProfile('local');
c = parcluster();
```

2 Create a job. Create job `j` on the cluster. (For more information, see "Create a Job" on page 6-3.)

```
j = createJob(c)
```

3 Create three tasks within the job `j`. Each task evaluates the `sum` of the array that is passed as an input argument. (For more information, see "Create Tasks" on page 6-4.)

```
createTask(j, @sum, 1, {[1 1]});
createTask(j, @sum, 1, {[2 2]});
createTask(j, @sum, 1, {[3 3]});
```

4 Submit the job to the queue for evaluation. The scheduler then distributes the job's tasks to MATLAB workers that are available for evaluating. The local cluster might now start MATLAB worker sessions. (For more information, see "Submit a Job to the Cluster" on page 6-5.)

```
submit(j);
```

5 Wait for the job to complete, then get the results from all the tasks of the job. (For more information, see "Fetch the Job Results" on page 6-5.)

```
wait(j)
results = fetchOutputs(j)
results =
    [2]
    [4]
    [6]
```

6 Delete the job. When you have the results, you can permanently remove the job from the scheduler's storage location.

```
delete(j)
```

Specify Your Parallel Preferences

You can access your parallel preferences in any of the following ways:

- On the **Home** tab in the **Environment** section, select **Parallel > Parallel Preferences**
- Select the desktop pool indicator icon, and select **Parallel Preferences**.
- In the command window, type `preferences`.

  ```
  preferences
  ```

  In the navigation tree of the Preferences dialog box, select `Parallel Computing Toolbox`.
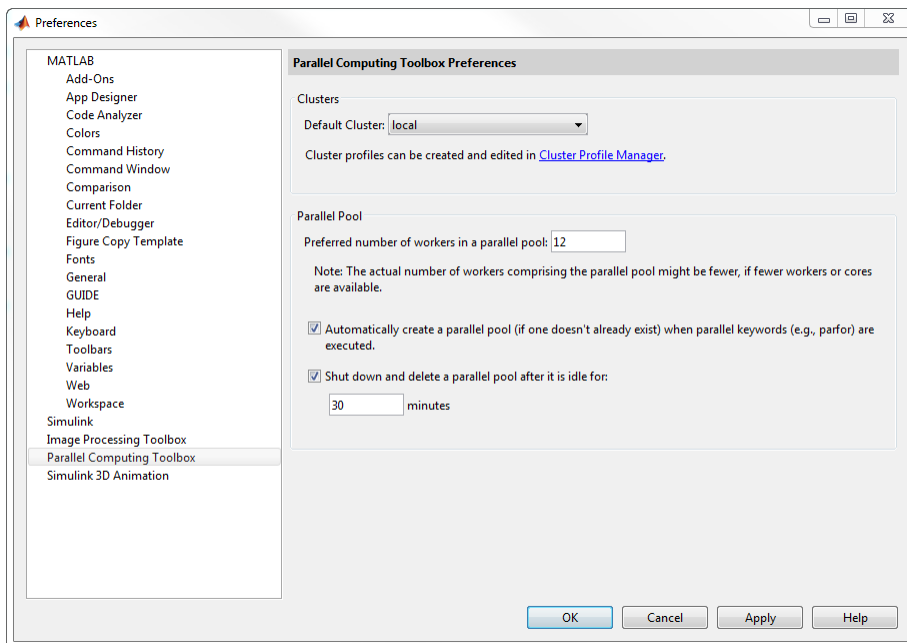
You can control your parallel preference settings as follows:

- **Default Cluster** — Choose the cluster you want to use. The default cluster is **local**. For more information, see "Add and Modify Cluster Profiles" on page 5-14.
- **Preferred number of workers** — Specify the number of workers in your parallel pool. The actual pool size is limited by licensing, cluster size, and cluster profile settings on page 5-14. See "Pool Size and Cluster Selection" on page 2-58. For the local profile, do not choose a preferred number of workers larger than 512. See also "Add and Modify Cluster Profiles" on page 5-14. Check your access to cloud computing from the **Parallel > Discover Clusters** menu.
- **Automatically create a parallel pool** — Select this option to start a pool automatically (if a pool does not yet exist). Many functions can automatically start a parallel pool, including:
  - `parfor`
  - `spmd`
  - `distributed`
  - `Composite`
  - `parfeval`

- `parfevalOnAll`
- `gcp`
- `mapreduce`
- `mapreducer`

If you have selected **Automatically create a parallel pool**, you do not need to open a pool manually using the `parpool` function. If a pool automatically opens, you can still access the pool object with `gcp`.

- **Shut down and delete a parallel pool** — To shut down a parallel pool automatically if the pool has been idle for the specified amount of time, use the `IdleTimeout` setting. If you use the pool (for example, using `parfor` or `parfeval`), the timeout counter is reset. When the timeout is about to expire, a tooltip on the desktop pool indicator warns you and allows you to reset the timer. Note that modifying this setting changes the `IdleTimeout` of any already started pool.

See Also

Related Examples
- "Run MATLAB Functions with Automatic Parallel Support" on page 1-19
- "Scale Up from Desktop to Cluster" on page 9-88

More About
- "Decide When to Use parfor" on page 2-2
- "Scale Up parfor-Loops to Cluster and Cloud" on page 2-20
- "Add and Modify Cluster Profiles" on page 5-14

Discover Clusters and Use Cluster Profiles

In this section...
"Create and Manage Cluster Profiles" on page 5-11
"Discover Clusters" on page 5-12
"Create Cloud Cluster" on page 5-14
"Add and Modify Cluster Profiles" on page 5-14
"Import and Export Cluster Profiles" on page 5-18
"Edit Number of Workers and Cluster Settings" on page 5-19
"Use Your Cluster from MATLAB" on page 5-19

Parallel Computing Toolbox comes pre-configured with the cluster profile `local` for running parallel code on your local desktop machine.

Control parallel behavior using the **Parallel** menu on the MATLAB **Home** tab.
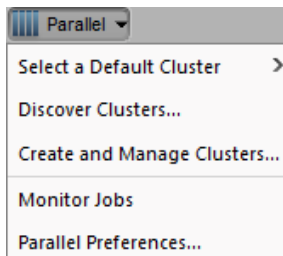
You can use the **Parallel** menu to:

- Discover other clusters running on your network or on Amazon EC2. Click **Parallel > Discover Clusters**. For more information, see "Discover Clusters" on page 5-12.

- Create and manage cluster profiles using the Cluster Profile Manager. Click **Parallel > Create and Manage Clusters**. For more information, see "Create and Manage Cluster Profiles" on page 5-11.

Create and Manage Cluster Profiles

Cluster profiles let you define certain properties for your cluster, then have these properties applied when you create cluster, job, and task objects in the MATLAB client. Some of the functions that support the use of cluster profiles are

- `batch`
- `parpool`
- `parcluster`

Manage cluster profiles using the Cluster Profile Manager. To open the Cluster Profile Manager, on the **Home** tab, in the **Environment** section, select **Parallel > Create and Manage Clusters**.
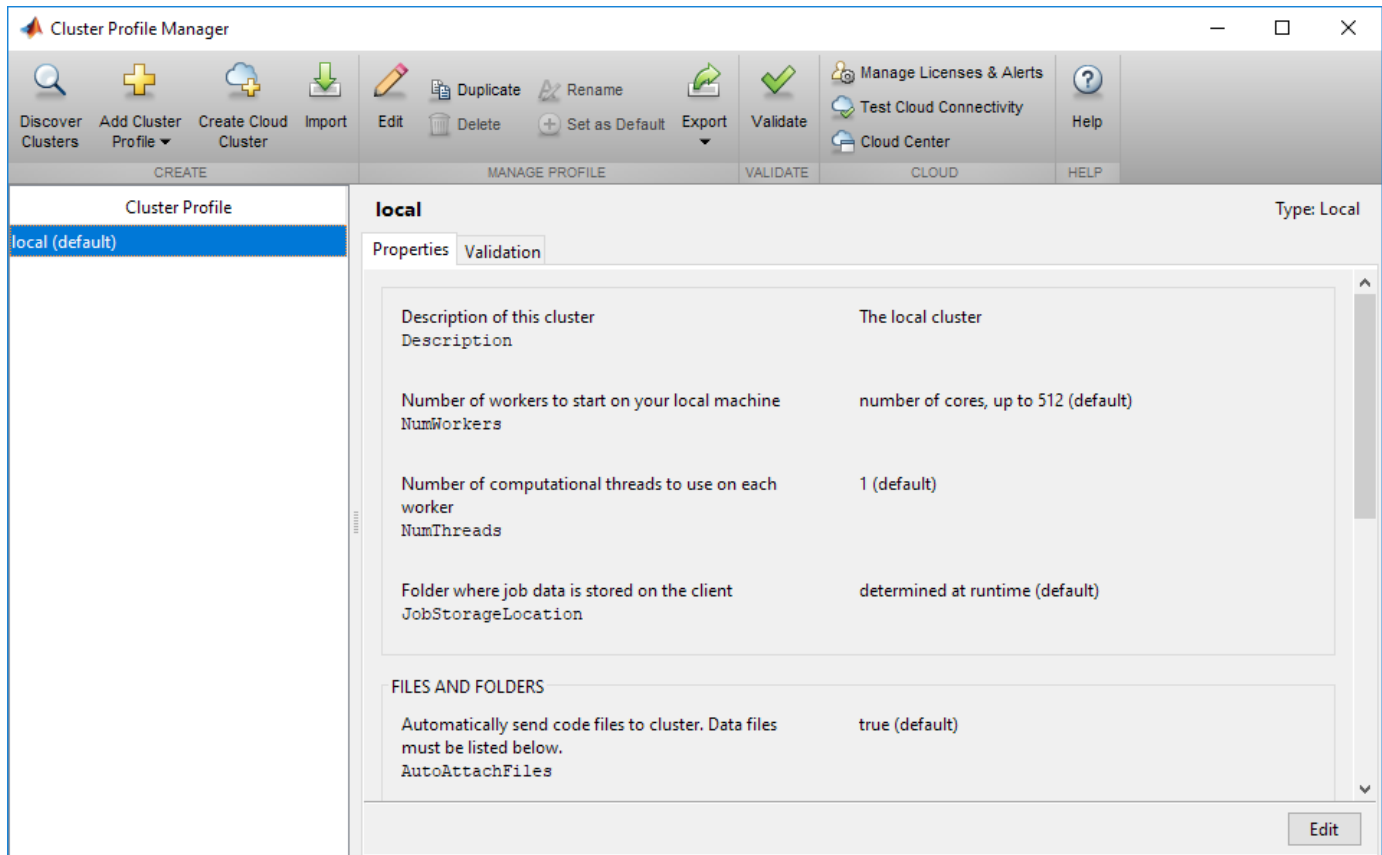
You can use the Cluster Profile Manager to:

- Discover other clusters running on your network or on Amazon® AWS®. For more information, see "Discover Clusters" on page 5-12.

- Create a cluster in the cloud, such as Amazon AWS. For more information, see "Create Cloud Cluster" on page 5-14.

- Add cluster profiles and modify their properties. For more information, see "Add and Modify Cluster Profiles" on page 5-14.

- Import and export cluster profiles. For more information, see "Import and Export Cluster Profiles" on page 5-18.

- Specify profile properties. For more information, see "Edit Number of Workers and Cluster Settings" on page 5-19.

- Validate that a cluster profile is ready for use in MATLAB.

Discover Clusters

You can let MATLAB discover clusters for you. Use either of the following techniques to discover those clusters which are available for you to use:

- On the **Home** tab in the **Environment** section, select **Parallel > Discover Clusters**

- In the Cluster Profile Manager, select **Discover Clusters**

This opens the Discover Clusters dialog box, where you can search for MATLAB Parallel Server clusters:
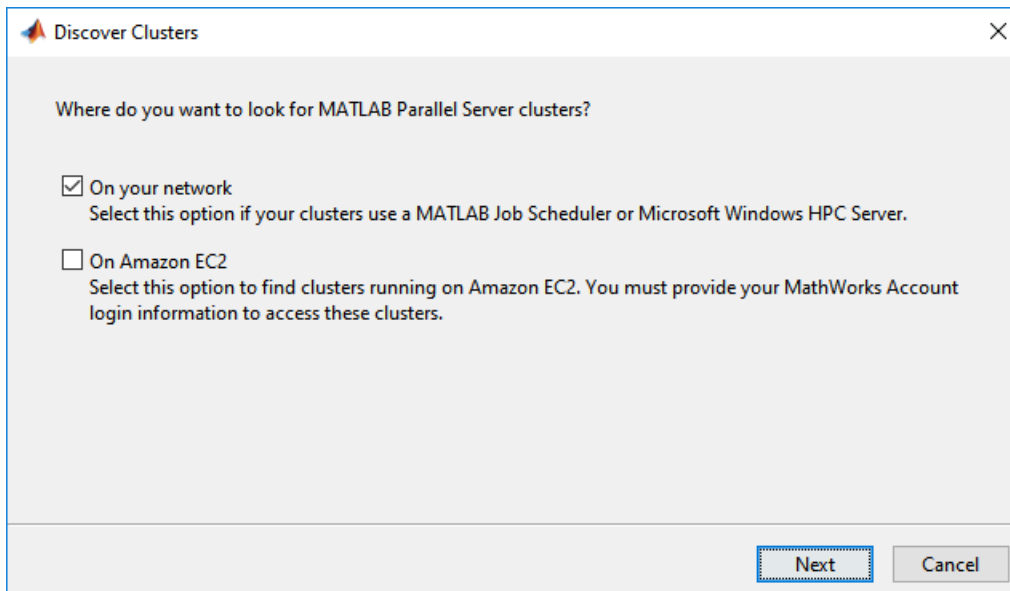
If you select **On your network**, you see a new window. Select this option if your clusters use a MATLAB Job Scheduler or Microsoft Windows HPC server. As clusters are discovered, they populate a list for your selection. If you already have a profile for any of the listed clusters, those profile names are included in the list. If you want to create a new profile for one of the discovered clusters, select the name of the cluster you want to use, and select **Next**. The subsequent dialog box lets you choose if you want to set the created profile as your default. This option is not supported in MATLAB Online.

If you select **On Amazon EC2**, you search for clusters running on Amazon EC2. To access these clusters, you must provide your MathWorks Account login information.

Requirements for Cluster Discovery

Cluster discovery is supported only for MATLAB Job Schedulers, Microsoft Windows HPC Server, and Amazon EC2 cloud clusters. If you need to integrate your scheduler with MATLAB Parallel Server, or create a cluster profile for a different supported scheduler, see "Get Started with MATLAB Parallel Server" (MATLAB Parallel Server). The following requirements apply to cluster discovery:

- MATLAB Job Scheduler — MATLAB Job Scheduler clusters support two different means of discovery:

  - Multicast: The discover clusters functionality uses the multicast networking protocol from the client to search for head nodes where a MATLAB Job Scheduler is running. This requires that the multicast networking protocol is enabled and working on the network that connects the MATLAB Job Scheduler head nodes (where the schedulers are running) and the client machines. This form of discovery might be limited to the client local subnet, and therefore not always able to discover a MATLAB Job Scheduler elsewhere in your network.

  - DNS SRV: An alternative discovery technique is to search for clusters by DNS service records.

    The Domain Name System (DNS) is a standard for identifying host names with IP addresses, either on the Internet or in a private network. Using DNS allows discovery of MATLAB Job Scheduler clusters by identifying specific hosts rather than broadcasting across your network.

A DNS service (SRV) record defines the location of hosts and ports of services, such as those related to the clusters you want to discover. Your system administrator creates DNS SRV records in your organization's DNS infrastructure. For a description of the required record, and validation information, see "DNS SRV Record" (MATLAB Parallel Server).

- HPC Server — The discover clusters functionality uses Active Directory Domain Services to discover head nodes. HPC Server head nodes are added to the Active Directory during installation of the HPC Server software.

- Amazon EC2 — The discover clusters functionality requires a working network connection between the client and the Cloud Center web services running in mathworks.com.

Create Cloud Cluster

You can create clusters in Amazon AWS cloud services directly from the Cluster Profile Manager. In the Cluster Profile Manager, select **Create Cloud Cluster**. Sign up with your MathWorks Account and complete the required steps. Then, you can create a cloud cluster and configure parameters, such as the number of machines or the number of workers per machine. For more information on each of the available parameters, see Create a Cloud Cluster. When you complete all the steps, MATLAB creates a new cluster profile for you. You can modify its properties from the Cluster Profile Manager.

To manage your licenses, test cloud connectivity, or manage your cloud clusters in MathWorks Cloud Center, go to Cluster Profile Manager toolstrip > **CLOUD** section.

Add and Modify Cluster Profiles

With the Cluster Profile Manager, you can add a cluster profile for a MATLAB job scheduler or a third-party scheduler. If you need to set up your cluster for use with MATLAB, see "Get Started with MATLAB Parallel Server" (MATLAB Parallel Server).

The following example provides instructions on how to add and modify profiles using the Cluster Profile Manager.

Suppose you want to create a profile to set several properties for jobs to run in a MATLAB Job Scheduler cluster. The following example illustrates a possible workflow, where you create two profiles differentiated only by the number of workers they use.

1 In the Cluster Profile Manager, select **Add Cluster Profile** > **MATLAB Job Scheduler**. This specifies that you want a new profile for a MATLAB Job Scheduler cluster.
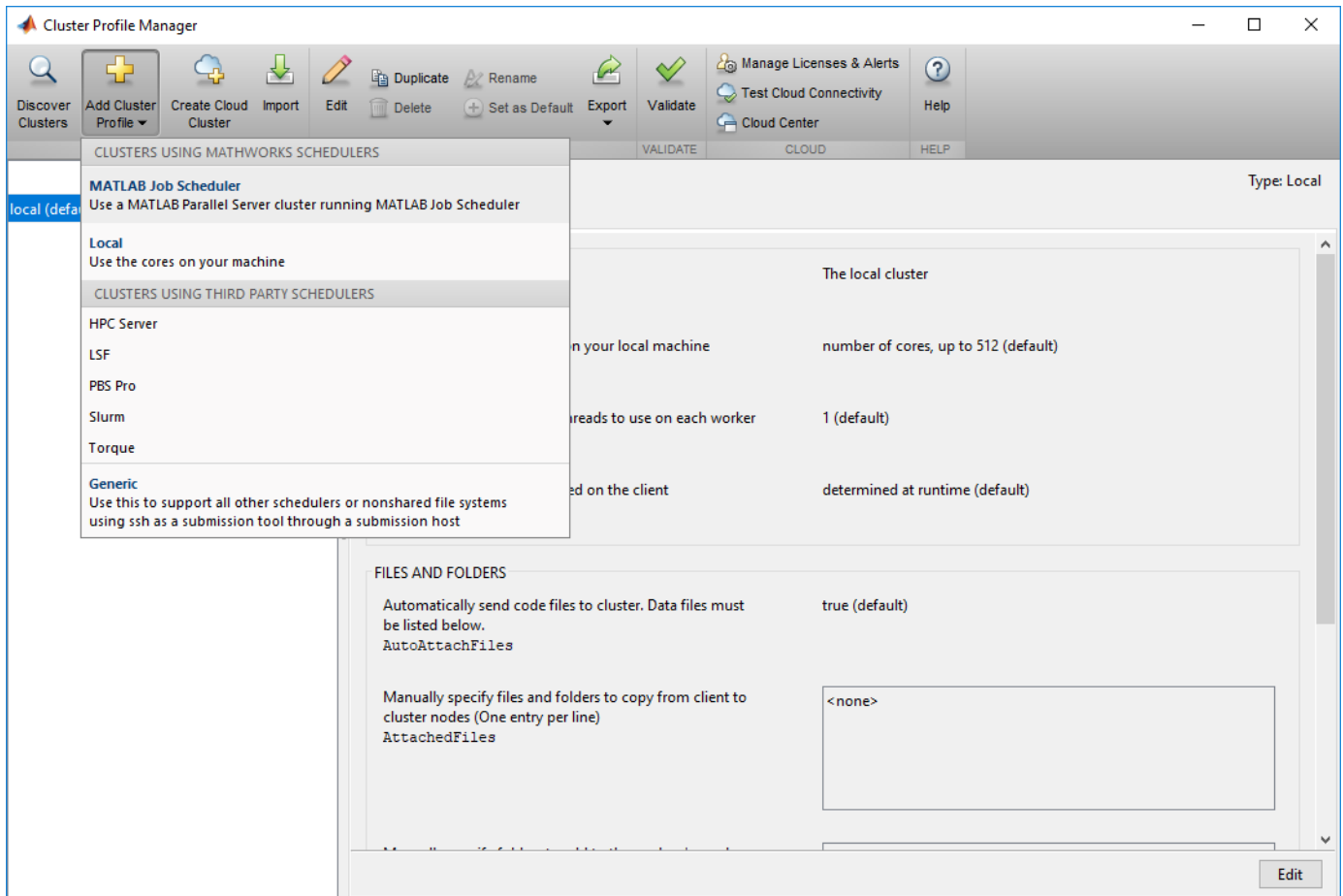
This creates and displays a new profile, called MJSProfile1.

2 Double-click the new profile name in the listing, and modify the profile name to be `MyMJSProfile1`.

3 Select **Edit** in the tool strip so that you can set your profile property values.

In the Description field, enter the text `MJS with 4 workers`, as shown in the following figure. Enter the host name for the machine on which the MATLAB Job Scheduler is running, and the name of the MATLAB Job Scheduler. If you are entering information for an actual MATLAB Job Scheduler already running on your network, enter the actual names. If you are unsure about the MATLAB Job Scheduler names and locations on your network, ask your system administrator for help.
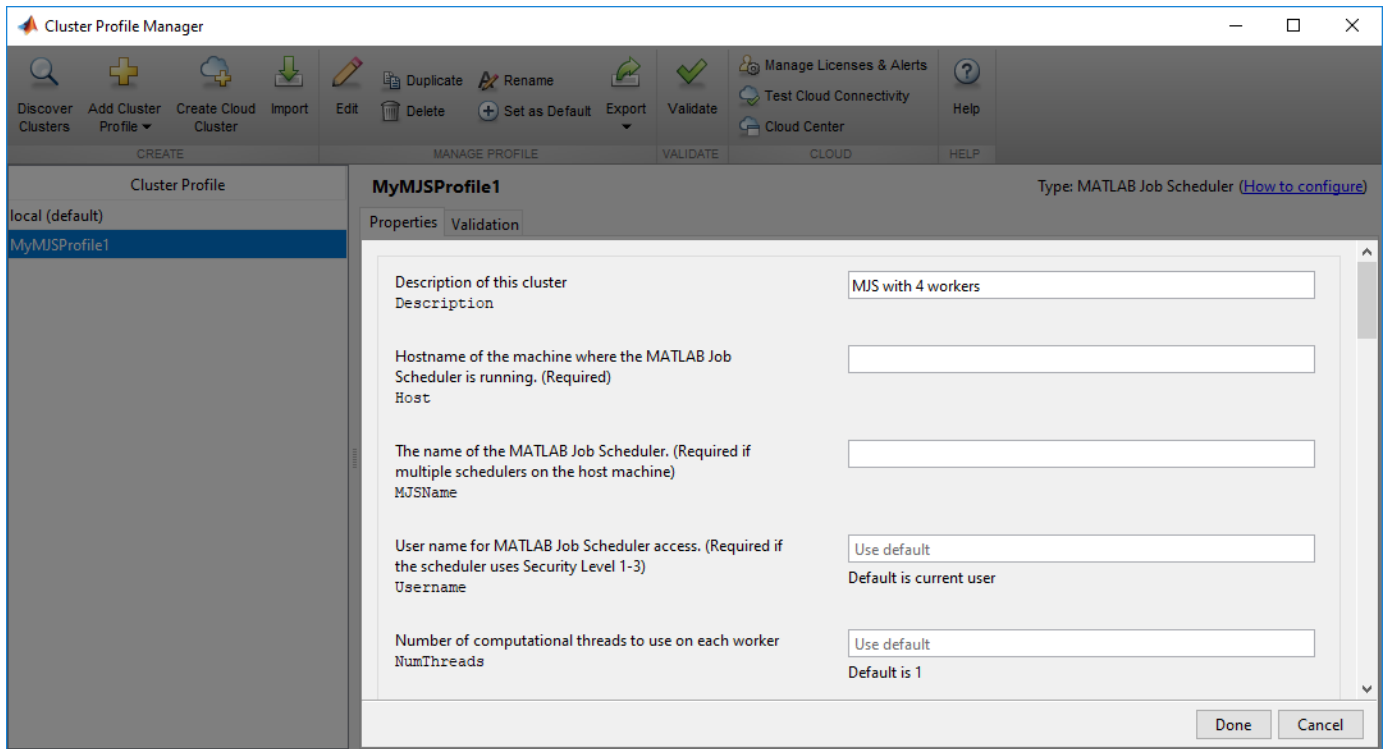
Note If the MATLAB Job Scheduler is using a nondefault BASE_PORT setting as defined in the mjs_def file, the Host property in the cluster profile must be appended with this BASE_PORT number. For example, MJS-Host:40000.

4 Scroll down to the Workers section, and for the Range of number of workers, enter the two-element vector [4 4]. This specifies that jobs using this profile require at least four workers and no more than four workers. Therefore, a job using this profile runs on exactly four workers, even if it has to wait until four workers are available before starting.
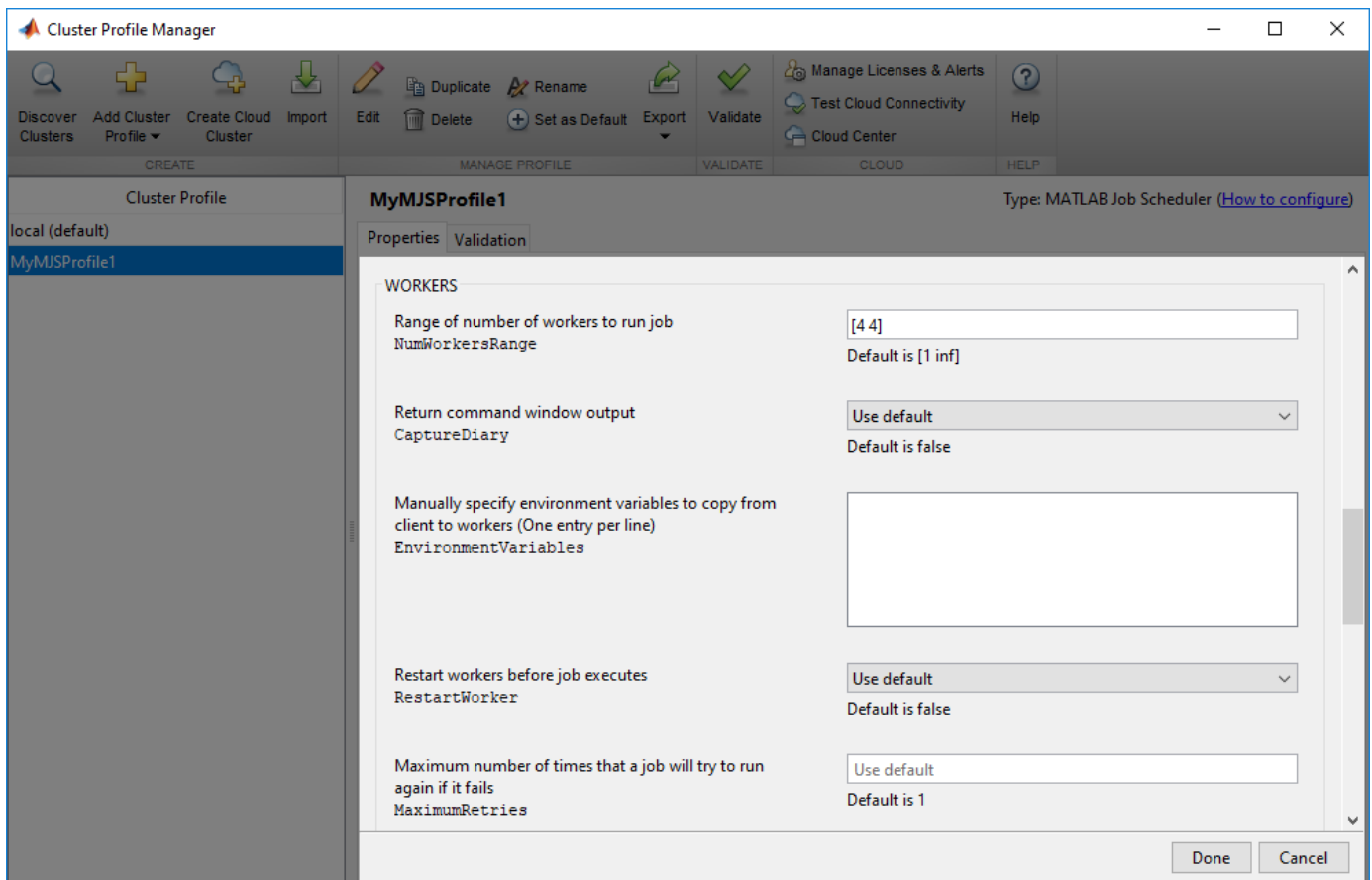
You might want to edit other properties depending on your particular network and cluster situation.

5   Select **Done** to save the profile settings.

To create a similar profile with just a few differences, you can duplicate an existing profile and modify only the parts you need to change, as follows:

1   In the Cluster Profile Manager, right-click the profile name `MyMJSProfile1` in the list and select **Duplicate**.

This creates a duplicate profile with a name based on the original profile name appended with `_Copy`.

2   Double-click the new profile name and edit its name to be `MyMJSprofile2`.

3   Select **Edit** to allow you to change the profile property values.

4   Edit the description field to change its text to `MJS with any workers`.

5   Scroll down to the Workers section, and for the Range of number of workers, clear the `[4 4]` and leave the field blank.

6   Select **Done** to save the profile settings and to close the properties editor.

You now have two profiles that differ only in the number of workers required for running a job.

When creating a job, you can apply either profile to that job as a way of specifying how many workers it should run on.

You can see examples of profiles for different kinds of supported schedulers in the MATLAB Parallel Server installation instructions at "Configure Your Cluster" (MATLAB Parallel Server).

Import and Export Cluster Profiles

Cluster profiles are stored as part of your MATLAB preferences, so they are generally available on an individual user basis. To make a cluster profile available to someone else, you can export it to a separate `.mlsettings` file. In this way, a repository of profiles can be created so that all users of a computing cluster can share common profiles.

To export a cluster profile:

1 In the Cluster Profile Manager, select (highlight) the profile you want to export.

2 Select **Export > Export**. (Alternatively, you can right-click the profile in the listing and select **Export**.)

If you want to export all your profiles to a single file, select **Export > Export All**

3 In the Export profiles to file dialog box, specify a location and name for the file. The default file name is the same as the name of the profile it contains, with a `.mlsettings` extension appended; you can alter the names if you want to.

Note that you cannot export profiles for Cloud Center personal clusters.

Profiles saved in this way can then be imported by other MATLAB users:

1 In the Cluster Profile Manager, select **Import**.

2 In the Import profiles from file dialog box, browse to find the `.mlsettings` file for the profile you want to import. Select the file and select **Open**.

The imported profile appears in your Cluster Profile Manager list. Note that the list contains the profile name, which is not necessarily the file name. If you already have a profile with the same name as the one you are importing, the imported profile gets an extension added to its name so you can distinguish it.

You can also export and import profiles programmatically with the `parallel.exportProfile` and `parallel.importProfile` functions.

Export Profiles for MATLAB Compiler

You can use an exported profile with MATLAB Compiler and MATLAB Compiler SDK to identify cluster setup information for running compiled applications on a cluster. For example, the `setmcruserdata` function can use the exported profile file name to set the value for the key `ParallelProfile`. For more information and examples of deploying parallel applications, see "Pass Parallel Computing Toolbox Profile at Run Time" (MATLAB Compiler), and "Use Parallel Computing Toolbox in Deployed Applications" (MATLAB Compiler SDK).

A compiled application has the same default profile and the same list of alternative profiles that the compiling user had when the application was compiled. This means that in many cases the profile file is not needed, as might be the case when using the `local` profile for local workers. If an exported file is used, the first profile in the file becomes the default when imported. If any of the imported profiles have the same name as any of the existing profiles, they are renamed during import (though their names in the file remain unchanged).

Edit Number of Workers and Cluster Settings

After you create a cluster profile, you can specify the number of workers and other profile properties:

- `NumWorkers`: the number of workers to start a pool. The actual pool size might be limited by licensing, cluster size, and cluster profile settings. See "Pool Size and Cluster Selection" on page 2-58

- `NumThreads`: the number of computational threads to use on each worker. You can change `NumThreads`, so that your workers can run in multithreaded mode and use all the cores on your cluster. This allows you to increase the number of computational threads `NumThreads` on each worker, without increasing the number of workers `NumWorkers`. If you have more cores available, increase `NumThreads` to take full advantage of the built-in parallelism provided by the multithreaded nature of many of the underlying MATLAB libraries. For details, see Run MATLAB on multicore and multiprocessor machines .

Note Do not increase the number of threads across all workers on a machine to exceed the number of physical cores. In other words, make sure that `NumWorkers` x `NumThreads` ≤ number of physical cores on your machine. Otherwise you might have reduced performance.

Use Your Cluster from MATLAB

To run parallel language functions, such as `parpool` or `batch`, on a cluster, set the cluster profile as default, or use cluster objects.

Specify Default Cluster

To set a cluster profile as the default, use one of the following ways:

- On the **Home** tab in the **Environment** section, select **Parallel > Select a Default Cluster**, and from there, all your profiles are available. The default profile is indicated. You can select any profile in the list as the default.

- The Cluster Profile Manager indicates which is the default profile. You can select any profile in the list, then select **Set as Default**.

- You can get or set the default profile programmatically by using the `parallel.defaultClusterProfile` function. The following sets of commands achieve the same thing:

```
parallel.defaultClusterProfile('MyMJSProfile1')
parpool
```

or

```
parpool('MyMJSProfile1')
```

Specify Cluster Programmatically (parcluster)

The `parcluster` function creates a cluster object in your workspace according to the specified profile. The profile identifies a particular cluster and applies property values. For example,

```
c = parcluster('MyMJSProfile1')
```

This command finds the cluster defined by the settings of the profile named `MyMJSProfile1` and sets property values on the cluster object based on settings in the profile. Use a cluster object in

functions such as `parpool` or `batch`. By applying different profiles, you can alter your cluster choices without changing your MATLAB application code.

See Also

`batch` | `createJob` | `parallel.defaultClusterProfile` | `parallel.exportProfile` | `parallel.importProfile` | `parcluster` | `parpool` | `setmcruserdata`

Related Examples

- "Run Code on Parallel Pools" on page 2-55
- "Scale Up from Desktop to Cluster" on page 9-88
- "Pass Parallel Computing Toolbox Profile at Run Time" (MATLAB Compiler)
- "Use Parallel Computing Toolbox in Deployed Applications" (MATLAB Compiler SDK)
- "Verify Network Communications for Cluster Discovery" (MATLAB Parallel Server)

More About

- "Get Started with MATLAB Parallel Server" (MATLAB Parallel Server)
- "Clusters and Clouds"

External Websites

- https://www.mathworks.com/help/cloudcenter/
- https://www.mathworks.com/licensecenter

Apply Callbacks to MATLAB Job Scheduler Jobs and Tasks

The MATLAB Job Scheduler has the ability to trigger callbacks in the client session whenever jobs or tasks in the MATLAB Job Scheduler cluster change to specific states.

Client objects representing jobs and tasks in a MATLAB Job Scheduler cluster include the following properties:

Callback Property	Object	Cluster Profile Manager Field	Description
QueuedFcn	Job only	JobQueuedFcn	Specifies the function to execute in the client when a job is submitted to the MATLAB Job Scheduler queue
RunningFcn	Job or task	JobRunningFcn TaskRunningFcn	Specifies the function to execute in the client when a job or task begins its execution
FinishedFcn	Job or task	JobFinishedFcn TaskFinishedFcn	Specifies the function to execute in the client when a job or task completes its execution

You can set each of these properties to any valid MATLAB callback value in the Cluster Profile Manager, see the table and "Add and Modify Cluster Profiles" on page 5-14. The callback follows the same behavior for Handle Graphics®, passing into the callback function the object (job or task) that makes the call and an empty argument of event data.

These properties apply only in the client MATLAB session in which they are set. Later sessions that access the same job or task objects do not inherit the settings from previous sessions. You can apply the properties to existing jobs and tasks at the command-line, but the cluster profile settings apply only at the time these objects are first created.

Note The callback properties are available only when using a MATLAB Job Scheduler cluster.

Example 5.1. Create Callbacks at the Command Line

This example shows how to create job and task callbacks at the client session command line.

Create and save a callback function `clientTaskCompleted.m` on the path of the MATLAB client, with the following content:

```
function clientTaskCompleted(task,eventdata)
    disp(['Finished task: ' num2str(task.ID)])
```

Create a job and set its `QueuedFcn`, `RunningFcn`, and `FinishedFcn` properties, using a function handle to an anonymous function that sends information to the display.

```
c = parcluster('MyMJS');
j = createJob(c,'Name','Job_52a');
j.QueuedFcn = @(job,eventdata) disp([job.Name ' now ' job.State]);
j.RunningFcn = @(job,eventdata) disp([job.Name ' now ' job.State]);
j.FinishedFcn = @(job,eventdata) disp([job.Name ' now ' job.State]);
```

Create a task whose `FinishedFcn` is a function handle to the separate function.

```
createTask(j,@rand,1,{2,4}, ...
    'FinishedFcn',@clientTaskCompleted);
```

Run the job and note the output messages from both the job and task callbacks.

```
submit(j)
```

```
Job_52a now queued
Job_52a now running
Finished task: 1
Job_52a now finished
```

To use the same callbacks for any jobs and tasks on a given cluster, you should set these properties in the cluster profile. For details on editing profiles in the profile manager, see "Discover Clusters and Use Cluster Profiles" on page 5-11. These property settings apply to any jobs and tasks created using a cluster derived from this profile. The sequence is important, and must occur in this order:

1   Set the callback property values for the profile in the profile manager.
2   Use the cluster profile to create a cluster object in MATLAB.
3   Use the cluster object to create jobs and then tasks.

Example 5.2. Set Callbacks in a Cluster Profile

This example shows how to set several job and task callback properties using the profile manager.

Edit your MATLAB Job Scheduler cluster profile in the profile manager so that you can set the callback properties to the same values in the previous example. The saves profile looks like this:

CALLBACKS

Function that runs on the client when job reaches the finished state JobFinishedFcn	@(job,eventdata)disp([job.Name,' now ',job.State])
Function that runs on the client when job reaches the running state JobRunningFcn	@(job,eventdata)disp([job.Name,' now ',job.State])
Function that runs on the client when job reaches the queued state JobQueuedFcn	@(job,eventdata)disp([job.Name,' now ',job.State])
Function that runs on the client when task reaches the finished state TaskFinishedFcn	@clientTaskCompleted
Function that runs on the client when task reaches the running state TaskRunningFcn	<none>

Create and save a callback function `clientTaskCompleted.m` on the path of the MATLAB client, with the following content. (If you created this function for the previous example, you can use that.)

```
function clientTaskCompleted(task,eventdata)
    disp(['Finished task: ' num2str(task.ID)])
```

Create objects for the cluster, job, and task. Then submit the job. All the callback properties are set from the profile when the objects are created.

```
c = parcluster('MyMJS');
j = createJob(c,'Name','Job_52a');
createTask(j,@rand,1,{2,4});

submit(j)

Job_52a now queued
Job_52a now running
Finished task: 1
Job_52a now finished
```

Tips

- You should avoid running code in your callback functions that might cause conflicts. For example, if every task in a job has a callback that plots its results, there is no guarantee to the order in which the tasks finish, so the plots might overwrite each other. Likewise, the `FinishFcn` callback for a job might be triggered to start before the `FinishFcn` callbacks for all its tasks are complete.

- Submissions made with `batch` use applicable job and task callbacks. Parallel pools can trigger job callbacks defined by their cluster profile.

Job Monitor

In this section...
"Typical Use Cases" on page 5-24
"Manage Jobs Using the Job Monitor" on page 5-24
"Identify Task Errors Using the Job Monitor" on page 5-25

The Job Monitor displays the jobs in the queue for the scheduler determined by your selection of a cluster profile. Open the Job Monitor from the MATLAB desktop on the **Home** tab in the **Environment** section, by selecting **Parallel > Monitor Jobs**.
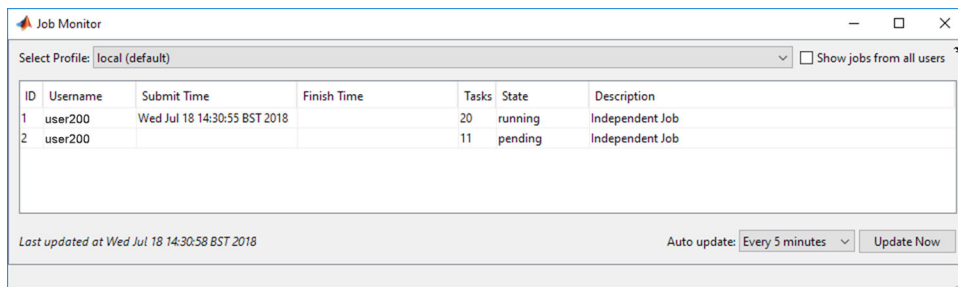
The job monitor lists all the jobs that exist for the cluster specified in the selected profile. You can choose any one of your profiles (those available in your current session Cluster Profile Manager), and whether to display jobs from all users or only your own jobs.

Typical Use Cases

The Job Monitor lets you accomplish many different goals pertaining to job tracking and queue management. Using the Job Monitor, you can:

- Discover and monitor all jobs submitted by a particular user
- Determine the status of a job
- Determine the cause of errors in a job
- Delete old jobs you no longer need
- Create a job object in MATLAB for access to a particular job in the queue

Manage Jobs Using the Job Monitor

Using the Job Monitor you can manage the listed jobs for your cluster. Right-click on any job in the list, and select any of the following options from the context menu. The available options depend on the type of job.

- **Cancel** — Stops a running job and changes its state to `'finished'`. If the job is pending or queued, the state changes to `'finished'` without its ever running. This is the same as the command-line `cancel` function for the job.
- **Delete** — Deletes the job data and removes the job from the queue. This is the same as the command-line `delete` function for the job. Also closes and deletes an interactive pool job.
- **Show Details** — This displays detailed information about the job in the Command Window.

- **Show Errors** — This displays all the tasks that generated an error in that job, with their error properties.
- **Fetch Outputs** — This collects all the task output arguments from the job into the client workspace.

Identify Task Errors Using the Job Monitor

Because the Job Monitor indicates if a job had a run-time error, you can use it to identify the tasks that generated the errors in that job. For example, the following script generates an error because it attempts to perform a matrix inverse on a vector:

```
A = [2 4 6 8];
B = inv(A);
```

If you save this script in a file named `invert_me.m`, you can try to run the script as a batch job on the default cluster:

```
batch('invert_me')
```

When updated after the job runs, the Job Monitor includes the job created by the `batch` command, with an error icon (⚠) for this job. Right-click the job in the list, and select **Show Errors**. For all the tasks with an error in that job, the task information, including properties related to the error, display in the MATLAB command window:

```
Task ID 1 from Job ID 2 Information
===================================

                  State : finished
               Function : @parallel.internal.cluster.executeScript
              StartTime : Tue Jun 28 11:46:28 EDT 2011
        Running Duration : 0 days 0h 0m 1s

- Task Result Properties

        ErrorIdentifier : MATLAB:square
           ErrorMessage : Matrix must be square.
            Error Stack : invert_me (line 2)
```

Programming Tips

In this section...

Program Development Guidelines

When writing code for Parallel Computing Toolbox software, you should advance one step at a time in the complexity of your application. Verifying your program at each step prevents your having to debug several potential problems simultaneously. If you run into any problems at any step along the way, back up to the previous step and reverify your code.

The recommended programming practice for distributed or parallel computing applications is

1   **Run code normally on your local machine.** First verify all your functions so that as you progress, you are not trying to debug the functions and the distribution at the same time. Run your functions in a single instance of MATLAB software on your local computer. For programming suggestions, see "Techniques to Improve Performance".

2   **Decide whether you need an independent or communicating job.** If your application involves large data sets on which you need simultaneous calculations performed, you might benefit from a communicating job with distributed arrays. If your application involves looped or repetitive calculations that can be performed independently of each other, an independent job might be appropriate.

3   **Modify your code for division.** Decide how you want your code divided. For an independent job, determine how best to divide it into tasks; for example, each iteration of a for-loop might define one task. For a communicating job, determine how best to take advantage of parallel processing; for example, a large array can be distributed across all your workers.

4   **Use spmd to develop parallel functionality.** Use spmd with a local pool to develop your functions on several workers in parallel. As you progress and use spmd on the remote cluster, that might be all you need to complete your work.

5   **Run the independent or communicating job with a local scheduler.** Create an independent or communicating job, and run the job using the local scheduler with several local workers. This verifies that your code is correctly set up for batch execution, and in the case of an independent job, that its computations are properly divided into tasks.

6   **Run the independent job on only one cluster node.** Run your independent job with one task to verify that remote distribution is working between your client and the cluster, and to verify proper transfer of additional files and paths.

7   **Run the independent or communicating job on multiple cluster nodes.** Scale up your job to include as many tasks as you need for an independent job, or as many workers as you need for a communicating job.

Note The client session of MATLAB must be running the Java® Virtual Machine (JVM™) to use Parallel Computing Toolbox software. Do not start MATLAB with the `-nojvm` flag.

Current Working Directory of a MATLAB Worker

The current directory of a MATLAB worker at the beginning of its session is

`CHECKPOINTBASE\HOSTNAME_WORKERNAME_mlworker_log\work`

where `CHECKPOINTBASE` is defined in the `mjs_def` file, `HOSTNAME` is the name of the node on which the worker is running, and `WORKERNAME` is the name of the MATLAB worker session.

For example, if the worker named `worker22` is running on host `nodeA52`, and its `CHECKPOINTBASE` value is `C:\TEMP\MJS\Checkpoint`, the starting current directory for that worker session is

`C:\TEMP\mjs\Checkpoint\nodeA52_worker22_mlworker_log\work`

Writing to Files from Workers

When multiple workers attempt to write to the same file, you might end up with a race condition, clash, or one worker might overwrite the data from another worker. This might be likely to occur when:

- There is more than one worker per machine, and they attempt to write to the same file.
- The workers have a shared file system, and use the same path to identify a file for writing.

In some cases an error can result, but sometimes the overwriting can occur without error. To avoid an issue, be sure that each worker or `parfor` iteration has unique access to any files it writes or saves data to. There is no problem when multiple workers read from the same file.

Saving or Sending Objects

Do not use the `save` or `load` function on Parallel Computing Toolbox objects. Some of the information that these objects require is stored in the MATLAB session persistent memory and would not be saved to a file.

Similarly, you cannot send a parallel computing object between parallel computing processes by means of an object's properties. For example, you cannot pass a MATLAB Job Scheduler, job, task, or worker object to MATLAB workers as part of a job's `JobData` property.

Also, system objects (e.g., Java classes, .NET classes, shared libraries, etc.) that are loaded, imported, or added to the Java search path in the MATLAB client, are not available on the workers unless explicitly loaded, imported, or added on the workers, respectively. Other than in the task function code, typical ways of loading these objects might be in `taskStartup`, `jobStartup`, and in the case of workers in a parallel pool, in `poolStartup` and using `pctRunOnAll`.

Using clear functions

Executing

```
clear functions
```

clears all Parallel Computing Toolbox objects from the current MATLAB session. They still remain in the MATLAB Job Scheduler. For information on recreating these objects in the client session, see "Recover Objects" on page 6-11.

Running Tasks That Call Simulink Software

The first task that runs on a worker session that uses Simulink software can take a long time to run, as Simulink is not automatically started at the beginning of the worker session. Instead, Simulink starts up when first called. Subsequent tasks on that worker session will run faster, unless the worker is restarted between tasks.

Using the pause Function

On worker sessions running on Macintosh or UNIX operating systems, `pause(Inf)` returns immediately, rather than pausing. This is to prevent a worker session from hanging when an interrupt is not possible.

Transmitting Large Amounts of Data

Operations that involve transmitting many objects or large amounts of data over the network can take a long time. For example, getting a job's `Tasks` property or the results from all of a job's tasks can take a long time if the job contains many tasks. See also "Attached Files Size Limitations" on page 5-42.

Interrupting a Job

Because jobs and tasks are run outside the client session, you cannot use **Ctrl+C** (^C) in the client session to interrupt them. To control or interrupt the execution of jobs and tasks, use such functions as `cancel`, `delete`, `demote`, `promote`, `pause`, and `resume`.

Speeding Up a Job

You might find that your code runs slower on multiple workers than it does on one desktop computer. This can occur when task startup and stop time is significant relative to the task run time. The most common mistake in this regard is to make the tasks too small, i.e., too fine-grained. Another common mistake is to send large amounts of input or output data with each task. In both of these cases, the time it takes to transfer data and initialize a task is far greater than the actual time it takes for the worker to evaluate the task function.

Control Random Number Streams on Workers

In this section...
"Client and Workers" on page 5-29
"Different Workers" on page 5-30
"Normally Distributed Random Numbers" on page 5-31

The random number generation functions `rand`, `randi`, and `randn` behave differently for parallel calculations compared to your MATLAB client. You can change the behavior of random number generators on parallel workers or on the client to generate reproducible streams of random numbers.

By default, the MATLAB client and MATLAB workers use different random number generators, even if the workers are part of a local cluster on the same machine as the client. The table below summarizes the default settings for the client and workers:

	Generator	Seed	Normal Transform
Client	`'Twister'` or `'mt19937ar'`	0	`'Ziggurat'`
Worker (local or remote)	`'Threefry'` or `'Threefry4x64_20'`	0	`'Inversion'`

For more information about the available generators and normal transforms, see "Choosing a Random Number Generator". Each worker in a cluster draws random numbers from an independent stream with the properties in the table. By default, the random numbers generated on each worker in a `parfor` loop are different from each other and from the random numbers generated on the client.

Note If you have a GPU on your worker, different settings apply to random number streams on the GPU. For more information, see "Random Number Streams on a GPU" on page 8-6.

Client and Workers

If it is necessary to generate the same stream of numbers in the client and workers, you can set one to match the other. You can set the generator algorithm and seed using `rng`.

For example, you might run a script as a batch job on a worker, and need the same generator or sequence as the client. Suppose you start with a script file named `randScript1.m` that contains the line:

```
R = rand(1,4);
```

You can run this script in the client, and then as a batch job on a worker. Notice that the default generated random number sequences in the results are different.

```
randScript1; % In client
R

R =
    0.8147    0.9058    0.1270    0.9134

parallel.defaultClusterProfile('local')
c = parcluster();
```

```
j = batch(c,'randScript1'); % On worker
wait(j);load(j);
R

R =
    0.1349    0.6744    0.9301    0.5332
```

For identical results, you can set the client and worker to use the same generator and seed. Here, the file randScript2.m contains the following code:

```
rng(1,'Threefry');
R = rand(1,4);
```

Now, run the new script in the client and on a worker:

```
randScript2; % In client
R

R =
    0.1404    0.8197    0.1073    0.4131

j = batch(c,'randScript2'); % On worker
wait(j); load(j);
R

R =
    0.1404    0.8197    0.1073    0.4131
```

Different Workers

By default, each worker in a cluster working on the same job has an independent random number stream. If rand, randi, or randn are called in parallel, each worker produces a unique sequence of random numbers.

Note Because rng('shuffle') seeds the random number generator based on the current time, do not use this command to set the random number stream on different workers if you want to ensure independent streams. This is especially true when the command is sent to multiple workers simultaneously, such as inside a parfor, spmd, or a communicating job. For independent streams on the workers, use the default behavior; or if that is not sufficient for your needs, consider using a unique substream on each worker using RandStream.

This example uses two workers in a parallel pool to show they generate unique random number sequences.

```
p = parpool(2);
spmd
    R = rand(1,4); % Different on each worker
end
R{1},R{2}

ans =
    0.1349    0.6744    0.9301    0.5332
ans =
    0.6383    0.5195    0.1398    0.6509

delete(p)
```

If you need all workers to generate the same sequence of numbers, you can set each worker to use the same generator settings:

```
p = parpool(2);
spmd
    rng(0,'Philox'); % Default seed 0.
    R = rand(1,4); % Same on all workers
end
R{1},R{2}

ans =
    0.3655    0.6975    0.1789    0.4549
ans =
    0.3655    0.6975    0.1789    0.4549

delete(p)
```

If you need to control the random numbers at each iteration of a `parfor`-loop, see "Repeat Random Numbers in parfor-Loops" on page 2-69.

Normally Distributed Random Numbers

If you are working with normally distributed random numbers using the `randn` function, you can use the same methods as above using `RandStream` to set the generator type, seed, and normal transformation algorithm on each worker and the client.

For example, suppose the file `randScript3.m` contains the code:

```
stream = RandStream('Threefry','Seed',0,'NormalTransform','Inversion');
RandStream.setGlobalStream(stream);
R = randn(1,7)
```

You can run this code on the client and on a worker in a parallel job (using `batch` or `spmd`) to produce the same sequence of random numbers:

```
R =
    -0.3479    0.1057    0.3969    0.6544    -1.8228    0.9587    0.5360
```

See Also
`RandStream` | `rng`

More About
*   "Repeat Random Numbers in parfor-Loops" on page 2-69
*   "Random Number Streams on a GPU" on page 8-6
*   "Creating and Controlling a Random Number Stream"

Profiling Parallel Code

In this section...
"Profile Parallel Code" on page 5-32
"Analyze Parallel Profile Data" on page 5-34

The parallel profiler provides an extension of the `profile` command and the profile viewer specifically for workers in a parallel pool, to enable you to see how much time each worker spends evaluating each function and how much time communicating or waiting for communications with the other workers. For more information about the standard profiler and its views, see "Profile Your Code to Improve Performance".

For parallel profiling, you use the `mpiprofile` command in a similar way to how you use `profile`.

Profile Parallel Code

This example shows how to profile parallel code using the parallel profiler on workers in a parallel pool.

Create a parallel pool.

```
numberOfWorkers = 3;
pool = parpool(numberOfWorkers);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 3).
```

Collect parallel profile data by enabling `mpiprofile.`

```
mpiprofile on
```

Run your parallel code. For the purposes of this example, use a simple `parfor` loop that iterates over a series of values.

```
values = [5 12 13 1 12 5];
tic;
parfor idx = 1:numel(values)
    u = rand(values(idx)*3e4,1);
    out(idx) = max(conv(u,u));
end
toc

Elapsed time is 31.228931 seconds.
```

After the code completes, view the results from the parallel profiler by calling `mpiprofile viewer`. This action also stops profile data collection.

```
mpiprofile viewer
```

The report shows execution time information for each function that runs on the workers. You can explore which functions take the most time in each worker.

Generally, comparing the workers with the minimum and maximum total execution times is useful. To do so, click **Compare (max vs. min TotalTime)** in the report. In this example, observe that `conv`

executes multiple times and takes significantly longer in one worker than in the other. This observation suggests that the load might not be distributed evenly across the workers.

Parallel Profile Summary *Generated 23-Aug-2019 16:29:24 using real time.*

Showing **all functions** called in worker **2** compared with worker **1** (worker 1 has min total time)

Automatic Comparison Selection	Manual Comparison Selection	Show Figures (all workers):	
Compare (max vs. min TotalTime)	Go to worker: max Total Time		No Plot
Compare (max vs. min CommTime)	Compare with: min Total Time		Plot Time Histograms
			Plot All Per Worker Communication
			Plot Communication Time Per Worker

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

Function Name comparison with worker 1	Calls	Total Time	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Inter-worker Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
remoteParallelFunction	3	31.200 s	0.008 s	0 s	0 s	0 b	100.0%	
remoteParallelFunction	3	3.579 s	0.008 s	0 s	0 s	0 b	100.0%	
...>make_general_channel/channel_general	2	31.172 s	0.017 s	0 s	0 s	0 b	100.0%	
...>make_general_channel/channel_general	2	3.548 s	0.007 s	0 s	0 s	0 b	100.0%	
conv	2	31.155 s	31.155 s	0 s	0 s	0 b	100.0%	
conv	2	3.541 s	3.541 s	0 s	0 s	0 b	100.0%	

- If you do not know the workload of each iteration, then a good practice is to randomize the iterations, such as in the following sample code.

```
values = values(randperm(numel(values)));
```

- If you do know the workload of each iteration in your `parfor` loop, then you can use `parforOptions` to control the partitioning of iterations into subranges for the workers. For more information, see `parforOptions`.

In this example, the greater `values(idx)` is, the more computationally intensive the iteration is. Create a set of `parfor` options to divide the `parfor` iterations into subranges of size 2 so that the workload is better distributed.

```
opts = parforOptions(pool,"RangePartitionMethod","fixed","SubrangeSize",2);
```

Enable the parallel profiler.

```
mpiprofile on
```

Run the same code as before. To use the `parfor` options, pass them to the second input argument of `parfor`.

```
values = [5 12 13 1 12 5];
tic;
parfor (idx = 1:numel(values),opts)
    u = rand(values(idx)*3e4,1);
    out(idx) = max(conv(u,u));
end
toc
```

```
Elapsed time is 21.077027 seconds.
```

Visualize the parallel profiler results.

```
mpiprofile viewer
```

In the report, select **Compare (max vs. min TotalTime)** to compare the workers with the minimum and maximum total execution times. Observe that this time, the multiple executions of `conv` take a similar amount of time in all workers. The workload is now better distributed.

Analyze Parallel Profile Data

The profiler collects information about the execution of code on each worker and the communications between the workers. Such information includes:

- Execution time of each function on each worker.
- Execution time of each line of code in each function.
- Amount of data transferred between each worker.
- Amount of time each worker spends waiting for communications.

The remainder of this section is an example that illustrates some of the features of the parallel profile viewer. The example profiles parallel execution of matrix multiplication of distributed arrays on a parallel pool of cluster workers.

```
parpool

Starting parallel pool (parpool) using the 'MyCluster' profile ...
Connected to the parallel pool (number of workers: 64).

R1 = rand(5e4,'distributed');
R2 = rand(5e4,'distributed');
mpiprofile on
```

```
R = R1*R2;
mpiprofile viewer
```

The last command opens the Profiler window, first showing the Parallel Profile Summary (or function summary report) for worker 1.

The function summary report displays the data for each function executed on a worker in sortable columns with the following headers:

Column Header	Description
Calls	How many times the function was called on this worker
Total Time	The total amount of time this worker spent executing this function
Self Time	The time this worker spent inside this function, not within children or local functions
Total Comm Time	The total time this worker spent transferring data with other workers, including waiting time to receive data
Self Comm Waiting Time	The time this worker spent during this function waiting to receive data from other workers
Total Inter-worker Data	The amount of data transferred to and from this worker for this function
Computation Time Ratio	The ratio of time spent in computation for this function vs. total time (which includes communication time) for this function
Total Time Plot	Bar graph showing relative size of Self Time, Self Comm Waiting Time, and Total Time for this function on this worker

Select the name of any function in the list for more details about the execution of that function. The function detail report for `codistributor1d.hMtimesImpl` includes this listing:

Parents (calling functions)

Function Name	Function Type	Calls
@codistributed/private/mtimesImpl	function	1

Lines where the most time was spent.

Line Number	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
121	`LPC = LPC + LPA*LPB(k, :);`	63	137.987 s	0 b	0 b	0 s	0 s	81.5%	▬▬▬▬
119	`LPA = labSendReceive(to, from,...`	63	27.883 s	18.34 Gb	18.34 Gb	3.341 s	22.523 s	16.5%	▪
114	`LPC(:, :) = LPA*LPB(k, :);`	1	3.262 s	0 b	0 b	0 s	0 s	1.9%	▎
130	`end  % End of hMtimesImpl`	1	0.056 s	0 b	0 b	0 s	0 s	0.0%	
120	`k = codistrA.globalIndices(2, ...`	63	0.043 s	0 b	0 b	0 s	0 s	0.0%	
All other lines			0.053 s	0 b	0 b	0 s	0 s	0.0%	
Totals			169.284 s	18.34 Gb	18.34 Gb	3.341 s	22.523 s	100%	

The code that the report displays comes from the client. If the code has changed on the client since the communicating job ran on the workers, or if the workers are running a different version of the functions, the display might not accurately reflect what actually executed.

You can display information for each worker, or use the comparison controls to display information for several workers simultaneously. Two buttons provide **Automatic Comparison Selection**, so you can compare the data from the workers that took the most versus the least amount of time to execute the code, or data from the workers that spent the most versus the least amount of time in performing interworker communication. **Manual Comparison Selection** allows you to compare data from specific workers or workers that meet certain criteria.

The following listing from the summary report shows the result of using the **Automatic Comparison Selection** of **Compare (max vs. min TotalTime)**. The comparison shows data from worker 50 compared to worker 62 because these are the workers that spend the most versus least amount of time executing the code.

Function Name	Function Type	Calls
@codistributed/private/mtimesImpl	function	1

Lines where the most time was spent including the top 5 code lines from the comparison worker (maroon)

Line Number (for worker 50 and 62)	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
121	LPC = LPC + LPA*LPB(k, :);	63 *63*	136.204 s *127.895 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	78.4% *76.0%*	
119	LPA = labSendReceive(to, from,...	63 *63*	34.335 s *37.679 s*	18.34 Gb *18.34 Gb*	18.34 Gb *18.34 Gb*	0.095 s *22.925 s*	32.579 s *13.525 s*	19.8% *22.4%*	
114	LPC(:, :) = LPA*LPB(k, :);	1 *1*	3.032 s *2.567 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	1.7% *1.5%*	
120	k = codistrA.globalIndices(2, ...	63 *63*	0.046 s *0.048 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	0.0% *0.0%*	
130	end % End of hMtimesImpl	1 *1*	0.041 s *0.019 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	0.0% *0.0%*	
111	codistrC = codistributor1d(2	1	0.041 s	0 b	0 b	0 s	0 s	0.0%	

The following figure shows a summary of all the functions executed during the profile collection time. The **Manual Comparison Selection** of **max Time Aggregate** means that data is considered from all the workers for all functions to determine which worker spent the maximum time on each function. Next to each function's name is the worker that took the longest time to execute that function. The other columns list the data from that worker.

Parallel Profile Summary *Generated 27-Aug-2019 12:01:47 using real time.*

Showing all functions called in worker max time

Automatic Comparison Selection	Manual Comparison Selection	Show Figures (all workers):	No Plot
Compare (max vs. min TotalTime)	Go to worker: max Time Aggregate		Plot Time Histograms
Compare (max vs. min CommTime)	Compare with: None		Plot All Per Worker Communication
			Plot Communication Time Per Worker

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

Function Name	Calls	Total Time	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Inter-worker Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
remoteBlockExecution (worker 64)	7	174.647 s	0.023 s	37.588 s	0 s	36.67 Gb	78.5%	
distributedSpmdWrapper>iInnerWrapper (worker 50)	1	173.832 s	0.006 s	32.674 s	0 s	36.67 Gb	81.2%	
spmd_feval_fcn>get_f/body (worker 50)	1	173.832 s	0 s	32.674 s	0 s	36.67 Gb	81.2%	
codistributed.mtimes (worker 50)	1	173.826 s	0.018 s	32.674 s	0 s	36.67 Gb	81.2%	
@codistributed/private/mtimesImpl (worker 50)	1	173.808 s	0.022 s	32.674 s	0 s	36.67 Gb	81.2%	
codistributor1d.hMtimesImpl (worker 50)	1	173.741 s	173.640 s	32.674 s	0.095 s	36.67 Gb	81.2%	

The next figure shows a summary report for the workers that spend the most versus least time for each function. A **Manual Comparison Selection** of **max Time Aggregate** against **min Time >0 Aggregate** generated this summary. Both aggregate settings indicate that the profiler should consider data from all workers for all functions, for both maximum and minimum. This report lists the data for `codistributor1d.hMtimesImpl` from workers 50 and 62, because they spent the maximum and minimum times on this function. Similarly, other functions are listed.

Parallel Profile Summary *Generated 27-Aug-2019 12:03:46 using real time.*

Showing **all functions** called in worker **max time** compared with worker **min time**

Automatic Comparison Selection	Manual Comparison Selection	Show Figures (all workers):	No Plot
Compare (max vs. min TotalTime)	Go to worker: max Time Aggregate ⌄		Plot Time Histograms
Compare (max vs. min CommTime)	Compare with: min Time >0 Aggregate ⌄		Plot All Per Worker Communication
			Plot Communication Time Per Worker

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

Function Name comparison with worker min time	Calls	Total Time	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Inter-worker Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
remoteBlockExecution (worker 64)	7	174.647 s	0.023 s	37.588 s	0 s	36.67 Gb	78.5%	
remoteBlockExecution (worker 1)	7	173.982 s	0.020 s	25.865 s	0 s	36.67 Gb	85.1%	
distributedSpmdWrapper>iInnerWrapper (worker 50)	1	173.832 s	0.006 s	32.674 s	0 s	36.67 Gb	81.2%	
distributedSpmdWrapper>iInnerWrapper (worker 62)	1	168.353 s	0.005 s	36.450 s	0 s	36.67 Gb	78.3%	
spmd_feval_fcn>get_f/body (worker 50)	1	173.832 s	0 s	32.674 s	0 s	36.67 Gb	81.2%	
spmd_feval_fcn>get_f/body (worker 62)	1	168.354 s	0.001 s	36.450 s	0 s	36.67 Gb	78.3%	
codistributed.mtimes (worker 50)	1	173.826 s	0.018 s	32.674 s	0 s	36.67 Gb	81.2%	
codistributed.mtimes (worker 62)	1	168.347 s	0.021 s	36.450 s	0 s	36.67 Gb	78.3%	
@codistributed/private/mtimesImpl (worker 50)	1	173.808 s	0.022 s	32.674 s	0 s	36.67 Gb	81.2%	
@codistributed/private/mtimesImpl (worker 62)	1	168.326 s	0.014 s	36.450 s	0 s	36.67 Gb	78.3%	
codistributor1d.hMtimesImpl (worker 50)	1	173.741 s	173.640 s	32.674 s	0.095 s	36.67 Gb	81.2%	
codistributor1d.hMtimesImpl (worker 62)	1	168.267 s	168.183 s	36.450 s	22.925 s	36.67 Gb	78.3%	

Select a function name in the summary listing of a comparison to get a detailed comparison. The detailed comparison for `codistributor1d.hMtimesImpl` looks like this, displaying line-by-line data from both workers:

Lines where the most time was spent including the top 5 code lines from the comparison worker (maroon)

Line Number (for worker 50 and *62*)	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
[121](#)	`LPC = LPC + LPA*LPB(k, :);`	63 *63*	136.204 s *127.895 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	78.4% *76.0%*	
[119](#)	`LPA = labSendReceive(to, from,...`	63 *63*	34.335 s *37.679 s*	18.34 Gb *18.34 Gb*	18.34 Gb *18.34 Gb*	0.095 s *22.925 s*	32.579 s *13.525 s*	19.8% *22.4%*	
[114](#)	`LPC(:, :) = LPA*LPB(k, :);`	1 *1*	3.032 s *2.567 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	1.7% *1.5%*	
[120](#)	`k = codistrA.globalIndices(2, ...`	63 *63*	0.046 s *0.048 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	0.0% *0.0%*	
[130](#)	`end  % End of hMtimesImpl`	1 *1*	0.041 s *0.019 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	0.0% *0.0%*	
[111](#)	`codistrC = codistributor1d(2, ...`	1 *1*	0.041 s *0.030 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	0.0% *0.0%*	
All other lines			0.083 s	0 b	0 b	0 s	0 s	0.0%	
Totals			173.741 s *168.267 s*	18.34 Gb *18.34 Gb*	18.34 Gb *18.34 Gb*	0.095 s *22.925 s*	32.579 s *13.525 s*	100%	

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

To see plots of communication data, select **Plot All Per Worker Communication** in the **Show Figures** menu. The top portion of the plot view report plots how much data each worker receives from each other worker for all functions.

Plot View
Generated 11-Oct-2019 09:41:23 using performance time.
Per Worker Communication Images

Show Figures (all workers):

| No Plot |
| Plot Time Histograms |
| Plot All Per Worker Communication |
| Plot Communication Time Per Worker |

All workers Data Received Per Worker for all functions
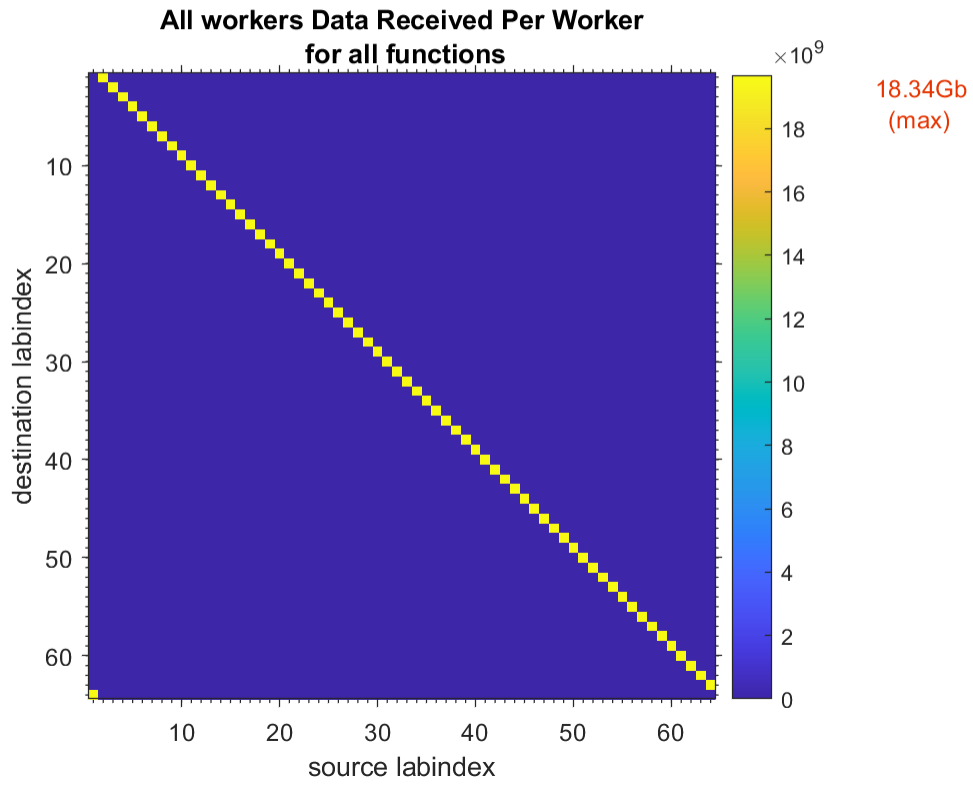
To see only a plot of interworker communication times, select **Plot Communication Time Per Worker** in the **Show Figures** menu.

Plot View
Generated 11-Oct-2019 09:44:40 using performance time.
Comm Time Per Worker Image

Show Figures (all workers):

No Plot
Plot Time Histograms
Plot All Per Worker Communication
Plot Communication Time Per Worker

All workers Receive Comm Time Per Worker for all functions
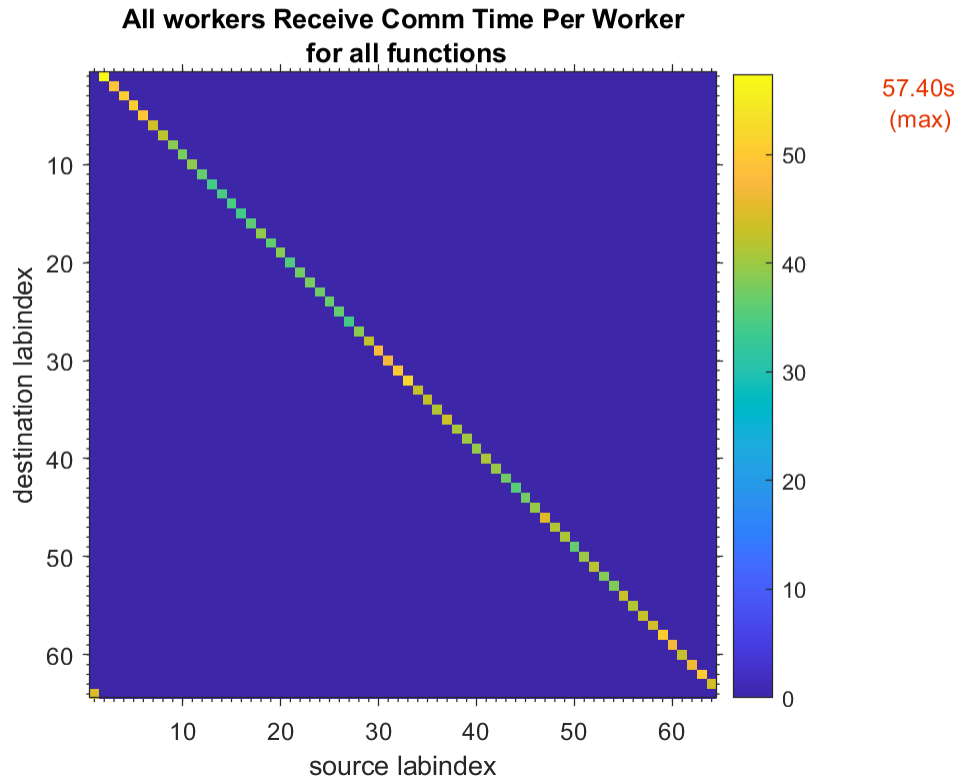
Plots like those in the previous two figures can help you determine the best way to balance work among your workers, perhaps by altering the partition scheme of your codistributed arrays.

Troubleshooting and Debugging

In this section...
"Attached Files Size Limitations" on page 5-42
"File Access and Permissions" on page 5-42
"No Results or Failed Job" on page 5-43
"Connection Problems Between the Client and MATLAB Job Scheduler" on page 5-44
"SFTP Error: Received Message Too Long" on page 5-44

Attached Files Size Limitations

The combined size of all attached files for a job is limited to 4 GB.

File Access and Permissions

Ensuring That Workers on Windows Operating Systems Can Access Files

By default, a worker on a Windows operating system is installed as a service running as `LocalSystem`, so it does not have access to mapped network drives.

Often a network is configured to not allow services running as `LocalSystem` to access UNC or mapped network shares. In this case, you must run the mjs service under a different user with rights to log on as a service. See the section "Set the User" (MATLAB Parallel Server) in the MATLAB Parallel Server System Administrator's Guide.

Task Function Is Unavailable

If a worker cannot find the task function, it returns the error message

```
Error using ==> feval
      Undefined command/function 'function_name'.
```

The worker that ran the task did not have access to the function `function_name`. One solution is to make sure the location of the function's file, `function_name.m`, is included in the job's `AdditionalPaths` property. Another solution is to transfer the function file to the worker by adding `function_name.m` to the `AttachedFiles` property of the job.

Load and Save Errors

If a worker cannot save or load a file, you might see the error messages

```
??? Error using ==> save
Unable to write file myfile.mat: permission denied.
??? Error using ==> load
Unable to read file myfile.mat: No such file or directory.
```

In determining the cause of this error, consider the following questions:

- What is the worker's current folder?
- Can the worker find the file or folder?

- What user is the worker running as?
- Does the worker have permission to read or write the file in question?

Tasks or Jobs Remain in Queued State

A job or task might get stuck in the queued state. To investigate the cause of this problem, look for the scheduler's logs:

- Platform LSF schedulers might send emails with error messages.
- Microsoft Windows HPC Server (including CCS), LSF®, PBS Pro, and TORQUE save output messages in a debug log. See the `getDebugLog` reference page.
- If using a generic scheduler, make sure the submit function redirects error messages to a log file.

Possible causes of the problem are:

- The MATLAB worker failed to start due to licensing errors, the executable is not on the default path on the worker machine, or is not installed in the location where the scheduler expected it to be.
- MATLAB could not read/write the job input/output files in the scheduler's job storage location. The storage location might not be accessible to all the worker nodes, or the user that MATLAB runs as does not have permission to read/write the job files.
- If using a generic scheduler:

  - The environment variable `PARALLEL_SERVER_DECODE_FUNCTION` was not defined before the MATLAB worker started.
  - The decode function was not on the worker's path.

No Results or Failed Job

Task Errors

If your job returned no results (i.e., `fetchOutputs(job)` returns an empty cell array), it is probable that the job failed and some of its tasks have their `Error` properties set.

You can use the following code to identify tasks with error messages:

```
errmsgs = get(yourjob.Tasks, {'ErrorMessage'});
nonempty = ~cellfun(@isempty, errmsgs);
celldisp(errmsgs(nonempty));
```

This code displays the nonempty error messages of the tasks found in the job object `yourjob`.

Debug Logs

If you are using a supported third-party scheduler, you can use the `getDebugLog` function to read the debug log from the scheduler for a particular job or task.

For example, find the failed job on your LSF scheduler, and read its debug log:

```
c = parcluster('my_lsf_profile')
failedjob = findJob(c, 'State', 'failed');
message = getDebugLog(c, failedjob(1))
```

Connection Problems Between the Client and MATLAB Job Scheduler

For testing connectivity between the client machine and the machines of your compute cluster, you can use Admin Center. For more information about Admin Center, including how to start it and how to test connectivity, see "Start Admin Center" (MATLAB Parallel Server) and "Test Connectivity" (MATLAB Parallel Server).

Detailed instructions for other methods of diagnosing connection problems between the client and MATLAB Job Scheduler can be found in some of the Bug Reports listed on the MathWorks Web site.

The following sections can help you identify the general nature of some connection problems.

Client Cannot See the MATLAB Job Scheduler

If you cannot locate or connect to your MATLAB Job Scheduler with `parcluster`, the most likely reasons for this failure are:

- The MATLAB Job Scheduler is currently not running.
- Firewalls do not allow traffic from the client to the MATLAB Job Scheduler.
- The client and the MATLAB Job Scheduler are not running the same version of the software.
- The client and the MATLAB Job Scheduler cannot resolve each other's short hostnames.
- The MATLAB Job Scheduler is using a nondefault `BASE_PORT` setting as defined in the `mjs_def` file, and the `Host` property in the cluster profile does not specify this port.

MATLAB Job Scheduler Cannot See the Client

If a warning message says that the MATLAB Job Scheduler cannot open a TCP connection to the client computer, the most likely reasons for this are

- Firewalls do not allow traffic from the MATLAB Job Scheduler to the client.
- The MATLAB Job Scheduler cannot resolve the short hostname of the client computer. Use `pctconfig` to change the hostname that the MATLAB Job Scheduler will use for contacting the client.

SFTP Error: Received Message Too Long

The example code for generic schedulers with non-shared file systems contacts an sftp server to handle the file transfer to and from the cluster's file system. This use of sftp is subject to all the normal sftp vulnerabilities. One problem that can occur results in an error message similar to this:

```
Caused by:
    Error using ==> RemoteClusterAccess>RemoteClusterAccess.waitForChoreToFinishOrError at 780
    The following errors occurred in the
        com.mathworks.toolbox.distcomp.clusteraccess.UploadFilesChore:
    Could not send Job3.common.mat for job 3:
    One of your shell's init files contains a command that is writing to stdout,
        interfering with sftp. Access help
    com.mathworks.toolbox.distcomp.remote.spi.plugin.SftpExtraBytesFromShellException:
    One of your shell's init files contains a command that is writing to stdout,
        interfering with sftp.
    Find and wrap the command with a conditional test, such as

        if ($?TERM != 0) then
            if ("$TERM" != "dumb") then
                /your command/
            endif
        endif

    : 4: Received message is too long: 1718579037
```

The telling symptom is the phrase "`Received message is too long:`" followed by a very large number.

The sftp server starts a shell, usually bash or tcsh, to set your standard read and write permissions appropriately before transferring files. The server initializes the shell in the standard way, calling files like .bashrc and .cshrc. This problem happens if your shell emits text to standard out when it starts. That text is transferred back to the sftp client running inside MATLAB, and is interpreted as the size of the sftp server's response message.

To work around this error, locate the shell startup file code that is emitting the text, and either remove it or bracket it within `if` statements to see if the sftp server is starting the shell:

```
if ($?TERM != 0) then
    if ("$TERM" != "dumb") then
        /your command/
    endif
endif
```

You can test this outside of MATLAB with a standard UNIX or Windows sftp command-line client before trying again in MATLAB. If the problem is not fixed, the error message persists:

```
> sftp yourSubmitMachine
Connecting to yourSubmitMachine...
Received message too long 1718579042
```

If the problem is fixed, you should see:

```
> sftp yourSubmitMachine
Connecting to yourSubmitMachine...
```

Big Data Workflow Using Tall Arrays and Datastores

In this section...
"Running Tall Arrays in Parallel" on page 5-47
"Use mapreducer to Control Where Your Code Runs" on page 5-47
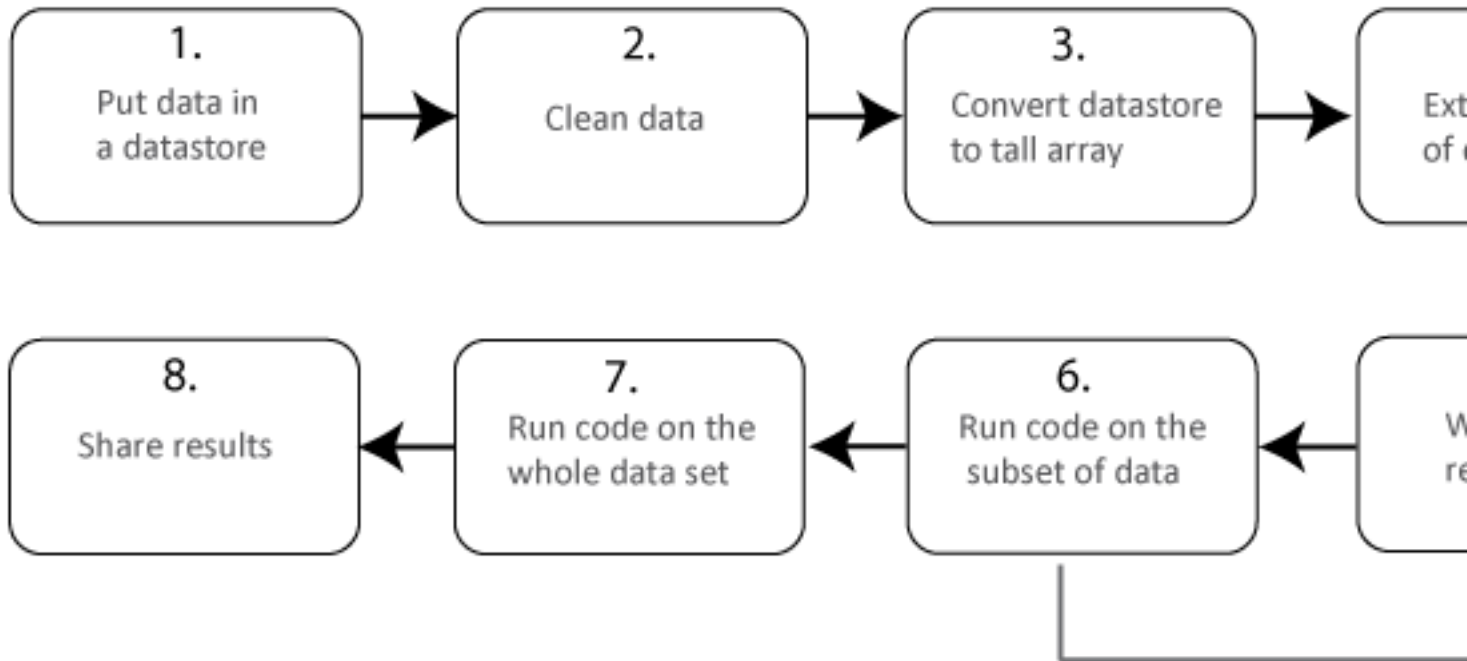
The illustration shows a typical workflow that uses tall arrays to analyze a large data set. In this workflow, you analyze a small subset of the data before scaling up to analyze the entire data set. Parallel computing can help you scale up from steps six to seven. That is, after checking that your code works on the small data set, run it on the whole data set. You can use MATLAB to enhance this workflow.

Problem	Solution	Required Products	More Information
Is your data too big?	To work with out-of-memory data with any number of rows, use tall arrays. This workflow is well suited to data analytics and machine learning.	MATLAB	"Tall Arrays for Out-of-Memory Data"
	Use tall arrays in parallel on your local machine.	MATLAB Parallel Computing Toolbox	"Use Tall Arrays on a Parallel Pool" on page 5-49

Problem	Solution	Required Products	More Information
	Use tall arrays in parallel on your cluster.	MATLAB Parallel Computing Toolbox MATLAB Parallel Server	To use tall arrays on a Hadoop cluster, see "Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52 For all other types of cluster, use a non-local cluster profile to set up a parallel pool. For an example, see "Use Tall Arrays on a Parallel Pool" on page 5-49
	If your data is large in multiple dimensions, use `distributed` instead.	MATLAB Parallel Computing Toolbox MATLAB Parallel Server	"Distributing Arrays to Parallel Workers" on page 3-10

Running Tall Arrays in Parallel

Parallel Computing Toolbox can immediately speed up your tall array calculations by using the full processing power of multicore computers to execute applications with a parallel pool of workers. If you already have Parallel Computing Toolbox installed, then you probably do not need to do anything special to take advantage of these capabilities. For more information about using tall arrays with Parallel Computing Toolbox, see "Use Tall Arrays on a Parallel Pool" on page 5-49.

Use mapreducer to Control Where Your Code Runs

When you execute tall arrays, the default execution environment uses either the local MATLAB session, or a local parallel pool if you have Parallel Computing Toolbox. The default pool uses local workers, typically one worker for each core in your machine. Use the `mapreducer` function to change the execution environment of tall arrays to use a different cluster.

One of the benefits of developing your algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, then use `mapreducer` to scale up and take advantage of the capabilities offered by Parallel Computing Toolbox and MATLAB Parallel Server.

See Also
datastore | gather | mapreducer | tall

Related Examples
• "Use Tall Arrays on a Parallel Pool" on page 5-49

• "Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52

• "Tall Arrays for Out-of-Memory Data"

• "Choose a Parallel Computing Solution" on page 1-16

More About

- "Datastore"

Use Tall Arrays on a Parallel Pool

If you have Parallel Computing Toolbox, you can use tall arrays in your local MATLAB session, or on a local parallel pool. You can also run tall array calculations on a cluster if you have MATLAB Parallel Server installed. This example uses the workers in a local cluster on your machine. You can develop code locally, and then scale up, to take advantage of the capabilities offered by Parallel Computing Toolbox and MATLAB Parallel Server without having to rewrite your algorithm. See also "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46.

Create a datastore and convert it into a tall table.

```
ds = datastore('airlinesmall.csv');
varnames = {'ArrDelay', 'DepDelay'};
ds.SelectedVariableNames = varnames;
ds.TreatAsMissing = 'NA';
```

If you have Parallel Computing Toolbox installed, when you use the `tall` function, MATLAB automatically starts a parallel pool of workers, unless you turn off the default parallel pool preference. The default cluster uses local workers on your machine.

Note If you want to turn off automatically opening a parallel pool, change your parallel preferences. If you turn off the **Automatically create a parallel pool** option, then you must explicitly start a pool if you want the `tall` function to use it for parallel processing. See "Specify Your Parallel Preferences" on page 5-9.

If you have Parallel Computing Toolbox, you can run the same code as the MATLAB tall table example and automatically execute it in parallel on the workers of your local machine.

Create a tall table `tt` from the datastore.

```
tt = tall(ds)

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

tt =

  M×2 tall table

    ArrDelay    DepDelay
    _____    _____

      8           12
      8            1
     21           20
     13           12
      4           -1
     59           63
      3           -2
     11           -1
      :            :
      :            :
```

The display indicates that the number of rows, M, is not yet known. M is a placeholder until the calculation completes.

Extract the arrival delay `ArrDelay` from the tall table. This action creates a new tall array variable to use in subsequent calculations.

```
a = tt.ArrDelay;
```

You can specify a series of operations on your tall array, which are not executed until you call `gather`. Doing so enables you to batch up commands that might take a long time. For example, calculate the mean and standard deviation of the arrival delay. Use these values to construct the upper and lower thresholds for delays that are within 1 standard deviation of the mean.

```
m = mean(a,'omitnan');
s = std(a,'omitnan');
one_sigma_bounds = [m-s m m+s];
```

Use `gather` to calculate `one_sigma_bounds`, and bring the answer into memory.

```
sig1 = gather(one_sigma_bounds)

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4.5 sec
Evaluation completed in 6.3 sec

sig1 =

  -23.4572    7.1201   37.6975
```

You can specify multiple inputs and outputs to `gather` if you want to evaluate several things at once. Doing so is faster than calling `gather` separately on each tall array . As an example, calculate the minimum and maximum arrival delay.

```
[max_delay, min_delay] = gather(max(a),min(a))


max_delay =

        1014

min_delay =

    -64
```

If you want to develop in serial and not use local workers or your specified cluster, enter the following command.

```
mapreducer(0);
```

If you use `mapreducer` to change the execution environment after creating a tall array, then the tall array is invalid and you must recreate it. To use local workers or your specified cluster again, enter the following command.

```
mapreducer(gcp);
```

Note One of the benefits of developing algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, and then use `mapreducer` to scale up to a cluster, without needing to rewrite your algorithm. For an example, see "Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52.

See Also
datastore | gather | mapreducer | parpool | table | tall

Related Examples
- "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46
- "Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52
- "Tall Arrays for Out-of-Memory Data"

More About
- "Datastore"

Use Tall Arrays on a Spark Enabled Hadoop Cluster

Creating and Using Tall Tables

This example shows how to modify a MATLAB example of creating a tall table to run on a Spark enabled Hadoop® cluster. You can use this tall table to create tall arrays and calculate statistical properties. You can develop code locally and then scale up, to take advantage of the capabilities offered by Parallel Computing Toolbox and MATLAB Parallel Server without having to rewrite your algorithm. See also "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46 and "Configure a Hadoop Cluster" (MATLAB Parallel Server)

First, you must set environment variables and cluster properties as appropriate for your specific Spark enabled Hadoop cluster configuration. See your system administrator for the values for these and other properties necessary for submitting jobs to your cluster.

```matlab
setenv('HADOOP_HOME', '/path/to/hadoop/install')
setenv('SPARK_HOME', '/path/to/spark/install');
cluster = parallel.cluster.Hadoop;

% Optionally, if you want to control the exact number of workers:
cluster.SparkProperties('spark.executor.instances') = '16';

mapreducer(cluster);
```

Note In the setup step, you use `mapreducer` to set the cluster execution environment. In the next step, you create a tall array. If you modify or delete the cluster execution environment after creating a tall array, then the tall array is invalid and you must recreate it.

Note If you want to develop in serial and not use local workers, enter the following command.

```matlab
mapreducer(0);
```

After setting your environment variables and cluster properties, you can run the MATLAB tall table example on the Spark enabled Hadoop cluster instead of on your local machine. Create a datastore and convert it into a tall table. MATLAB automatically starts a Spark job to run subsequent calculations on the tall table.

```matlab
ds = datastore('airlinesmall.csv');
varnames = {'ArrDelay', 'DepDelay'};
ds.SelectedVariableNames = varnames;
ds.TreatAsMissing = 'NA';
```

Create a tall table `tt` from the datastore.

```matlab
tt = tall(ds)

Starting a Spark Job on the Hadoop cluster. This could take a few minutes ...done.

tt =

  M×2 tall table
```

ArrDelay	DepDelay
8	12
8	1
21	20
13	12
4	-1
59	63
3	-2
11	-1
:	:
:	:

The display indicates that the number of rows, M, is not yet known. M is a placeholder until the calculation completes.

Extract the arrival delay ArrDelay from the tall table. This action creates a new tall array variable to use in subsequent calculations.

```
a = tt.ArrDelay;
```

You can specify a series of operations on your tall array, which are not executed until you call gather. Doing so allows you to batch up commands that might take a long time. As an example, calculate the mean and standard deviation of the arrival delay. Use these values to construct the upper and lower thresholds for delays that are within 1 standard deviation of the mean.

```
m = mean(a,'omitnan');
s = std(a,'omitnan');
one_sigma_bounds = [m-s m m+s];
```

Use gather to calculate one_sigma_bounds, and bring the answer into memory.

```
sig1 = gather(one_sigma_bounds)

Evaluating tall expression using the Spark Cluster:
- Pass 1 of 1: Completed in 0.95 sec
Evaluation completed in 1.3 sec

sig1 =

  -23.4572    7.1201   37.6975
```

You can specify multiple inputs and outputs to gather if you want to evaluate several things at once. Doing so is faster than calling gather separately on each tall array. For example, calculate the minimum and maximum arrival delay.

```
[max_delay, min_delay] = gather(max(a),min(a))


max_delay =

      1014

min_delay =

  -64
```

> **Note** These examples take more time to complete the first time if MATLAB is starting on the cluster workers.

When using tall arrays on a Spark enabled Hadoop cluster, compute resources from the Hadoop cluster will be reserved for the lifetime of the mapreducer execution environment. To clear these resources, you must delete the mapreducer:

```
delete(gcmr);
```

Alternatively, you can change to a different execution environment, for example:

```
mapreducer(0);
```

See Also
`datastore` | `gather` | `mapreducer` | `parallel.cluster.Hadoop` | `table` | `tall`

Related Examples
- "Big Data Workflow Using Tall Arrays and Datastores" on page 5-46
- "Use Tall Arrays on a Parallel Pool" on page 5-49
- "Configure a Hadoop Cluster" (MATLAB Parallel Server)
- "Tall Arrays for Out-of-Memory Data"
- "Read and Analyze Hadoop Sequence File"

More About
- "Datastore"

Run mapreduce on a Parallel Pool

In this section...
"Start Parallel Pool" on page 5-55
"Compare Parallel mapreduce" on page 5-55

Start Parallel Pool

If you have Parallel Computing Toolbox installed, execution of `mapreduce` can open a parallel pool on the cluster specified by your default profile, for use as the execution environment.

You can set your parallel preferences so that a pool does not automatically open. In this case, you must explicitly start a pool if you want `mapreduce` to use it for parallelization of its work. See "Specify Your Parallel Preferences" on page 5-9.

For example, the following conceptual code starts a pool, and some time later uses that open pool for the `mapreducer` configuration.

```
p = parpool('local',n);
mr = mapreducer(p);
outds = mapreduce(tds,@MeanDistMapFun,@MeanDistReduceFun,mr)
```

Note `mapreduce` can run on any cluster that supports parallel pools. The examples in this topic use a local cluster, which works for all Parallel Computing Toolbox installations.

Compare Parallel mapreduce

The following example calculates the mean arrival delay from a datastore of airline data. First it runs `mapreduce` in the MATLAB client session, then it runs in parallel on a local cluster. The `mapreducer` function explicitly controls the execution environment.

Begin by starting a parallel pool on a local cluster.

```
p = parpool('local',4);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

Create two MapReducer objects for specifying the different execution environments for `mapreduce`.

```
inMatlab = mapreducer(0);
inPool = mapreducer(p);
```

Create and preview the datastore. The data set used in this example is available in *matlabroot*/toolbox/matlab/demos.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames','ArrDelay','ReadSize',1000);
preview(ds)

    ArrDelay
    _____
```

```
        8
        8
       21
       13
        4
       59
        3
       11
```

Next, run the `mapreduce` calculation in the MATLAB client session. The map and reduce functions are available in *matlabroot*/toolbox/matlab/demos.

```
meanDelay = mapreduce(ds,@meanArrivalDelayMapper,@meanArrivalDelayReducer,inMatlab);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map    0% Reduce    0%
Map   10% Reduce    0%
Map   20% Reduce    0%
Map   30% Reduce    0%
Map   40% Reduce    0%
Map   50% Reduce    0%
Map   60% Reduce    0%
Map   70% Reduce    0%
Map   80% Reduce    0%
Map   90% Reduce    0%
Map  100% Reduce  100%
```

```
readall(meanDelay)
```

```
         Key               Value

    _____    _____

    'MeanArrivalDelay'    [7.1201]
```

Then, run the calculation on the current parallel pool. Note that the output text indicates a parallel `mapreduce`.

```
meanDelay = mapreduce(ds,@meanArrivalDelayMapper,@meanArrivalDelayReducer,inPool);
```

```
Parallel mapreduce execution on the parallel pool:
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map    0% Reduce    0%
Map  100% Reduce   50%
Map  100% Reduce  100%
```

```
readall(meanDelay)
```

```
         Key               Value

    _____    _____

    'MeanArrivalDelay'    [7.1201]
```

With this relatively small data set, a performance improvement with the parallel pool is not likely. This example is to show the mechanism for running `mapreduce` on a parallel pool. As the data set grows, or the map and reduce functions themselves become more computationally intensive, you

might expect to see improved performance with the parallel pool, compared to running `mapreduce` in the MATLAB client session.

Note When running parallel `mapreduce` on a cluster, the order of the key-value pairs in the output is different compared to running `mapreduce` in MATLAB. If your application depends on the arrangement of data in the output, you must sort the data according to your own requirements.

See Also

Functions
`datastore` | `mapreduce` | `mapreducer`

Related Examples

- "Getting Started with MapReduce"
- "Run mapreduce on a Hadoop Cluster" on page 5-58

More About

- "MapReduce"
- "Datastore"

Run mapreduce on a Hadoop Cluster

In this section...
"Cluster Preparation" on page 5-58
"Output Format and Order" on page 5-58
"Calculate Mean Delay" on page 5-58

Cluster Preparation

Before you can run `mapreduce` on a Hadoop cluster, make sure that the cluster and client machine are properly configured. Consult your system administrator, or see "Configure a Hadoop Cluster" (MATLAB Parallel Server).

Output Format and Order

When running `mapreduce` on a Hadoop cluster with binary output (the default), the resulting `KeyValueDatastore` points to Hadoop Sequence files, instead of binary MAT-files as generated by `mapreduce` in other environments. For more information, see the `'OutputType'` argument description on the `mapreduce` reference page.

When running `mapreduce` on a Hadoop cluster, the order of the key-value pairs in the output is different compared to running `mapreduce` in other environments. If your application depends on the arrangement of data in the output, you must sort the data according to your own requirements.

Calculate Mean Delay

This example shows how to modify the MATLAB example for calculating mean airline delays to run on a Hadoop cluster.

First, you must set environment variables and cluster properties as appropriate for your specific Hadoop configuration. See your system administrator for the values for these and other properties necessary for submitting jobs to your cluster.

```
setenv('HADOOP_HOME', '/path/to/hadoop/install')
cluster = parallel.cluster.Hadoop;
```

Note The specified `outputFolder` must not already exist. The `mapreduce` output from a Hadoop cluster cannot overwrite an existing folder.

You will lose your data, if `mapreducer` is changed or deleted.

Create a MapReducer object to specify that `mapreduce` should use your Hadoop cluster.

```
mr = mapreducer(cluster);
```

Create and preview the datastore. The data set is available in *matlabroot*/toolbox/matlab/ `demos`.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
     'SelectedVariableNames','ArrDelay','ReadSize',1000);
preview(ds)

    ArrDelay
    _____

      8
      8
     21
     13
      4
     59
      3
     11
```

Next, specify your output folder, output `outds` and call `mapreduce` to execute on the Hadoop cluster specified by `mr`. The map and reduce functions are available in *matlabroot*/toolbox/matlab/demos.

```
outputFolder = 'hdfs:///home/myuser/out1';
outds = mapreduce(ds,@myMapperFcn,@myReducerFcn,'OutputFolder',outputFolder);
meanDelay = mapreduce(ds,@meanArrivalDelayMapper,@meanArrivalDelayReducer,mr,...
           'OutputFolder',outputFolder)

Parallel mapreduce execution on the Hadoop cluster:
********************************
*       MAPREDUCE PROGRESS       *
********************************
Map   0% Reduce   0%
Map  66% Reduce   0%
Map 100% Reduce  66%
Map 100% Reduce 100%

meanDelay =

  KeyValueDatastore with properties:

      Files: {
             ' .../tmp/alafleur/tpc00621b1_4eef_4abc_8078_646aa916e7d9/part0.seq'
             }
   ReadSize: 1 key-value pairs
   FileType: 'seq'
```

Read the result.

```
readall(meanDelay)

        Key             Value
    _____    _____

    'MeanArrivalDelay'    [7.1201]
```

Although for demonstration purposes this example uses a local data set, it is likely when using Hadoop that your data set is stored in an HDFS™ file system. Likewise, you might be required to store the `mapreduce` output in HDFS. For details about accessing HDFS in MATLAB, see "Work with Remote Data".

See Also

Functions
`datastore` | `mapreduce` | `mapreducer` | `parallel.cluster.Hadoop`

Related Examples

- "Getting Started with MapReduce"
- "Run mapreduce on a Parallel Pool" on page 5-55

More About

- "MapReduce"
- "Datastore"

Partition a Datastore in Parallel

Partitioning a datastore in parallel, with a portion of the datastore on each worker in a parallel pool, can provide benefits in many cases:

- Perform some action on only one part of the whole datastore, or on several defined parts simultaneously.
- Search for specific values in the data store, with all workers acting simultaneously on their own partitions.
- Perform a reduction calculation on the workers across all partitions.

This example shows how to use `partition` to parallelize the reading of data from a datastore. It uses a small datastore of airline data provided in MATLAB, and finds the mean of the non-NaN values from its `'ArrDelay'` column.

A simple way to calculate the mean is to divide the sum of all the non-NaN values by the number of non-NaN values. The following code does this for the datastore first in a non-parallel way. To begin, you define a function to amass the count and sum. If you want to run this example, copy and save this function in a folder on the MATLAB command search path.

```matlab
function [total,count] = sumAndCountArrivalDelay(ds)
    total = 0;
    count = 0;
    while hasdata(ds)
        data = read(ds);
        total = total + sum(data.ArrDelay,1,'OmitNaN');
        count = count + sum(~isnan(data.ArrDelay));
    end
end
```

The following code creates a datastore, calls the function, and calculates the mean without any parallel execution. The `tic` and `toc` functions are used to time the execution, here and in the later parallel cases.

```matlab
ds = datastore(repmat({'airlinesmall.csv'},20,1),'TreatAsMissing','NA');
ds.SelectedVariableNames = 'ArrDelay';

reset(ds);
tic
  [total,count] = sumAndCountArrivalDelay(ds)
sumtime = toc
mean = total/count

total =

    17211680


count =

     2417320


sumtime =

    7.7905
```

```
mean =

    7.1201
```

The `partition` function allows you to partition the datastore into smaller parts, each represented as a datastore itself. These smaller datastores work completely independently of each other, so that you can work with them inside of parallel language features such as `parfor` loops and `spmd` blocks.

The number of partitions in the following code is set by the `numpartitions` function, based on the datastore itself (`ds`) and the parallel pool (`gcp`) size. This does not necessarily equal the number of workers in the pool. In this case, the number of loop iterations is then set to the number of partitions (`N`).

The following code starts a parallel pool on a local cluster, then partitions the datastore among workers for iterating over the loop. Again, a separate function is called, which includes the `parfor` loop to amass the count and sum totals. Copy and save this function if you want to run the example.

```matlab
function [total, count] = parforSumAndCountArrivalDelay(ds)
    N = numpartitions(ds,gcp);
    total = 0;
    count = 0;
    parfor ii = 1:N
        % Get partition ii of the datastore.
        subds = partition(ds,N,ii);

        [localTotal,localCount] = sumAndCountArrivalDelay(subds);
        total = total + localTotal;
        count = count + localCount;
    end
end
```

Now the MATLAB code calls this new function, so that the counting and summing of the non-NAN values can occur in parallel loop iterations.

```matlab
p = parpool('local',4);

reset(ds);
tic
[total,count] = parforSumAndCountArrivalDelay(ds)
parfortime = toc
mean = total/count

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).

total =

    17211680


count =

    2417320
```

```
parfortime =

    6.4133


mean =

    7.1201
```

Rather than let the software calculate the number of partitions, you can explicitly set this value, so that the data can be appropriately partitioned to fit your algorithm. For example, to parallelize data from within an `spmd` block, you can specify the number of workers (`numlabs`) as the number of partitions to use. The following function uses an `spmd` block to perform a parallel read, and explicitly sets the number of partitions equal to the number of workers. To run this example, copy and save the function.

```matlab
function [total,count] = spmdSumAndCountArrivalDelay(ds)
    spmd
        subds = partition(ds,numlabs,labindex);
        [total,count] = sumAndCountArrivalDelay(subds);
    end
    total = sum([total{:}]);
    count = sum([count{:}]);
end
```

Now the MATLAB code calls the function that uses an `spmd` block.

```matlab
reset(ds);
tic
[total,count] = spmdSumAndCountArrivalDelay(ds)
spmdtime = toc
mean = total/count
```

```
total =

    17211680


count =

     2417320


spmdtime =

    4.6729


mean =

    7.1201
```

```matlab
delete(p);
```

```
Parallel pool using the 'local' profile is shutting down.
```

You might get some idea of modest performance improvements by comparing the times recorded in the variables `sumtime`, `parfortime`, and `spmdtime`. Your results might vary, as the performance can be affected by the datastore size, parallel pool size, hardware configuration, and other factors.

Program Independent Jobs

Program Independent Jobs

The tasks in an independent job do not directly communicate with each other and are independent. The tasks do not need to run simultaneously, and a worker can run several tasks of the same job in succession. Typically, all tasks perform the same or similar functions on different data sets in an *embarrassingly parallel* configuration.

Some of the details of a job and its tasks can depend on the type of scheduler you are using:

- "Program Independent Jobs on a Local Cluster" on page 6-3
- "Program Independent Jobs for a Supported Scheduler" on page 6-7
- "Plugin Scripts for Generic Schedulers" on page 6-17

Program Independent Jobs on a Local Cluster

In this section...
"Create and Run Jobs with a Local Cluster" on page 6-3
"Local Cluster Behavior" on page 6-5

Create and Run Jobs with a Local Cluster

Some jobs require more control than the functionality offered by high-level constructs like `spmd` and `parfor`. In such cases, you have to program all the steps for creating and running the job. Using the local cluster (or local scheduler) on your machine lets you create and test your jobs without using the resources of your network cluster. Distributing tasks to workers that are all running on your client machine do not offer any performance enhancement. Therefore this feature is provided primarily for code development, testing, and debugging.

Note Workers running in a local cluster on a Microsoft Windows operating system can display Simulink graphics and the output from certain functions such as `uigetfile` and `uigetdir`. (With other platforms or schedulers, workers cannot display any graphical output.) This behavior is subject to removal in a future release.

This section details the steps of a typical programming session with Parallel Computing Toolbox software using a local cluster:

- "Create a Cluster Object" on page 6-3
- "Create a Job" on page 6-3
- "Create Tasks" on page 6-4
- "Submit a Job to the Cluster" on page 6-5
- "Fetch the Job Results" on page 6-5

The objects used by the client session to interact with the cluster are only references to data in the cluster job storage location, not in the client session. After jobs and tasks are created, you can close your client session and restart it, and your job still resides in the storage location. You can find existing jobs using the `findJob` function or the `Jobs` property of the cluster object.

Create a Cluster Object

You use the `parcluster` function to create an object in your local MATLAB session representing the local scheduler.

```
parallel.defaultClusterProfile('local');
c = parcluster();
```

Create a Job

You create a job with the `createJob` function. This statement creates a job in the cluster job storage location and creates the job object `job1` in the client session. If you omit the semicolon at the end of the command, it displays some information about the job.

```
job1 = createJob(c)
```

```
Job

   Properties:

                 ID: 2
               Type: Independent
           Username: eng864
              State: pending
         SubmitTime:
          StartTime:
   Running Duration: 0 days 0h 0m 0s

      AutoAttachFiles: true
 Auto Attached Files: List files
        AttachedFiles: {}
       AdditionalPaths: {}

   Associated Tasks:

       Number Pending: 0
       Number Running: 0
      Number Finished: 0
     Task ID of Errors: []
```

The State property of the job is pending. This means that the job has not yet been submitted (queued) for running, so you can now add tasks to it.

The scheduler display now indicates the existence of your job, which is the pending one, as appears in this partial listing:

```
c

 Local Cluster

   Associated Jobs

              Number Pending: 1
               Number Queued: 0
              Number Running: 0
             Number Finished: 0
```

Create Tasks

After you have created your job, you can create tasks for the job using the createTask function. Tasks define the functions to be evaluated by the workers during the running of the job. Often, the tasks of a job are all identical. In this example, five tasks each generate a 3-by-3 matrix of random numbers.

```
createTask(job1, @rand, 1, {{3,3} {3,3} {3,3} {3,3} {3,3}});
```

The Tasks property of job1 is now a 5-by-1 matrix of task objects.

```
job1.Tasks

        ID     State    FinishTime  Function  Error
   -------------------------------------------------
    1    1    pending                @rand
    2    2    pending                @rand
```

```
   3     3    pending                    @rand
   4     4    pending                    @rand
   5     5    pending                    @rand
```

Submit a Job to the Cluster

To run your job and have its tasks evaluated, you submit the job to the cluster with the `submit` function.

```
submit(job1)
```

The local scheduler starts the workers on your machine, and distributes the tasks of `job1` to these workers for evaluation.

Fetch the Job Results

The results of each task evaluation are stored in the task object `OutputArguments` property as a cell array. After waiting for the job to complete, use the function `fetchOutputs` to retrieve the results from all the tasks in the job.

```
wait(job1)
results = fetchOutputs(job1);
```

Display the results from each task.

```
results{1:5}
```

```
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

    0.4447    0.9218    0.4057
    0.6154    0.7382    0.9355
    0.7919    0.1763    0.9169

    0.4103    0.3529    0.1389
    0.8936    0.8132    0.2028
    0.0579    0.0099    0.1987

    0.6038    0.0153    0.9318
    0.2722    0.7468    0.4660
    0.1988    0.4451    0.4186

    0.8462    0.6721    0.6813
    0.5252    0.8381    0.3795
    0.2026    0.0196    0.8318
```

After the job is complete, you can repeat the commands to examine the updated status of the cluster, job, and task objects:

```
c
job1
job1.Tasks
```

Local Cluster Behavior

The local scheduler runs in the MATLAB client session, so you do not have to start any separate scheduler or MATLAB Job Scheduler process for the local scheduler. When you submit a job to the

local cluster, the scheduler starts a MATLAB worker for each task in the job. You can do this for as many workers as allowed by the local profile. If your job has more tasks than allowed workers, the scheduler waits for one of the current tasks to complete before starting another MATLAB worker to evaluate the next task. You can modify the number of allowed workers in the `local` cluster profile. If not specified, the default is to run only as many workers as computational cores on the machine.

The local cluster has no interaction with any other scheduler or MATLAB Job Scheduler, nor with any other workers that can also be running on your client machine under the mjs service. Multiple MATLAB sessions on your computer can each start its own local scheduler with its own workers, but these groups do not interact with each other.

When you end your MATLAB client session, its local scheduler and any workers that happen to be running also stop immediately.

Program Independent Jobs for a Supported Scheduler

In this section...
"Create and Run Jobs" on page 6-7
"Manage Objects in the Scheduler" on page 6-11

Create and Run Jobs

This section details the steps of a typical programming session with Parallel Computing Toolbox software using a supported job scheduler on a cluster. Supported schedulers include the MATLAB Job Scheduler, Platform LSF (Load Sharing Facility), Microsoft Windows HPC Server (including CCS), PBS Pro, or a TORQUE scheduler.

This section assumes that you have a MATLAB Job Scheduler, LSF, PBS Pro, TORQUE, or Windows HPC Server (including CCS and HPC Server 2008) scheduler installed and running on your network. With all of these cluster types, the basic job programming sequence is the same:

- "Define and Select a Profile" on page 6-7
- "Find a Cluster" on page 6-8
- "Create a Job" on page 6-8
- "Create Tasks" on page 6-9
- "Submit a Job to the Job Queue" on page 6-10
- "Retrieve Job Results" on page 6-10

Note that the objects that the client session uses to interact with the MATLAB Job Scheduler are only references to data that is actually contained in the MATLAB Job Scheduler, not in the client session. After jobs and tasks are created, you can close your client session and restart it, and your job is still stored in the MATLAB Job Scheduler. You can find existing jobs using the `findJob` function or the `Jobs` property of the MATLAB Job Scheduler cluster object.

Define and Select a Profile

A cluster profile identifies the type of cluster to use and its specific properties. In a profile, you define how many workers a job can access, where the job data is stored, where MATLAB is accessed and many other cluster properties. The exact properties are determined by the type of cluster.

The step in this section all assume the profile with the name MyProfile identifies the cluster you want to use, with all necessary property settings. With the proper use of a profile, the rest of the programming is the same, regardless of cluster type. After you define or import your profile, you can set it as the default profile in the Profile Manager GUI, or with the command:

```
parallel.defaultClusterProfile('MyProfile')
```

A few notes regarding different cluster types and their properties:

Notes In a shared file system, all nodes require access to the folder specified in the cluster object's `JobStorageLocation` property.

Because Windows HPC Server requires a shared file system, all nodes require access to the folder specified in the cluster object's `JobStorageLocation` property.

In a shared file system, MATLAB clients on many computers can access the same job data on the network. Properties of a particular job or task should be set from only one client computer at a time.

When you use an LSF scheduler in a nonshared file system, the scheduler might report that a job is in the finished state even though the LSF scheduler might not yet have completed transferring the job's files.

Find a Cluster

You use the `parcluster` function to identify a cluster and to create an object representing the cluster in your local MATLAB session.

To find a specific cluster, user the cluster profile to match the properties of the cluster you want to use. In this example, `MyProfile` is the name of the profile that defines the specific cluster.

```
c = parcluster('MyProfile');

 MJS Cluster

   Properties

                        Name: my_mjs
                     Profile: MyProfile
                    Modified: false
                        Host: node345
                    Username: mylogin

                  NumWorkers: 1
              NumBusyWorkers: 0
              NumIdleWorkers: 1

          JobStorageLocation: Database on node345
           ClusterMatlabRoot: C:\apps\matlab
             OperatingSystem: windows
            AllHostAddresses: 0:0:0:0
                SecurityLevel: 0 (No security)
       HasSecureCommunication: false

   Associated Jobs

              Number Pending: 0
               Number Queued: 0
              Number Running: 0
             Number Finished: 0
```

Create a Job

You create a job with the `createJob` function. Although this command executes in the client session, it actually creates the job on the cluster, `c`, and creates a job object, `job1`, in the client session.

```
job1 = createJob(c)

 Job

    Properties:
                  ID: 1
                Type: Independent
            Username: mylogin
               State: pending
          SubmitTime:
           StartTime:
    Running Duration: 0 days 0h 0m 0s
```

```
       AutoAttachFiles: true
  Auto Attached Files: List files
        AttachedFiles: {}
      AdditionalPaths: {}

   Associated Tasks:

       Number Pending: 0
       Number Running: 0
      Number Finished: 0
    Task ID of Errors: []
```

Note that the job's `State` property is `pending`. This means the job has not been queued for running yet, so you can now add tasks to it.

The cluster's display now includes one pending job, as shown in this partial listing:

```
c

Associated Jobs

                  Number Pending: 1
                   Number Queued: 0
                  Number Running: 0
                 Number Finished: 0
```

You can transfer files to the worker by using the `AttachedFiles` property of the job object. For details, see "Share Code with the Workers" on page 6-13.

Create Tasks

After you have created your job, you can create tasks for the job using the `createTask` function. Tasks define the functions to be evaluated by the workers during the running of the job. Often, the tasks of a job are all identical. In this example, each task will generate a 3-by-3 matrix of random numbers.

```
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
```

The `Tasks` property of `job1` is now a 5-by-1 matrix of task objects.

```
job1.Tasks
```

	ID	State	FinishTime	Function	Error
1	1	pending		@rand	
2	2	pending		@rand	
3	3	pending		@rand	
4	4	pending		@rand	
5	5	pending		@rand	

Alternatively, you can create the five tasks with one call to `createTask` by providing a cell array of five cell arrays defining the input arguments to each task.

```
T = createTask(job1, @rand, 1, {{3,3} {3,3} {3,3} {3,3} {3,3}});
```

In this case, T is a 5-by-1 matrix of task objects.

Submit a Job to the Job Queue

To run your job and have its tasks evaluated, you submit the job to the job queue with the `submit` function.

```
submit(job1)
```

The job manager distributes the tasks of `job1` to its registered workers for evaluation.

Each worker performs the following steps for task evaluation:

1 Receive `AttachedFiles` and `AdditionalPaths` from the job. Place files and modify the path accordingly.

2 Run the `jobStartup` function the first time evaluating a task for this job. You can specify this function in `AttachedFiles` or `AdditionalPaths`. When using a MATLAB Job Scheduler, if the same worker evaluates subsequent tasks for this job, `jobStartup` does not run between tasks.

3 Run the `taskStartup` function. You can specify this function in `AttachedFiles` or `AdditionalPaths`. This runs before every task evaluation that the worker performs, so it could occur multiple times on a worker for each job.

4 If the worker is part of forming a new parallel pool, run the `poolStartup` function. (This occurs when executing `parpool` or when running other types of jobs that form and use a parallel pool, such as `batch`.)

5 Receive the task function and arguments for evaluation.

6 Evaluate the task function, placing the result in the task's `OutputArguments` property. Any error information goes in the task's `Error` property.

7 Run the `taskFinish` function.

Retrieve Job Results

The results of each task's evaluation are stored in that task object's `OutputArguments` property as a cell array. Use the function `fetchOutputs` to retrieve the results from all the tasks in the job.

```
wait(job1)
results = fetchOutputs(job1);
```

Display the results from each task.

```
results{1:5}
```

```
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

    0.4447    0.9218    0.4057
    0.6154    0.7382    0.9355
    0.7919    0.1763    0.9169

    0.4103    0.3529    0.1389
    0.8936    0.8132    0.2028
    0.0579    0.0099    0.1987

    0.6038    0.0153    0.9318
```

```
0.2722    0.7468    0.4660
0.1988    0.4451    0.4186

0.8462    0.6721    0.6813
0.5252    0.8381    0.3795
0.2026    0.0196    0.8318
```

Manage Objects in the Scheduler

Because all the data of jobs and tasks resides in the cluster job storage location, these objects continue to exist even if the client session that created them has ended. The following sections describe how to access these objects and how to permanently remove them:

- "What Happens When the Client Session Ends" on page 6-11
- "Recover Objects" on page 6-11
- "Reset Callback Properties (MATLAB Job Scheduler Only)" on page 6-12
- "Remove Objects Permanently" on page 6-12

What Happens When the Client Session Ends

When you close the client session of Parallel Computing Toolbox software, all of the objects in the workspace are cleared. However, the objects in MATLAB Parallel Server software or other cluster resources remain in place. When the client session ends, only the local reference objects are lost, not the actual job and task data in the cluster.

Therefore, if you have submitted your job to the cluster job queue for execution, you can quit your client session of MATLAB, and the job will be executed by the cluster. You can retrieve the job results later in another client session.

Recover Objects

A client session of Parallel Computing Toolbox software can access any of the objects in MATLAB Parallel Server software, whether the current client session or another client session created these objects.

You create cluster objects in the client session by using the `parcluster` function.

```
c = parcluster('MyProfile');
```

When you have access to the cluster by the object `c`, you can create objects that reference all those job contained in that cluster. The jobs are accessible in cluster object's `Jobs` property, which is an array of job objects:

```
all_jobs = c.Jobs
```

You can index through the array `all_jobs` to locate a specific job.

Alternatively, you can use the `findJob` function to search in a cluster for any jobs or a particular job identified by any of its properties, such as its `State`.

```
all_jobs = findJob(c);
finished_jobs = findJob(c,'State','finished')
```

This command returns an array of job objects that reference all finished jobs on the cluster `c`.

Reset Callback Properties (MATLAB Job Scheduler Only)

When restarting a client session, you lose the settings of any callback properties (for example, the `FinishedFcn` property) on jobs or tasks. These properties are commonly used to get notifications in the client session of state changes in their objects. When you create objects in a new client session that reference existing jobs or tasks, you must reset these callback properties if you intend to use them.

Remove Objects Permanently

Jobs in the cluster continue to exist even after they are finished, and after the MATLAB Job Scheduler is stopped and restarted. The ways to permanently remove jobs from the cluster are explained in the following sections:

- "Delete Selected Objects" on page 6-12
- "Start a MATLAB Job Scheduler from a Clean State" on page 6-12

Delete Selected Objects

From the command line in the MATLAB client session, you can call the `delete` function for any job or task object. If you delete a job, you also remove all tasks contained in that job.

For example, find and delete all finished jobs in your cluster that belong to the user `joep`.

```
c = parcluster('MyProfile')
finished_jobs = findJob(c,'State','finished','Username','joep')
delete(finished_jobs)
clear finished_jobs
```

The `delete` function permanently removes these jobs from the cluster. The `clear` function removes the object references from the local MATLAB workspace.

Start a MATLAB Job Scheduler from a Clean State

When a MATLAB Job Scheduler starts, by default it starts so that it resumes its former session with all jobs intact. Alternatively, a MATLAB Job Scheduler can start from a clean state with all its former history deleted. Starting from a clean state permanently removes all job and task data from the MATLAB Job Scheduler of the specified name on a particular host.

As a network administration feature, the `-clean` flag of the `startjobmanager` script is described in "Start in a Clean State" (MATLAB Parallel Server) in the MATLAB Parallel Server System Administrator's Guide.

Share Code with the Workers

Because the tasks of a job are evaluated on different machines, each machine must have access to all the files needed to evaluate its tasks. The basic mechanisms for sharing code are explained in the following sections:

In this section...
"Workers Access Files Directly" on page 6-13
"Pass Data to and from Worker Sessions" on page 6-14
"Pass MATLAB Code for Startup and Finish" on page 6-15

Note For an example that shows how to share code with workers using `batch`, see "Run Batch Job and Access Files from Workers" on page 11-26.

Workers Access Files Directly

If the workers all have access to the same drives on the network, they can access the necessary files that reside on these shared resources. This is the preferred method for sharing data, as it minimizes network traffic.

You must define each worker session's search path so that it looks for files in the right places. You can define the path:

- By using the job's `AdditionalPaths` property. This is the preferred method for setting the path, because it is specific to the job.

  `AdditionalPaths` identifies folders to be added to the top of the command search path of worker sessions for this job. If you also specify `AttachedFiles`, the `AttachedFiles` are above `AdditionalPaths` on the workers' path.

  When you specify `AdditionalPaths` at the time of creating a job, the settings are combined with those specified in the applicable cluster profile. Setting `AdditionalPaths` on a job object after it is created does not combine the new setting with the profile settings, but overwrites existing settings for that job.

  `AdditionalPaths` is empty by default. For a mixed-platform environment, the character vectors can specify both UNIX and Microsoft Windows style paths; those setting that are not appropriate or not found for a particular machine generate warnings and are ignored.

  This example sets the MATLAB worker path in a mixed-platform environment to use functions in both the central repository `/central/funcs` and the department archive `/dept1/funcs`, which each also have a Windows UNC path.

  ```
  c = parcluster(); % Use default
  job1 = createJob(c);
  ap = {'/central/funcs','/dept1/funcs', ...
      '\\OurDomain\central\funcs','\\OurDomain\dept1\funcs'};
  job1.AdditionalPaths = ap;
  ```

- By putting the `path` command in any of the appropriate startup files for the worker:

- *matlabroot*\toolbox\local\startup.m
- *matlabroot*\toolbox\parallel\user\jobStartup.m
- *matlabroot*\toolbox\parallel\user\taskStartup.m

Access to these files can be passed to the worker by the job's `AttachedFiles` or `AdditionalPaths` property. Otherwise, the version of each of these files that is used is the one highest on the worker's path.

Access to files among shared resources can depend upon permissions based on the user name. You can set the user name with which the MATLAB Job Scheduler and worker services of MATLAB Parallel Server software run by setting the `MJSUSER` value in the `mjs_def` file before starting the services. For Microsoft Windows operating systems, there is also `MJSPASS` for providing the account password for the specified user. For an explanation of service default settings and the `mjs_def` file, see "Define Script Defaults" (MATLAB Parallel Server) in the MATLAB Parallel Server System Administrator's Guide.

Pass Data to and from Worker Sessions

A number of properties on task and job objects are designed for passing code or data from client to scheduler to worker, and back. This information could include MATLAB code necessary for task evaluation, or the input data for processing or output data resulting from task evaluation. The following properties facilitate this communication:

- `InputArguments` — This property of each task contains the input data you specified when creating the task. This data gets passed into the function when the worker performs its evaluation.

- `OutputArguments` — This property of each task contains the results of the function's evaluation.

- `JobData` — This property of the job object contains data that gets sent to every worker that evaluates tasks for that job. This property works efficiently because the data is passed to a worker only once per job, saving time if that worker is evaluating more than one task for the job. (Note: Do not confuse this property with the `UserData` property on any objects in the MATLAB client. Information in `UserData` is available only in the client, and is not available to the scheduler or workers.)

- `AttachedFiles` — This property of the job object is a cell array in which you manually specify all the folders and files that get sent to the workers. On the worker, the files are installed and the entries specified in the property are added to the search path of the worker session.

  `AttachedFiles` contains a list of folders and files that the worker need to access for evaluating a job's tasks. The value of the property (empty by default) is defined in the cluster profile or in the client session. You set the value for the property as a cell array of character vectors. Each character vector is an absolute or relative pathname to a folder or file. (Note: If these files or folders change while they are being transferred, or if any of the folders are empty, a failure or error can result. If you specify a pathname that does not exist, an error is generated.)

  The first time a worker evaluates a task for a particular job, the scheduler passes to the worker the files and folders in the `AttachedFiles` property. On the worker machine, a folder structure is created that is exactly the same as that accessed on the client machine where the property was set. Those entries listed in the property value are added to the top of the command search path in the worker session. (Subfolders of the entries are not added to the path, even though they are included in the folder structure.) To find out where the files are placed on the worker machine, use the function `getAttachedFilesFolder` in code that runs on the worker.

When the worker runs subsequent tasks for the same job, it uses the folder structure already set up by the job's `AttachedFiles` property for the first task it ran for that job.

When you specify `AttachedFiles` at the time of creating a job, the settings are combined with those specified in the applicable profile. Setting `AttachedFiles` on a job object after it is created does not combine the new setting with the profile settings, but overwrites the existing settings for that job.

The transfer of `AttachedFiles` occurs for each worker running a task for that particular job on a machine, regardless of how many workers run on that machine. Normally, the attached files are deleted from the worker machine when the job is completed, or when the next job begins.

- `AutoAttachFiles` — This property of the job object uses a logical value to specify that you want MATLAB to perform an analysis on the task functions in the job and on manually attached files to determine which code files are necessary for the workers, and to automatically send those files to the workers. You can set this property value in a cluster profile using the Profile Manager, or you can set it programmatically on a job object at the command line.

  ```
  c = parcluster();
  j = createJob(c);
  j.AutoAttachFiles = true;
  ```

  The supported code file formats for automatic attachment are MATLAB files (`.m` extension), P-code files (`.p`), and MEX-files (`.mex`). Note that `AutoAttachFiles` does not include data files for your job; use the `AttachedFiles` property to explicitly transfer these files to the workers.

  Use `listAutoAttachedFiles` to get a listing of the code files that are automatically attached to a job.

  If the `AutoAttachFiles` setting is `true` for the cluster profile used when starting a parallel pool, MATLAB performs an analysis on `spmd` blocks, `parfor`-loops, and other attached files to determine what other code files are necessary for execution, then automatically attaches those files to the parallel pool so that the code is available to the workers.

Note There is a default maximum amount of data that can be sent in a single call for setting properties. This limit applies to the `OutputArguments` property as well as to data passed into a job as input arguments or `AttachedFiles`. If the limit is exceeded, you get an error message. For more information about this data transfer size limit, see "Attached Files Size Limitations" on page 5-42.

Pass MATLAB Code for Startup and Finish

As a session of MATLAB, a worker session executes its `startup.m` file each time it starts. You can place the `startup.m` file in any folder on the worker's MATLAB search path, such as `toolbox/parallel/user`.

These additional files can initialize and clean up a worker session as it begins or completes evaluations of tasks for a job:

- `jobStartup.m` automatically executes on a worker when the worker runs its first task of a job.
- `taskStartup.m` automatically executes on a worker each time the worker begins evaluation of a task.
- `poolStartup.m` automatically executes on a worker each time the worker is included in a newly started parallel pool.

- `taskFinish.m` automatically executes on a worker each time the worker completes evaluation of a task.

Empty versions of these files are provided in the folder:

matlabroot`/toolbox/parallel/user`

You can edit these files to include whatever MATLAB code you want the worker to execute at the indicated times.

Alternatively, you can create your own versions of these files and pass them to the job as part of the `AttachedFiles` property, or include the path names to their locations in the `AdditionalPaths` property.

The worker gives precedence to the versions provided in the `AttachedFiles` property, then to those pointed to in the `AdditionalPaths` property. If any of these files is not included in these properties, the worker uses the version of the file in the `toolbox/parallel/user` folder of the worker's MATLAB installation.

See Also

Related Examples
- "Run Batch Job and Access Files from Workers" on page 9-34

Plugin Scripts for Generic Schedulers

In this section...
"Sample Plugin Scripts" on page 6-17
"Writing Custom Plugin Scripts" on page 6-19
"Adding User Customization" on page 6-24
"Managing Jobs with Generic Scheduler" on page 6-25
"Submitting from a Remote Host" on page 6-26
"Submitting without a Shared File System" on page 6-27

The generic scheduler interface provides complete flexibility to configure the interaction of the MATLAB client, MATLAB workers, and a third-party scheduler. The plugin scripts define how MATLAB interacts with your setup.

The following table lists the supported plugin script functions and the stage at which they are evaluated:

File Name	Stage
`independentSubmitFcn.m`	Submitting an independent job
`communicatingSubmitFcn.m`	Submitting a communicating job
`getJobStateFcn.m`	Querying the state of a job
`canceJobFcn.m`	Canceling a job
`cancelTaskFcn.m`	Canceling a task
`deleteJobFcn.m`	Deleting a job
`deleteTaskFcn.m`	Deleting a task
`postConstructFcn.m`	After creating a `parallel.cluster.Generic` instance

These plugin scripts are evaluated only if they have the expected file name and are located in the folder specified by the **PluginScriptsLocation** property of the cluster. For more information about how to configure a generic cluster profile, see "Configure Using the Generic Scheduler Interface" (MATLAB Parallel Server).

Note The `independentSubmitFcn.m` must exist to submit an independent job, and the `communicatingSubmitFcn.m` must exist to submit a communicating job.

Sample Plugin Scripts

To support usage of the generic scheduler interface, plugin scripts are available for the following third-party schedulers:

- LSF
- Grid Engine family
- PBS family

- SLURM

Each installer provides scripts for three possible submission modes:

- Shared – The client can submit directly to the scheduler, and the client and the cluster nodes (or machines) have a shared file system.
- Remote – The client and cluster nodes have a shared file system, but the client machine cannot submit directly to the scheduler, such as when the client utilities of the scheduler are not installed. This mode uses the `ssh` protocol to submit commands to the scheduler using a remote host.
- Nonshared – The client and cluster nodes do not have a shared file system. This mode uses the `ssh` protocol to submit commands to the scheduler using a remote host, and it uses the `sftp` protocol to copy job and task files to the cluster file system.

Each submission mode has its own subfolder within the installation folder. This subfolder contains a `README` file that provides specific instructions on how to use the scripts. Before using the scripts, decide which submission mode describes your network setup.

To run the installer, download the appropriate support package for your scheduler, and open it in your MATLAB client. The installer includes a wizard to guide you through creating a cluster profile for your cluster configuration.

If you want to customize the behavior of the plugin scripts, you can set additional properties, such as `AdditionalSubmitArgs`. For more information, see "Customize Behavior of Sample Plugin Scripts" (MATLAB Parallel Server).

If your scheduler or cluster configuration is not supported by one of the support packages, it is recommended that you modify the scripts of one of these packages. For more information on how to write a set of plugin scripts for generic schedulers, see "Writing Custom Plugin Scripts" on page 6-19.

Wrapper Scripts

The sample plugin scripts use *wrapper* scripts to simplify the implementation of `independentSubmitFcn.m` and `communicatingSubmitFcn.m`. These scripts are not required, however, using them is a good practice to make your code more readable. This table describes these scripts:

File name	Description
`independentJobWrapper.sh`	Used in `independentSubmitFcn.m` to embed a call to the MATLAB executable with the appropriate arguments. It uses environment variables for the location of the executable and its arguments. For an example of its use, see "Sample script for a SLURM scheduler" on page 6-20.
`communicatingJobWrapper.sh`	Used in `communicatingSubmitFcn.m` to distribute a communicating job in your cluster. This script implements the steps in "Submit scheduler job to launch MPI process" on page 6-21. For an example of its use, see "Sample script for a SLURM scheduler" on page 6-22.

Writing Custom Plugin Scripts

Note When writing your own plugin scripts, it is a good practice to start by modifying one of the sample plugin scripts that most closely matches your setup (see "Sample Plugin Scripts" on page 6-17).

---

independentSubmitFcn

When you submit an independent job to a generic cluster, the `independentSubmitFcn.m` function executes in the MATLAB client session.

The declaration line of this function must be:

```
function independentSubmitFcn(cluster,job,environmentProperties)
```

Each task in a MATLAB independent job corresponds to a single job on your scheduler. The purpose of this function is to submit N jobs to your third-party scheduler, where N is the number of tasks in the independent job. Each of these jobs must:

1 Set the five environment variables required by the worker MATLAB to identify the individual task to run. For more information, see "Configure the worker environment" on page 6-19.

2 Call the appropriate MATLAB executable to start the MATLAB worker and run the task. For more information, see "Submit scheduler jobs to run MATLAB workers" on page 6-20.

Configure the worker environment

This table identifies the five environment variables and values that must be set on the worker MATLAB to run an individual task:

Environment Variable Name	Environment Variable Value
PARALLEL_SERVER_DECODE_FUNCTION	`'parallel.cluster.generic.independentDecodeFcn'`
PARALLEL_SERVER_STORAGE_CONSTRUCTOR	`environmentProperties.StorageConstructor`
PARALLEL_SERVER_STORAGE_LOCATION	• If you have a shared file system between the client and cluster nodes, use `environmentProperties.StorageLocation`. • If you do not have a shared file system between the client and cluster nodes, select a folder visible to all cluster nodes. For instructions on copying job and task files between client and cluster nodes, see "Submitting without a Shared File System" on page 6-27.
PARALLEL_SERVER_JOB_LOCATION	`environmentProperties.JobLocation`
PARALLEL_SERVER_TASK_LOCATION	`environmentProperties.TaskLocation{n}` for the `nth` task

Many schedulers support copying the client environment as part of the submission command. If so, you can set the previous environment variables in the client, so the scheduler can copy them to the worker environment. If not, you must modify your submission command to forward these variables.

Submit scheduler jobs to run MATLAB workers

Once the five required parameters for a given job and task are defined on a worker, the task is run by calling the MATLAB executable with suitable arguments. The MATLAB executable to call is defined in `environmentProperties.MatlabExecutable`. The arguments to pass are defined in `environmentProperties.MatlabArguments`.

Note If you cannot submit directly to your scheduler from the client machine, see "Submitting from a Remote Host" on page 6-26 for instructions on how to submit using `ssh`.

Sample script for a SLURM scheduler

This script shows a basic submit function for a SLURM scheduler with a shared file system. For a more complete example, see "Sample Plugin Scripts" on page 6-17.

```
function independentSubmitFcn(cluster,job,environmentProperties)
    % Specify the required environment variables.
    setenv('PARALLEL_SERVER_DECODE_FUNCTION', 'parallel.cluster.generic.independentDecodeFcn');
    setenv('PARALLEL_SERVER_STORAGE_CONSTRUCTOR', environmentProperties.StorageConstructor);
    setenv('PARALLEL_SERVER_STORAGE_LOCATION', environmentProperties.StorageLocation);
    setenv('PARALLEL_SERVER_JOB_LOCATION', environmentProperties.JobLocation);

    % Specify the MATLAB executable and arguments to run on the worker.
    % These are used in the independentJobWrapper.sh script.
    setenv('PARALLEL_SERVER_MATLAB_EXE', environmentProperties.MatlabExecutable);
    setenv('PARALLEL_SERVER_MATLAB_ARGS', environmentProperties.MatlabArguments);

    for ii = 1:environmentProperties.NumberOfTasks
        % Specify the environment variable required to identify which task to run.
        setenv('PARALLEL_SERVER_TASK_LOCATION', environmentProperties.TaskLocations{ii});
        % Specify the command to submit the job to the SLURM scheduler.
        % SLURM will automatically copy environment variables to workers.
        commandToRun = 'sbatch --ntasks=1 independentJobWrapper.sh';
        [cmdFailed, cmdOut] = system(commandToRun);
    end
end
```

The previous example submits a simple bash script, `independentJobWrapper.sh`, to the scheduler. The `independentJobWrapper.sh` script embeds the MATLAB executable and arguments using environment variables:

```
#!/bin/sh
# PARALLEL_SERVER_MATLAB_EXE - the MATLAB executable to use
# PARALLEL_SERVER_MATLAB_ARGS - the MATLAB args to use
exec "${PARALLEL_SERVER_MATLAB_EXE}" ${PARALLEL_SERVER_MATLAB_ARGS}
```

communicatingSubmitFcn

When you submit a communicating job to a generic cluster, the `communicatingSubmitFcn.m` function executes in the MATLAB client session.

The declaration line of this function must be:

```
function communicatingSubmitFcn(cluster,job,environmentProperties)
```

The purpose of this function is to submit a single job to your scheduler. This job must:

1 Set the four environment variables required by the MATLAB workers to identify the job to run. For more information, see "Configure the worker environment" on page 6-21.

2 Call MPI to distribute your job to `N` MATLAB workers. `N` corresponds to the maximum value specified in the `NumWorkersRange` property of the MATLAB job. For more information, see "Submit scheduler job to launch MPI process" on page 6-21.

Configure the worker environment

This table identifies the four environment variables and values that must be set on the worker MATLAB to run a task of a communicating job:

Environment Variable Name	Environment Variable Value
`PARALLEL_SERVER_DECODE_FUNCTION`	`'parallel.cluster.generic.communicatingDecodeFcn'`
`PARALLEL_SERVER_STORAGE_CONSTRUCTOR`	`environmentProperties.StorageConstructor`
`PARALLEL_SERVER_STORAGE_LOCATION`	• If you have a shared file system between the client and cluster nodes, use `environmentProperties.StorageLocation`. • If you do not have a shared file system between the client and cluster nodes, select a folder which exists on all cluster nodes. For instructions on copying job and task files between client and cluster nodes, see "Submitting without a Shared File System" on page 6-27.
`PARALLEL_SERVER_JOB_LOCATION`	`environmentProperties.JobLocation`

Many schedulers support copying the client environment as part of the submission command. If so, you can set the previous environment variables in the client, so the scheduler can copy them to the worker environment. If not, you must modify your submission command to forward these variables.

Submit scheduler job to launch MPI process

After you define the four required parameters for a given job, run your job by launching `N` worker MATLAB processes using `mpiexec`. `mpiexec` is software shipped with the Parallel Computing Toolbox that implements the Message Passing Interface (MPI) standard to allow communication between the worker MATLAB processes. For more information about `mpiexec`, see the MPICH home page.

To run your job, you must submit a job to your scheduler, which executes the following steps. Note that `matlabroot` refers to the MATLAB installation location on your worker nodes.

1 Request `N` processes from the scheduler. `N` corresponds to the maximum value specified in the `NumWorkersRange` property of the MATLAB job.

2 Call `mpiexec` to start worker MATLAB processes. The number of worker MATLAB processes to start on each host should match the number of processes allocated by your scheduler. The `mpiexec` executable is located at `matlabroot/bin/mw_mpiexec`.

The `mpiexec` command automatically forwards environment variables to the launched processes. Therefore, ensure the environment variables listed in "Configure the worker environment" on page 6-21 are set before running `mpiexec`.

To learn more about options for `mpiexec`, see Using the Hydra Process Manager.

Note For a complete example of the previous steps, see the `communicatingJobWrapper.sh` script provided with any of the sample plugin scripts in "Sample Plugin Scripts" on page 6-17. Use this script as a starting point if you need to write your own script.

Sample script for a SLURM scheduler

The following script shows a basic submit function for a SLURM scheduler with a shared file system.

The submitted job is contained in a bash script, `communicatingJobWrapper.sh`. This script implements the relevant steps in "Submit scheduler job to launch MPI process" on page 6-21 for a SLURM scheduler. For a more complete example, see "Sample Plugin Scripts" on page 6-17.

```
function communicatingSubmitFcn(cluster,job,environmentProperties)
    % Specify the four required environment variables.
    setenv('PARALLEL_SERVER_DECODE_FUNCTION', 'parallel.cluster.generic.communicatingDecodeFcn');
    setenv('PARALLEL_SERVER_STORAGE_CONSTRUCTOR', environmentProperties.StorageConstructor);
    setenv('PARALLEL_SERVER_STORAGE_LOCATION', environmentProperties.StorageLocation);
    setenv('PARALLEL_SERVER_JOB_LOCATION', environmentProperties.JobLocation);

    % Specify the MATLAB executable and arguments to run on the worker.
    % Specify the location of the MATLAB install on the cluster nodes.
    % These are used in the communicatingJobWrapper.sh script.
    setenv('PARALLEL_SERVER_MATLAB_EXE', environmentProperties.MatlabExecutable);
    setenv('PARALLEL_SERVER_MATLAB_ARGS', environmentProperties.MatlabArguments);
    setenv('PARALLEL_SERVER_CMR', cluster.ClusterMatlabRoot);

    numberOfTasks = environmentProperties.NumberOfTasks;

    % Specify the command to submit a job to the SLURM scheduler which
    % requests as many processes as tasks in the job.
    % SLURM will automatically copy environment variables to workers.
    commandToRun = sprintf('sbatch --ntasks=%d communicatingJobWrapper.sh', numberOfTasks);
    [cmdFailed, cmdOut] = system(commandToRun);
end
```

getJobStateFcn

When you query the state of a job created with a generic cluster, the `getJobStateFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function state = getJobStateFcn(cluster,job,state)
```

When using a third-party scheduler, it is possible that the scheduler can have more up-to-date information about your jobs than what is available to the toolbox from the local job storage location. This situation is especially true if using a nonshared file system, where the remote file system could be slow in propagating large data files back to your local data location.

To retrieve that information from the scheduler, add a function called `getJobStateFcn.m` to the **PluginScriptsLocation** of your cluster.

The state passed into this function is the state derived from the local job storage. The body of this function can then query the scheduler to determine a more accurate state for the job and return it in place of the stored state. The function you write for this purpose must return a valid value for the state of a job object. Allowed values are 'pending', 'queued', 'running', 'finished', or 'failed'.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see "Managing Jobs with Generic Scheduler" on page 6-25.

cancelJobFcn

When you cancel a job created with a generic cluster, the `cancelJobFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function OK = cancelJobFcn(cluster,job)
```

When you cancel a job created using the generic scheduler interface, by default this action affects only the job data in storage. To cancel the corresponding jobs on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `cancelJobFcn.m` to the **PluginScriptsLocation** of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding jobs from the queue. The function must return a logical scalar indicating the success or failure of canceling the jobs on the scheduler.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see "Managing Jobs with Generic Scheduler" on page 6-25.

cancelTaskFcn

When you cancel a task created with a generic cluster, the `cancelTaskFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function OK = cancelTaskFcn(cluster,task)
```

When you cancel a task created using the generic scheduler interface, by default, this affects only the task data in storage. To cancel the corresponding job on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `cancelTaskFcn.m` to the **PluginScriptsLocation** of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding job from the scheduler queue. The function must return a logical scalar indicating the success or failure of canceling the job on the scheduler.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see "Managing Jobs with Generic Scheduler" on page 6-25.

deleteJobFcn

When you delete a job created with a generic cluster, the `deleteJobFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function deleteTaskFcn(cluster,task)
```

When you delete a job created using the generic scheduler interface, by default, this affects only the job data in storage. To remove the corresponding jobs on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `deleteJobFcn.m` to the **PluginScriptsLocation** of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding jobs from the scheduler queue.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see "Managing Jobs with Generic Scheduler" on page 6-25.

deleteTaskFcn

When you delete a task created with a generic cluster, the `deleteTaskFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function deleteTaskFcn(cluster,task)
```

When you delete a task created using the generic scheduler interface, by default, this affects only the task data in storage. To remove the corresponding job on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `deleteTaskFcn.m` to the **PluginScriptsLocation** of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding job from the scheduler queue.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see "Managing Jobs with Generic Scheduler" on page 6-25.

postConstructFcn

After you create an instance of your cluster in MATLAB, the `postConstructFcn.m` function executes in the MATLAB client session. For example, the following line of code creates an instance of your cluster and runs the `postConstructFcn` function associated with the 'myProfile' cluster profile:

```
c = parcluster('myProfile');
```

The declaration line of the `postConstructFcn` function must be:

```
function postConstructFcn(cluster)
```

If you need to perform custom configuration of your cluster before its use, add a function called `postConstructFcn.m` to the `PluginScriptsLocation` of your cluster. The body of this function can contain any extra setup steps you require.

Adding User Customization

If you need to modify the functionality of your plugin scripts at run time, then use the **AdditionalProperties** property of the generic scheduler interface.

As an example, consider the SLURM scheduler. The submit command for SLURM accepts a `--nodelist` argument that allows you to specify the nodes you want to run on. You can change the value of this argument without having to modify your plugin scripts. To add this functionality, include the following code pattern in your `independentSubmitFcn.m` and `communicatingSubmitFcn.m` scripts:

```
% Basic SLURM submit command
submitCommand = 'sbatch';

% Check if property is defined
if isprop(cluster.AdditionalProperties, 'NodeList')
    % Add appropriate argument and value to submit string
```

```
    submitCommand = [submitCommand ' --nodelist=' cluster.AdditionalProperties.NodeList];
end
```

For an example of how to use this coding pattern, see the nonshared submit functions of the scripts in "Sample Plugin Scripts" on page 6-17.

Setting AdditionalProperties from the Cluster Profile Manager

With the modification to your scripts in the previous example, you can add an **AdditionalProperties** entry to your generic cluster profile to specify a list of nodes to use. This provides a method of documenting customization added to your plugin scripts for anyone you share the cluster profile with.

To add the `NodeList` property to your cluster profile:

1 Start the Cluster Profile Manager from the MATLAB desktop by selecting **Parallel** > **Manage Cluster Profiles**.

2 Select the profile for your generic cluster, and click **Edit**.

3 Navigate to the **AdditionalProperties** table, and click **Add**.

4 Enter `NodeList` as the **Name**.

5 Set **String** as the **Type**.

6 Set the **Value** to the list of nodes.

Setting AdditionalProperties from the MATLAB Command Line

With the modification to your scripts in "Adding User Customization" on page 6-24, you can edit the list of nodes from the MATLAB command line by setting the appropriate property of the cluster object before submitting a job:

```
c = parcluster;
c.AdditionalProperties.NodeList = 'gpuNodeName';
j = c.batch('myScript');
```

Display the `AdditionalProperties` object to see all currently defined properties and their values:

```
>> c.AdditionalProperties
ans =
  AdditionalProperties with properties:
                 ClusterHost: 'myClusterHost'
                    NodeList: 'gpuNodeName'
    RemoteJobStorageLocation: '/tmp/jobs'
```

Managing Jobs with Generic Scheduler

The first requirement for job management is to identify the jobs on the scheduler corresponding to a MATLAB job object. When you submit a job to the scheduler, the command that does the submission in your submit function can return some data about the job from the scheduler. This data typically includes a job ID. By storing that scheduler job ID with the MATLAB job object, you can later refer to the scheduler job by this job ID when you send management commands to the scheduler. Similarly, you can store a map of MATLAB task IDs to scheduler job IDs to help manage individual tasks. The toolbox function that stores this cluster data is `setJobClusterData`.

Save Job Scheduler Data

This example shows how to modify the `independentSubmitFcn.m` function to parse the output of each command submitted to a SLURM scheduler. You can use regular expressions to extract the scheduler job ID for each task and then store it using `setJobClusterData`.

```matlab
% Pattern to extract scheduler job ID from SLURM sbatch output
searchPattern = '.*Submitted batch job ([0-9]+).*';

jobIDs = cell(numberOfTasks, 1);
for ii = 1:numberOfTasks
    setenv('PARALLEL_SERVER_TASK_LOCATION', environmentProperties.TaskLocations{ii});
    commandToRun = 'sbatch --ntasks=1 independentJobWrapper.sh';
    [cmdFailed, cmdOut] = system(commandToRun);
    jobIDs{ii} = regexp(cmdOut, searchPattern, 'tokens', 'once');
end

% set the job IDs on the job cluster data
cluster.setJobClusterData(job, struct('ClusterJobIDs', {jobIDs}));
```

Retrieve Job Scheduler Data

This example modifies the `cancelJobFcn.m` to cancel the corresponding jobs on the SLURM scheduler. The example uses `getJobClusterData` to retrieve job scheduler data.

```matlab
function OK = cancelJobFcn(cluster, job)

% Get the scheduler information for this job
data = cluster.getJobClusterData(job);
jobIDs = data.ClusterJobIDs;

for ii = 1:length(jobIDs)
    % Tell the SLURM scheduler to cancel the job
    commandToRun = sprintf('scancel ''%s''', jobIDs{ii});
    [cmdFailed, cmdOut] = system(commandToRun);
end

OK = true;
```

Submitting from a Remote Host

If the MATLAB client is unable to submit directly to your scheduler, use `parallel.cluster.RemoteClusterAccess` to establish a connection and run commands on a remote host.

This object uses the `ssh` protocol, and hence requires an `ssh` daemon service running on the remote host. To establish a connection, you must either provide a user name and password for the remote host, or a valid identity file.

The following code executes a command on a remote host, `remoteHostname`, as the user, `user`.

```matlab
% This will prompt for the password of user
access = parallel.cluster.RemoteClusterAccess.getConnectedAccess('remoteHostname', 'user');
% Execute a command on remoteHostname
[cmdFailed, cmdOut] = access.runCommand(commandToRun);
```

For an example of plugin scripts using remote host submission, see the remote submission mode in "Sample Plugin Scripts" on page 6-17.

Submitting without a Shared File System

If the MATLAB client does not have a shared file system with the cluster nodes, use `parallel.cluster.RemoteClusterAccess` to establish a connection and copy job and task files between the client and cluster nodes.

This object uses the `ssh` protocol, and hence requires an `ssh` daemon service running on the remote host. To establish a connection, you must either provide a user name and password for the remote host or a valid identity file.

When using nonshared submission, you must specify both a local job storage location to use on the client and a remote job storage location to use on the cluster. The remote job storage location must be available to all nodes of the cluster.

`parallel.cluster.RemoteClusterAccess` uses file mirroring to continuously synchronize the local job and task files with those on the cluster. When file mirroring first starts, local job and task files are uploaded to the remote job storage location. As the job executes, the file mirroring continuously checks the remote job storage location for new files and updates, and copies the files to the local storage on the client. This procedure ensures the MATLAB client always has an up-to-date view of the jobs and tasks executing on the scheduler.

This example connects to the remote host, `remoteHostname`, as the user, `user`, and establishes `/remote/storage` as the remote cluster storage location to synchronize with. It then starts file mirroring for a job, copying the local files of the job to `/remote/storage` on the cluster, and then syncing any changes back to the local machine.

```
% This will prompt for the password of user
access = parallel.cluster.RemoteClusterAccess.getConnectedAccessWithMirror('remoteHostname', '/re
% Start file mirroring for a job
access.startMirrorForJob(job);
```

For an example of plugin scripts without a shared file system, see the nonshared submission mode in "Sample Plugin Scripts" on page 6-17.

See Also

Related Examples
• "Configure Using the Generic Scheduler Interface" (MATLAB Parallel Server)

Program Communicating Jobs

Program Communicating Jobs

Communicating jobs are those in which the workers can communicate with each other during the evaluation of their tasks. A communicating job consists of only a single task that runs simultaneously on several workers, usually with different data. More specifically, the task is duplicated on each worker, so each worker can perform the task on a different set of data, or on a particular segment of a large data set. The workers can communicate with each other as each executes its task. The function that the task runs can take advantage of a worker's awareness of how many workers are running the job, which worker this is among those running the job, and the features that allow workers to communicate with each other.

In principle, you create and run communicating jobs similarly to the way you "Program Independent Jobs" on page 6-2:

1   Define and select a cluster profile.
2   Find a cluster.
3   Create a communicating job.
4   Create a task.
5   Submit the job for running. For details about what each worker performs for evaluating a task, see "Submit a Job to the Job Queue" on page 6-10.
6   Retrieve the results.

The differences between independent jobs and communicating jobs are summarized in the following table.

Independent Job	Communicating Job
MATLAB workers perform the tasks but do not communicate with each other.	MATLAB workers can communicate with each other during the running of their tasks.
You define any number of tasks in a job.	You define only one task in a job. Duplicates of that task run on all workers running the communicating job.
Tasks need not run simultaneously. Tasks are distributed to workers as the workers become available, so a worker can perform several of the tasks in a job.	Tasks run simultaneously, so you can run the job only on as many workers as are available at run time. The start of the job might be delayed until the required number of workers is available.

Some of the details of a communicating job and its tasks might depend on the type of scheduler you are using. The following sections discuss different schedulers and explain programming considerations:

•   "Program Communicating Jobs for a Supported Scheduler" on page 7-3
•   "Plugin Scripts for Generic Schedulers" on page 6-17
•   "Further Notes on Communicating Jobs" on page 7-6

Program Communicating Jobs for a Supported Scheduler

In this section...
"Schedulers and Conditions" on page 7-3
"Code the Task Function" on page 7-3
"Code in the Client" on page 7-4

Schedulers and Conditions

You can run a communicating job using any type of scheduler. This section illustrates how to program communicating jobs for supported schedulers (MATLAB Job Scheduler, local scheduler, Microsoft Windows HPC Server (including CCS), Platform LSF, PBS Pro, or TORQUE).

To use this supported interface for communicating jobs, the following conditions must apply:

- You must have a shared file system between client and cluster machines
- You must be able to submit jobs directly to the scheduler from the client machine

Note When using any third-party scheduler for running a communicating job, if all these conditions are not met, you must use the generic scheduler interface. (Communicating jobs also include `parpool`, `spmd`, and `parfor`.) See "Plugin Scripts for Generic Schedulers" on page 6-17.

Code the Task Function

In this section a simple example illustrates the basic principles of programming a communicating job with a third-party scheduler. In this example, the worker whose `labindex` value is 1 creates a magic square comprised of a number of rows and columns that is equal to the number of workers running the job (`numlabs`). In this case, four workers run a communicating job with a 4-by-4 magic square. The first worker broadcasts the matrix with `labBroadcast` to all the other workers, each of which calculates the sum of one column of the matrix. All of these column sums are combined with the `gplus` function to calculate the total sum of the elements of the original magic square.

The function for this example is shown below.

```
function total_sum = colsum
if labindex == 1
    % Send magic square to other workers
    A = labBroadcast(1,magic(numlabs))
else
    % Receive broadcast on other workers
    A = labBroadcast(1)
end

% Calculate sum of column identified by labindex for this worker
column_sum = sum(A(:,labindex))

% Calculate total sum by combining column sum from all workers
total_sum = gplus(column_sum)
```

This function is saved as the file `colsum.m` on the path of the MATLAB client. It will be sent to each worker by the job's `AttachedFiles` property.

While this example has one worker create the magic square and broadcast it to the other workers, there are alternative methods of getting data to the workers. Each worker could create the matrix for itself. Alternatively, each worker could read its part of the data from a file on disk, the data could be passed in as an argument to the task function, or the data could be sent in a file contained in the job's `AttachedFiles` property. The solution to choose depends on your network configuration and the nature of the data.

Code in the Client

As with independent jobs, you choose a profile and create a cluster object in your MATLAB client by using the `parcluster` function. There are slight differences in the profiles, depending on the scheduler you use, but using profiles to define as many properties as possible minimizes coding differences between the scheduler types.

You can create and configure the cluster object with this code:

```
c = parcluster('MyProfile')
```

where `'MyProfile'` is the name of a cluster profile for the type of scheduler you are using. Any required differences for various cluster options are controlled in the profile. You can have one or more separate profiles for each type of scheduler. For complete details, see "Discover Clusters and Use Cluster Profiles" on page 5-11. Create or modify profiles according to the instructions of your system administrator.

When your cluster object is defined, you create the job object with the `createCommunicatingJob` function. The job `Type` property must be set as `'SPMD'` when you create the job.

```
cjob = createCommunicatingJob(c,'Type','SPMD');
```

The function file `colsum.m` (created in "Code the Task Function" on page 7-3) is on the MATLAB client path, but it has to be made available to the workers. One way to do this is with the job's `AttachedFiles` property, which can be set in the profile you used, or by:

```
cjob.AttachedFiles = {'colsum.m'}
```

Here you might also set other properties on the job, for example, setting the number of workers to use. Again, profiles might be useful in your particular situation, especially if most of your jobs require many of the same property settings. To run this example on four workers, you can established this in the profile, or by the following client code:

```
cjob.NumWorkersRange = 4
```

You create the job's one task with the usual `createTask` function. In this example, the task returns only one argument from each worker, and there are no input arguments to the `colsum` function.

```
t = createTask(cjob, @colsum, 1, {})
```

Use `submit` to run the job.

```
submit(cjob)
```

Make the MATLAB client wait for the job to finish before collecting the results. The results consist of one value from each worker. The `gplus` function in the task shares data between the workers, so that each worker has the same result.

```
wait(cjob)
results = fetchOutputs(cjob)
results =
    [136]
    [136]
    [136]
    [136]
```

Further Notes on Communicating Jobs

Number of Tasks in a Communicating Job

Although you create only one task for a communicating job, the system copies this task for each worker that runs the job. For example, if a communicating job runs on four workers, the `Tasks` property of the job contains four task objects. The first task in the job's `Tasks` property corresponds to the task run by the worker whose `labindex` is `1`, and so on, so that the `ID` property for the task object and `labindex` for the worker that ran that task have the same value. Therefore, the sequence of results returned by the `fetchOutputs` function corresponds to the value of `labindex` and to the order of tasks in the job's `Tasks` property.

Avoid Deadlock and Other Dependency Errors

Because code running in one worker for a communicating job can block execution until some corresponding code executes on another worker, the potential for deadlock exists in communicating jobs. This is most likely to occur when transferring data between workers or when making code dependent upon the `labindex` in an `if` statement. Some examples illustrate common pitfalls.

Suppose you have a codistributed array D, and you want to use the `gather` function to assemble the entire array in the workspace of a single worker.

```
if labindex == 1
    assembled = gather(D);
end
```

The reason this fails is because the `gather` function requires communication between all the workers across which the array is distributed. When the `if` statement limits execution to a single worker, the other workers required for execution of the function are not executing the statement. As an alternative, you can use `gather` itself to collect the data into the workspace of a single worker:
`assembled = gather(D, 1)`.

In another example, suppose you want to transfer data from every worker to the next worker on the right (defined as the next higher `labindex`). First you define for each worker what the workers on the left and right are.

```
from_lab_left = mod(labindex - 2, numlabs) + 1;
to_lab_right  = mod(labindex, numlabs) + 1;
```

Then try to pass data around the ring.

```
labSend (outdata, to_lab_right);
indata = labReceive(from_lab_left);
```

The reason this code might fail is because, depending on the size of the data being transferred, the `labSend` function can block execution in a worker until the corresponding receiving worker executes its `labReceive` function. In this case, all the workers are attempting to send at the same time, and none are attempting to receive while `labSend` has them blocked. In other words, none of the workers

get to their `labReceive` statements because they are all blocked at the `labSend` statement. To avoid this particular problem, you can use the `labSendReceive` function.

8

GPU Computing

GPU Capabilities and Performance

In this section...
"Capabilities" on page 8-2
"Performance Benchmarking" on page 8-2

Capabilities

Parallel Computing Toolbox enables you to program MATLAB to use your computer's graphics processing unit (GPU) for matrix operations. In many cases, execution in the GPU is faster than in the CPU, so this feature might offer improved performance.

Toolbox capabilities with the GPU let you:

- "Identify and Select a GPU Device" on page 8-19
- "Create GPU Arrays from Existing Data" on page 8-3
- "Run MATLAB Functions on a GPU" on page 8-9
- "Run CUDA or PTX Code on GPU" on page 8-20
- "Run MEX-Functions Containing CUDA Code" on page 8-28

Performance Benchmarking

You can use `gputimeit` to measure the execution time of functions that run on the GPU. For more details, see "Measure and Improve GPU Performance" on page 8-31.

The MATLAB Central file exchange offers a function called gpuBench, which measures the execution time for various MATLAB GPU tasks and estimates the peak performance of your GPU. See https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.

Establish Arrays on a GPU

In this section...

A `gpuArray` in MATLAB represents an array that is stored on the GPU. For a complete list of functions that support arrays on the GPU, see "Run MATLAB Functions on a GPU" on page 8-9.

Create GPU Arrays from Existing Data

Send Arrays to the GPU

GPU arrays can be created by transferring existing arrays from the workspace to the GPU. Use the `gpuArray` function to transfer an array from MATLAB to the GPU:

```
N = 6;
M = magic(N);
G = gpuArray(M);
```

You can accomplish this in a single line of code:

```
G = gpuArray(magic(N));
```

G is now a MATLAB gpuArray object that represents the magic square stored on the GPU. The input provided to `gpuArray` must be numeric (for example: `single`, `double`, `int8`, etc.) or logical. (See also "Work with Complex Numbers on a GPU" on page 8-17.)

Retrieve Arrays from the GPU

Use the `gather` function to retrieve arrays from the GPU to the MATLAB workspace. This takes an array that is on the GPU represented by a gpuArray object, and transfers it to the MATLAB workspace as a regular MATLAB array. You can use `isequal` to verify that you get the correct values back:

```
G = gpuArray(ones(100,'uint32'));
D = gather(G);
OK = isequal(D,ones(100,'uint32'))
```

Gathering back to the CPU can be costly, and is generally not necessary unless you need to use your result with functions that do not support `gpuArray`.

Example: Transfer Array to the GPU

Create a 1000-by-1000 random matrix in MATLAB, and then transfer it to the GPU:

```
X = rand(1000);
G = gpuArray(X);
```

Example: Transfer Array of a Specified Precision

Create a matrix of double-precision random values in MATLAB, and then transfer the matrix as single-precision from MATLAB to the GPU:

```
X = rand(1000);
G = gpuArray(single(X));
```

Create GPU Arrays Directly

A number of functions allow you to directly construct arrays on the GPU by specifying the `'gpuArray'` type as an input argument. These functions require only array size and data class information, so they can construct an array without having to transfer any elements from the MATLAB workspace. For more information, see `gpuArray`.

Example: Construct an Identity Matrix on the GPU

To create a 1024-by-1024 identity matrix of type `int32` on the GPU, type

```
II = eye(1024,'int32','gpuArray');
size(II)

        1024         1024
```

With one numerical argument, you create a 2-dimensional matrix.

Example: Construct a Multidimensional Array on the GPU

To create a 3-dimensional array of ones with data class `double` on the GPU, type

```
G = ones(100,100,50,'gpuArray');
size(G)

   100    100     50
```

```
underlyingType(G)
```

```
double
```

The default class of the data is `double`, so you do not have to specify it.

Example: Construct a Vector on the GPU

To create a 8192-element column vector of zeros on the GPU, type

```
Z = zeros(8192,1,'gpuArray');
size(Z)

        8192                1
```

For a column vector, the size of the second dimension is 1.

Examine gpuArray Characteristics

There are several functions available for examining the characteristics of a gpuArray object:

Function	Description
underlyingType	Class of the underlying data in the array
existsOnGPU	Indication if array exists on the GPU and is accessible
isreal	Indication if array data is real
isUnderlyingType	Determine if underlying array data is of specified class, such as `double`
isequal	Determine if two or more arrays are equal
isnumeric	Determine if an array is of a numeric data type
issparse	Determine if an array is sparse
length	Length of vector or largest array dimension
mustBeUnderlyingType	Validate that array has specified underlying type, such as double
ndims	Number of dimensions in the array
size	Size of array dimensions

For example, to examine the size of the gpuArray object G, type:

```
G = rand(100,'gpuArray');
s = size(G)

    100   100
```

Save and Load gpuArrays

You can save gpuArray variables as MAT files for later use. When you save a gpuArray from the MATLAB workspace, the data is saved as a gpuArray variable in a MAT file. When you load a MAT file containing a gpuArray variable, the data is loaded onto the GPU as a gpuArray.

Note You can load MAT files containing gpuArray data as in-memory arrays when a GPU is not available. A gpuArray loaded without a GPU is limited and you cannot use it for computations. To use a gpuArray loaded without a GPU, retrieve the contents using `gather`.

For more information about how to save and load variables in the MATLAB workspace, see "Save and Load Workspace Variables".

See Also
gpuArray

More About

Random Number Streams on a GPU

By default, the random number generation functions `rand`, `randi`, and `randn` use different generator settings for calculations on a GPU compared to those on a CPU. You can change the behavior of random number generators to generate reproducible sequences of random numbers on the GPU and CPU.

The table below summarizes the default settings for the GPU and CPU on client and worker MATLAB sessions:

	Generator	Seed	Normal Transform
Client CPU	`'Twister'` or `'mt19937ar'`	0	`'Ziggurat'`
Worker CPU	`'Threefry'` or `'Threefry4x64_20'`	0	`'Inversion'`
GPU (on client or worker)	`'Threefry'` or `'Threefry4x64_20'`	0	`'BoxMuller'`

In most cases, it does not matter that the default random number generator on the GPU is not the same as the default generators on the client or worker CPU. However, if you need to reproduce the same results on both the GPU and CPU, you can set the generators accordingly.

Client CPU and GPU

In a fresh MATLAB session, MATLAB generates different sequences of random numbers on the CPU and GPU.

```
Rc = rand(1,4)

Rc =
    0.8147    0.9058    0.1270    0.9134

Rg = rand(1,4,'gpuArray')

Rg =
    0.3640    0.5421    0.6543    0.7436
```

If you need to generate the same sequence of random numbers on both the GPU and CPU, you can set the generator settings to match.

There are three random number generator algorithms available on the GPU: `'Threefry'`, `'Philox'`, and `'CombRecursive'`. All are supported on the CPU. The following table lists the algorithms for these generators and their properties.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period in Full Precision
`'Threefry'` or `'Threefry4x64_20'`	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})
`'Philox'` or `'Philox4x32_10'`	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
`'CombRecursive'` or `'mrg32k3a'`	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})

You can use `rng` and `gpurng` to set the generator algorithm and seed on the CPU and GPU, respectively.

```
sc = rng(1,'Threefry');
Rc = rand(1,4)

Rc =
    0.1404    0.8197    0.1073    0.4131

sg = gpurng(1,'Threefry');
Rg = rand(1,4,'gpuArray')

Rg =
    0.1404    0.8197    0.1073    0.4131
```

`rand` and `randi` now generate the same sequences of random numbers on the client CPU and GPU.

Worker CPU and GPU

A parallel worker CPU uses the same default random number generator type and seed as the client GPU and the worker GPU, if it has one. The GPU and CPU do not share the same stream. By default, `rand` and `randi` generate the same sequence of numbers on a GPU and a worker CPU.

The settings are different from those on the client CPU. For more information, see "Control Random Number Streams on Workers" on page 5-29

If you need to generate different random numbers on each worker, you can change the generator settings. In this example, each worker creates the same sequence on its GPU and CPU, but different sequences are generated on each worker.

```
p = parpool(2);
spmd
    rng(labindex,'Threefry');
    Rc = rand(1,4)

    gpurng(labindex,'Threefry');
    Rg = rand(1,4,'gpuArray')
end
delete(p)
```

Normally Distributed Random Numbers

For normally distributed random numbers created using the `randn` function, MATLAB produces different results on a client CPU, a worker CPU and a GPU. The transformation of uniform random numbers into normally distributed random numbers is controlled by the `NormalTransform` setting. You can control this on the GPU using `parallel.gpu.RandStream`.

On a client CPU, the default `'NormalTransform'` setting is `'Ziggurat'`. On a worker CPU, the default setting is `'Inversion'`.

Unless otherwise specified, GPU code uses the `'BoxMuller'` transform for the `'Threefry'` and `'Philox'` generators and the `'Inversion'` transform for the `'CombRecursive'` generator.

You can set the same generators and transforms on the CPU and the GPU to get the same `randn` sequences. The only transform supported on both the CPU and GPU is the `'Inversion'` transform.

```
sc = RandStream('Threefry','NormalTransform','Inversion','Seed',1);
RandStream.setGlobalStream(sc)

sg = parallel.gpu.RandStream('Threefry','NormalTransform','Inversion','Seed',1);
parallel.gpu.RandStream.setGlobalStream(sg);

Rc = randn(1,4)

Rc =
    -1.0783    0.9144   -1.2412   -0.2196

Rg = randn(1,4,'gpuArray')

Rg =
    -1.0783    0.9144   -1.2412   -0.2196
```

See Also

RandStream | gpurng | parallel.gpu.RandStream | rng

More About

- "Control Random Number Streams on Workers" on page 5-29
- "Creating and Controlling a Random Number Stream"

Run MATLAB Functions on a GPU

MATLAB Functions with gpuArray Arguments

Hundreds of functions in MATLAB and other toolboxes run automatically on a GPU if you supply a `gpuArray` argument.

```
A = gpuArray([1 0 1; -1 -2 0; 0 1 -1]);
e = eig(A);
```

Whenever you call any of these functions with at least one gpuArray as a data input argument, the function executes on the GPU. The function generates a gpuArray as the result, unless returning MATLAB data is more appropriate (for example, `size`). You can mix inputs using both gpuArray and MATLAB arrays in the same function call. To learn more about when a function runs on GPU or CPU, see "Special Conditions for gpuArray Inputs" on page 8-18. GPU-enabled functions include the discrete Fourier transform (`fft`), matrix multiplication (`mtimes`), left matrix division (`mldivide`), and hundreds of others. For more information, see "Check GPU-Supported Functions" on page 8-9.

Check GPU-Supported Functions

If a MATLAB function has support for gpuArrays, you can consult additional GPU usage information on its function page. See **GPU Arrays** in the **Extended Capabilities** section at the end of the function page.

Tip For a filtered list of MATLAB functions that support GPU arrays, see Function List (GPU-arrays).

Several MATLAB toolboxes include functions with built-in GPU support. To view lists of all functions in these toolboxes that support gpuArrays, use the links in the following table. Functions in the lists with warning indicators have limitations or usage notes specific to running the function on a GPU. You can check the usage notes and limitations in the Extended Capabilities section of the function reference page. For information about updates to individual GPU-enabled functions, see the release notes.

Toolbox name	List of functions with gpuArray support	GPU-specific documentation
MATLAB	Functions with gpuArray support	
Statistics and Machine Learning Toolbox	Functions with gpuArray support	
Image Processing Toolbox™	Functions with gpuArray support	"GPU Computing" (Image Processing Toolbox)
Deep Learning Toolbox™	Functions with gpuArray support *(see also "Deep Learning with GPUs" on page 8-10)	• "Deep Learning with Big Data on GPUs and in Parallel" (Deep Learning Toolbox) • "Deep Learning with MATLAB on Multiple GPUs" (Deep Learning Toolbox)

Toolbox name	List of functions with gpuArray support	GPU-specific documentation
Computer Vision Toolbox™	Functions with gpuArray support	"GPU Code Generation and Acceleration" (Computer Vision Toolbox)
Communications Toolbox™	Functions with gpuArray support	"Code Generation and Acceleration Support" (Communications Toolbox)
Signal Processing Toolbox™	Functions with gpuArray support	"Code Generation and GPU Support" (Signal Processing Toolbox)
Audio Toolbox™	Functions with gpuArray support	"Code Generation and GPU Support" (Audio Toolbox)
Wavelet Toolbox™	Functions with gpuArray support	"Code Generation and GPU Support" (Wavelet Toolbox)
Curve Fitting Toolbox™	Functions with gpuArray support	

You can browse GPU-supported functions from all MathWorks products at the following link: GPU-supported functions. Alternatively, you can filter by product. On the **Help** bar, click **Functions**. In the function list, browse the left pane to select a product, for example, MATLAB. At the bottom of the left pane, select **GPU Arrays**. If you select a product that does not have GPU-enabled functions, then the **GPU Arrays** filter is not available.

Deep Learning with GPUs

For many functions in Deep Learning Toolbox, GPU support is automatic if you have a suitable GPU and Parallel Computing Toolbox. You do not need to convert your data to gpuArray. The following is a non-exhaustive list of functions that, by default, run on the GPU if available.

- `trainNetwork`
- `predict`
- `predictAndUpdateState`
- `classify`
- `classifyAndUpdateState`
- `activations`

For more information about automatic GPU-support in Deep Learning Toolbox, see "Deep Learning with Big Data on GPUs and in Parallel" (Deep Learning Toolbox).

For advanced networks and workflows that use networks defined as `dlnetwork` objects or model functions, convert your data to gpuArray. Use functions with gpuArray support to run custom training loops or prediction on the GPU.

Check or Select a GPU

If you have a GPU, then MATLAB automatically uses it for GPU computations. You can check your GPU using the `gpuDevice` function. If you have multiple GPUs, then you can use `gpuDevice` to select one of them, or use multiple GPUs with a parallel pool. For an example, see "Identify and

Select a GPU" on page 10-32 and "Use Multiple GPUs in a Parallel Pool" on page 10-33. To check if your GPU is supported, see "GPU Support by Release" on page 8-38.

For deep learning, MATLAB provides automatic parallel support for multiple GPUs. See "Deep Learning with MATLAB on Multiple GPUs" (Deep Learning Toolbox).

Use MATLAB Functions with a GPU

This example shows how to use GPU-enabled MATLAB functions to operate with gpuArrays. You can check the properties of your GPU using the `gpuDevice` function.

```
gpuDevice

ans =
  CUDADevice with properties:

                      Name: 'GeForce GTX 1080'
                     Index: 1
         ComputeCapability: '6.1'
            SupportsDouble: 1
             DriverVersion: 10.1000
            ToolkitVersion: 10.1000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
         MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 8.5899e+09
           AvailableMemory: 6.9012e+09
       MultiprocessorCount: 20
              ClockRateKHz: 1733500
               ComputeMode: 'Default'
       GPUOverlapsTransfers: 1
     KernelExecutionTimeout: 1
           CanMapHostMemory: 1
            DeviceSupported: 1
             DeviceSelected: 1
```

Create a row vector that repeats values from -15 to 15. To transfer it to the GPU and create a gpuArray, use the `gpuArray` function.

```
X = [-15:15 0 -15:15 0 -15:15];
gpuX = gpuArray(X);
whos gpuX

  Name      Size            Bytes  Class       Attributes

  gpuX      1x95                4  gpuArray
```

To operate with gpuArrays, use any GPU-enabled MATLAB function. MATLAB automatically runs calculations on the GPU. For more information, see "Run MATLAB Functions on a GPU" on page 8-9. For example, use a combination of `diag`, `expm`, `mod`, `round`, `abs`, and `fliplr`.

```
gpuE = expm(diag(gpuX,-1)) * expm(diag(gpuX,1));
gpuM = mod(round(abs(gpuE)),2);
gpuF = gpuM + fliplr(gpuM);
```

Plot the results.

```
imagesc(gpuF);
colormap(flip(gray));
```

If you need to transfer the data back from the GPU, use `gather`. Gathering back to the CPU can be costly, and is generally not necessary unless you need to use your result with functions that do not support gpuArray.

```
result = gather(gpuF);
whos result
```

```
  Name        Size            Bytes  Class     Attributes

  result      96x96           73728  double
```

In general there can be differences in the results if you run the code on the CPU, due to numerical precision and algorithmic differences between GPU and CPU. Answers on CPU and GPU are both equally valid floating point approximations to the true analytical result, having been subjected to different roundoff during computation. In this example, the results are integers and `round` eliminates the roundoff errors.

Sharpen an Image Using the GPU

This example shows how to sharpen an image using gpuArrays and GPU-enabled functions.

Read the image, and send it to the GPU using the `gpuArray` function.

```
image = gpuArray(imread('peppers.png'));
```

Convert the image to doubles, and apply convolutions to obtain the gradient image. Then, using the gradient image, sharpen the image by a factor of `amount`.

```
dimage = im2double(image);
gradient = convn(dimage,ones(3)./9,'same') - convn(dimage,ones(5)./25,'same');
amount = 5;
sharpened = dimage + amount.*gradient;
```

Resize, plot and compare the original and sharpened images.

```
imshow(imresize([dimage, sharpened],0.7));
title('Original image (left) vs sharpened image (right)');
```


Original image (left) vs sharpened image (right)

Compute the Mandelbrot Set using GPU-Enabled Functions

This example shows how to use GPU-enabled MATLAB functions to compute a well-known mathematical construction: the Mandelbrot set. Check your GPU using the `gpuDevice` function.

Define the parameters. The Mandelbrot algorithm iterates over a grid of real and imaginary parts. The following code defines the number of iterations, grid size, and grid limits.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862,  0.123640851045266];
```

You can use the `gpuArray` function to transfer data to the GPU and create a `gpuArray`, or you can create an array directly on the GPU. `gpuArray` provides GPU versions of many functions that you can use to create data arrays, such as `linspace`. For more information, see "Create GPU Arrays Directly" on page 8-4.

```
x = gpuArray.linspace(xlim(1),xlim(2),gridSize);
y = gpuArray.linspace(ylim(1),ylim(2),gridSize);
whos x y
```

```
  Name       Size                Bytes  Class       Attributes

  x          1x1000                  4  gpuArray
  y          1x1000                  4  gpuArray
```

Many MATLAB functions support gpuArrays. When you supply a gpuArray argument to any GPU-enabled function, the function runs automatically on the GPU. For more information, see "Run MATLAB Functions on a GPU" on page 8-9. Create a complex grid for the algorithm, and create the array `count` for the results. To create this array directly on the GPU, use the `ones` function, and specify `'gpuArray'`.

```
[xGrid,yGrid] = meshgrid(x,y);
z0 = complex(xGrid,yGrid);
count = ones(size(z0),'gpuArray');
```

The following code implements the Mandelbrot algorithm using GPU-enabled functions. Because the code uses gpuArrays, the calculations happen on the GPU.

```
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end
count = log(count);
```

When computations are done, plot the results.

```
imagesc(x,y,count)
colormap([jet();flipud(jet());0 0 0]);
axis off
```

Work with Sparse Arrays on a GPU

The following functions support sparse gpuArrays.

abs	end	nonzeros
acos	eps	norm
acosd	exp	numel
acosh	expint	nzmax
acot	expm1	pcg
acotd	find	plus
acoth	fix	qmr
acsc	floor	rad2deg
acscd	full	real
acsch	gmres	reallog
angle	gpuArray.speye	realsqrt
asec	imag	round
asecd	isaUnderlying	sec
asech	isdiag	secd
asin	isempty	sech
asind	isequal	sign
asinh	isequaln	sin
atan	isfinite	sind
atand	isfloat	sinh
atanh	isinteger	sinpi
bicg	islogical	size
bicgstab	isnumeric	sparse
ceil	isreal	spfun
cgs	issparse	spones
classUnderlying	istril	sprandsym
conj	istriu	sqrt
cos	isUnderlyingType	sum
cosd	length	tan
cosh	log	tand
cospi	log2	tanh
cot	log10	tfqmr
cotd	log1p	times (.*)
coth	lsqr	trace
csc	minus	transpose
cscd	mtimes	tril
csch	mustBeUnderlyingType	triu
ctranspose	ndims	uminus
deg2rad	nextpow2	underlyingType
diag	nnz	uplus

You can create a sparse gpuArray either by calling `sparse` with a gpuArray input, or by calling `gpuArray` with a sparse input. For example,

```
x = [0 1 0 0 0; 0 0 0 0 1]

     0     1     0     0     0
     0     0     0     0     1

s = sparse(x)

   (1,2)        1
   (2,5)        1

g = gpuArray(s);   % g is a sparse gpuArray
gt = transpose(g); % gt is a sparse gpuArray
f = full(gt)       % f is a full gpuArray
```

```
0    0
1    0
0    0
0    0
0    1
```

Sparse gpuArrays do not support indexing. Instead, use `find` to locate nonzero elements of the array and their row and column indices. Then, replace the values you want and construct a new sparse gpuArray.

Work with Complex Numbers on a GPU

If the output of a function running on the GPU could potentially be complex, you must explicitly specify its input arguments as complex. This applies to `gpuArray` or to functions called in code run by `arrayfun`.

For example, if creating a gpuArray that might have negative elements, use `G = gpuArray(complex(p))`, then you can successfully execute `sqrt(G)`.

Or, within a function passed to `arrayfun`, if `x` is a vector of real numbers, and some elements have negative values, `sqrt(x)` generates an error; instead you should call `sqrt(complex(x))`.

If the result is a gpuArray of complex data and all the imaginary parts are zero, these parts are retained and the data remains complex. This could have an impact when using `sort`, `isreal`, and so on.

The following table lists the functions that might return complex data, along with the input range over which the output remains real.

Function	Input Range for Real Output
`acos(x)`	`abs(x) <= 1`
`acosh(x)`	`x >= 1`
`acoth(x)`	`abs(x) >= 1`
`acsc(x)`	`abs(x) >= 1`
`asec(x)`	`abs(x) >= 1`
`asech(x)`	`0 <= x <= 1`
`asin(x)`	`abs(x) <= 1`
`atanh(x)`	`abs(x) <= 1`
`log(x)`	`x >= 0`
`log1p(x)`	`x >= -1`
`log10(x)`	`x >= 0`
`log2(x)`	`x >= 0`
`power(x,y)`	`x >= 0`
`reallog(x)`	`x >= 0`
`realsqrt(x)`	`x >= 0`
`sqrt(x)`	`x >= 0`

Special Conditions for gpuArray Inputs

GPU-enabled functions run on the GPU only when the data is on the GPU. For example, the following code runs on GPU because the data, the first input, is on the GPU:

```
>> sum(gpuArray(magic(10)),2);
```

However, this code does not run on GPU because the data, the first input, is not on the GPU:

```
>> sum(magic(10),gpuArray(2));
```

If your input argument gpuArrays contain items such as dimensions, scaling factors, or number of iterations, then the function gathers them and computes on the CPU. Functions only run on the GPU when the actual data arguments are gpuArrays.

Acknowledgments

MAGMA is a library of linear algebra routines that take advantage of GPU acceleration. Linear algebra functions implemented for gpuArrays in Parallel Computing Toolbox leverage MAGMA to achieve high performance and accuracy.

See Also
gpuArray | gpuDevice

Related Examples
- "Identify and Select a GPU Device" on page 8-19
- "Establish Arrays on a GPU" on page 8-3

More About
- "GPU Support by Release" on page 8-38
- "GPU Capabilities and Performance" on page 8-2
- MAGMA

Identify and Select a GPU Device

This example shows how to identify and select a GPU for your computations.

To determine how many GPU devices are available in your computer, use the `gpuDeviceCount` function.

```
gpuDeviceCount

    2
```

When there are multiple devices, the first is the default. You can examine its properties with the `gpuDevice` function to determine if that is the one you want to use.

```
d = gpuDevice

d =

  CUDADevice with properties:

                      Name: 'GeForce GTX 1080'
                     Index: 1
         ComputeCapability: '6.1'
            SupportsDouble: 1
             DriverVersion: 9.2000
            ToolkitVersion: 9
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
          MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 8.5899e+09
           AvailableMemory: 7.0053e+09
        MultiprocessorCount: 20
               ClockRateKHz: 1733500
               ComputeMode: 'Default'
        GPUOverlapsTransfers: 1
      KernelExecutionTimeout: 1
          CanMapHostMemory: 1
            DeviceSupported: 1
             DeviceSelected: 1
```

If `d` is the device you want to use, you can proceed. To run computations on the GPU, use `gpuArray` enabled functions. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

To use another device, call `gpuDevice` with the index of the other device.

```
gpuDevice(2)
```

See Also

"Establish Arrays on a GPU" on page 8-3 | "Measure and Improve GPU Performance" on page 8-31 | "Run MATLAB Functions on a GPU" on page 8-9 | "GPU Capabilities and Performance" on page 8-2

More About

- "GPU Support by Release" on page 8-38

Run CUDA or PTX Code on GPU

Overview

This topic explains how to create an executable kernel from CU or PTX (parallel thread execution) files, and run that kernel on a GPU from MATLAB. The kernel is represented in MATLAB by a CUDAKernel object, which can operate on MATLAB array or gpuArray variables.

The following steps describe the CUDAKernel general workflow:

1 Use compiled PTX code to create a CUDAKernel object, which contains the GPU executable code.

2 Set properties on the CUDAKernel object to control its execution on the GPU.

3 Call feval on the CUDAKernel with the required inputs, to run the kernel on the GPU.

MATLAB code that follows these steps might look something like this:

```matlab
% 1. Create CUDAKernel object.
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu','entryPt1');

% 2. Set object properties.
k.GridSize = [8 1];
k.ThreadBlockSize = [16 1];

% 3. Call feval with defined inputs.
g1 = gpuArray(in1); % Input gpuArray.
g2 = gpuArray(in2); % Input gpuArray.

result = feval(k,g1,g2);
```

The following sections provide details of these commands and workflow steps.

Create a CUDAKernel Object

- "Compile a PTX File from a CU File" on page 8-21
- "Construct CUDAKernel Object with CU File Input" on page 8-21
- "Construct CUDAKernel Object with C Prototype Input" on page 8-21
- "Supported Data Types" on page 8-21
- "Argument Restrictions" on page 8-22
- "CUDAKernel Object Properties" on page 8-23
- "Specify Entry Points" on page 8-23
- "Specify Number of Threads" on page 8-24

Compile a PTX File from a CU File

If you have a CU file you want to execute on the GPU, you must first compile it to create a PTX file. One way to do this is with the nvcc compiler in the NVIDIA® CUDA® Toolkit. For example, if your CU file is called myfun.cu, you can create a compiled PTX file with the shell command:

```
nvcc -ptx myfun.cu
```

This generates the file named myfun.ptx.

Construct CUDAKernel Object with CU File Input

With a .cu file and a .ptx file you can create a CUDAKernel object in MATLAB that you can then use to evaluate the kernel:

```
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu');
```

Note You cannot save or load CUDAKernel objects.

Construct CUDAKernel Object with C Prototype Input

If you do not have the CU file corresponding to your PTX file, you can specify the C prototype for your C kernel instead of the CU file. For example:

```
k = parallel.gpu.CUDAKernel('myfun.ptx','float *, const float *, float');
```

Another use for C prototype input is when your source code uses an unrecognized renaming of a supported data type. (See the supported types below.) Suppose your kernel comprises the following code.

```
typedef float ArgType;
__global__ void add3( ArgType * v1, const ArgType * v2 )
{
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

ArgType itself is not recognized as a supported data type, so the CU file that includes it cannot be directly used as input when creating the CUDAKernel object in MATLAB. However, the supported input types to the add3 kernel can be specified as C prototype input to the CUDAKernel constructor. For example:

```
k = parallel.gpu.CUDAKernel('test.ptx','float *, const float *','add3');
```

Supported Data Types

The supported C/C++ standard data types are listed in the following table.

Float Types	Integer Types	Boolean and Character Types
double, double2 float, float2	short, unsigned short, short2, ushort2 int, unsigned int, int2, uint2 long, unsigned long, long2, ulong2 long long, unsigned long long, longlong2, ulonglong2 ptrdiff_t, size_t	bool char, unsigned char, char2, uchar2

Also, the following integer types are supported when you include the `tmwtypes.h` header file in your program.

Integer Types
int8_T, int16_T, int32_T, int64_T uint8_T, uint16_T, uint32_T, uint64_T

The header file is shipped as *matlabroot*/extern/include/tmwtypes.h. You include the file in your program with the line:

```
#include "tmwtypes.h"
```

Argument Restrictions

All inputs can be scalars or pointers, and can be labeled `const`.

The C declaration of a kernel is always of the form:

```
__global__ void aKernel(inputs ...)
```

- The kernel must return nothing, and operate only on its input arguments (scalars or pointers).
- A kernel is unable to allocate any form of memory, so all outputs must be pre-allocated before the kernel is executed. Therefore, the sizes of all outputs must be known before you run the kernel.
- In principle, all pointers passed into the kernel that are not `const` could contain output data, since the many threads of the kernel could modify that data.

When translating the definition of a kernel in C into MATLAB:

- All scalar inputs in C (`double`, `float`, `int`, etc.) must be scalars in MATLAB, or scalar (i.e., single-element) gpuArray variables.
- All `const` pointer inputs in C (`const double *`, etc.) can be scalars or matrices in MATLAB. They are cast to the correct type, copied onto the device, and a pointer to the first element is passed to the kernel. No information about the original size is passed to the kernel. It is as though the kernel has directly received the result of `mxGetData` on an `mxArray`.
- All nonconstant pointer inputs in C are transferred to the kernel exactly as nonconstant pointers. However, because a nonconstant pointer could be changed by the kernel, this will be considered as an output from the kernel.

- Inputs from MATLAB workspace scalars and arrays are cast into the requested type and then passed to the kernel. However, gpuArray inputs are not automatically cast, so their type and complexity must exactly match those expected.

These rules have some implications. The most notable is that every output from a kernel must necessarily also be an input to the kernel, since the input allows the user to define the size of the output (which follows from being unable to allocate memory on the GPU).

CUDAKernel Object Properties

When you create a kernel object without a terminating semicolon, or when you type the object variable at the command line, MATLAB displays the kernel object properties. For example:

```
k = parallel.gpu.CUDAKernel('conv.ptx','conv.cu')

k =
  parallel.gpu.CUDAKernel handle
  Package: parallel.gpu

  Properties:
       ThreadBlockSize: [1 1 1]
    MaxThreadsPerBlock: 512
              GridSize: [1 1 1]
      SharedMemorySize: 0
            EntryPoint: '_Z8theEntryPf'
    MaxNumLHSArguments: 1
       NumRHSArguments: 2
         ArgumentTypes: {'in single vector'  'inout single vector'}
```

The properties of a kernel object control some of its execution behavior. Use dot notation to alter those properties that can be changed.

For a descriptions of the object properties, see the CUDAKernel object reference page. A typical reason for modifying the settable properties is to specify the number of threads, as described below.

Specify Entry Points

If your PTX file contains multiple entry points, you can identify the particular kernel in myfun.ptx that you want the kernel object k to refer to:

```
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu','myKernel1');
```

A single PTX file can contain multiple entry points to different kernels. Each of these entry points has a unique name. These names are generally mangled (as in C++ mangling). However, when generated by nvcc the PTX name always contains the original function name from the CU file. For example, if the CU file defines the kernel function as

```
__global__ void simplestKernelEver( float * x, float val )
```

then the PTX code contains an entry that might be called _Z18simplestKernelEverPff.

When you have multiple entry points, specify the entry name for the particular kernel when calling CUDAKernel to generate your kernel.

Note The `CUDAKernel` function searches for your entry name in the PTX file, and matches on any substring occurrences. Therefore, you should not name any of your entries as substrings of any others.

You might not have control over the original entry names, in which case you must be aware of the unique mangled derived for each. For example, consider the following function template.

```
template <typename T>
__global__ void add4( T * v1, const T * v2 )
{
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

When the template is expanded out for float and double, it results in two entry points, both of which contain the substring `add4`.

```
template __global__ void add4<float>(float *, const float *);
template __global__ void add4<double>(double *, const double *);
```

The PTX has corresponding entries:

```
_Z4add4IfEvPT_PKS0_
_Z4add4IdEvPT_PKS0_
```

Use entry point `add4If` for the float version, and `add4Id` for the double version.

```
k = parallel.gpu.CUDAKernel('test.ptx','double *, const double *','add4Id');
```

Specify Number of Threads

You specify the number of computational threads for your CUDAKernel by setting two of its object properties:

- `GridSize` — A vector of three elements, the product of which determines the number of blocks.
- `ThreadBlockSize` — A vector of three elements, the product of which determines the number of threads per block. (Note that the product cannot exceed the value of the property `MaxThreadsPerBlock`.)

The default value for both of these properties is [1 1 1], but suppose you want to use 500 threads to run element-wise operations on vectors of 500 elements in parallel. A simple way to achieve this is to create your CUDAKernel and set its properties accordingly:

```
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu');
k.ThreadBlockSize = [500,1,1];
```

Generally, you set the grid and thread block sizes based on the sizes of your inputs. For information on thread hierarchy, and multiple-dimension grids and blocks, see the NVIDIA CUDA C Programming Guide.

Run a CUDAKernel

- "Use Workspace Variables" on page 8-25
- "Use gpuArray Variables" on page 8-25

- "Determine Input and Output Correspondence" on page 8-25

Use the `feval` function to evaluate a CUDAKernel on the GPU. The following examples show how to execute a kernel using MATLAB workspace variables and gpuArray variables.

Use Workspace Variables

Assume that you have already written some kernels in a native language and want to use them in MATLAB to execute on the GPU. You have a kernel that does a convolution on two vectors; load and run it with two random input vectors:

```
k = parallel.gpu.CUDAKernel('conv.ptx','conv.cu');

result = feval(k,rand(100,1),rand(100,1));
```

Even if the inputs are constants or variables for MATLAB workspace data, the output is `gpuArray`.

Use gpuArray Variables

It might be more efficient to use `gpuArray` objects as input when running a kernel:

```
k = parallel.gpu.CUDAKernel('conv.ptx','conv.cu');

i1 = gpuArray(rand(100,1,'single'));
i2 = gpuArray(rand(100,1,'single'));

result1 = feval(k,i1,i2);
```

Because the output is a `gpuArray`, you can now perform other operations using this input or output data without further transfers between the MATLAB workspace and the GPU. When all your GPU computations are complete, gather your final result data into the MATLAB workspace:

```
result2 = feval(k,i1,i2);

r1 = gather(result1);
r2 = gather(result2);
```

Determine Input and Output Correspondence

When calling `[out1, out2] = feval(kernel, in1, in2, in3)`, the inputs `in1`, `in2`, and `in3` correspond to each of the input arguments to the C function within your CU file. The outputs `out1` and `out2` store the values of the first and second non-const pointer input arguments to the C function after the C kernel has been executed.

For example, if the C kernel within a CU file has the following signature:

```
void reallySimple( float * pInOut, float c )
```

the corresponding kernel object (k) in MATLAB has the following properties:

```
MaxNumLHSArguments: 1
   NumRHSArguments: 2
      ArgumentTypes: {'inout single vector'  'in single scalar'}
```

Therefore, to use the kernel object from this code with `feval`, you need to provide `feval` two input arguments (in addition to the kernel object), and you can use one output argument:

```
y = feval(k,x1,x2)
```

The input values `x1` and `x2` correspond to `pInOut` and `c` in the C function prototype. The output argument `y` corresponds to the value of `pInOut` in the C function prototype after the C kernel has executed.

The following is a slightly more complicated example that shows a combination of const and non-const pointers:

```
void moreComplicated( const float * pIn, float * pInOut1, float * pInOut2 )
```

The corresponding kernel object in MATLAB then has the properties:

```
MaxNumLHSArguments: 2
   NumRHSArguments: 3
     ArgumentTypes: {'in single vector'  'inout single vector'  'inout single vector'}
```

You can use `feval` on this code's kernel (k) with the syntax:

```
[y1,y2] = feval(k,x1,x2,x3)
```

The three input arguments `x1`, `x2`, and `x3`, correspond to the three arguments that are passed into the C function. The output arguments `y1` and `y2`, correspond to the values of `pInOut1` and `pInOut2` after the C kernel has executed.

Complete Kernel Workflow

- "Add Two Numbers" on page 8-26
- "Add Two Vectors" on page 8-27
- "Example with CU and PTX Files" on page 8-27

Add Two Numbers

This example adds two doubles together in the GPU. You should have the NVIDIA CUDA Toolkit installed, and have CUDA-capable drivers for your device.

1 The CU code to do this is as follows.

```
__global__ void add1( double * pi, double c )
{
    *pi += c;
}
```

The directive `__global__` indicates that this is an entry point to a kernel. The code uses a pointer to send out the result in `pi`, which is both an input and an output. Put this code in a file called `test.cu` in the current directory.

2 Compile the CU code at the shell command line to generate a PTX file called `test.ptx`.

```
nvcc -ptx test.cu
```

3 Create the kernel in MATLAB. Currently this PTX file only has one entry so you do not need to specify it. If you were to put more kernels in, you would specify `add1` as the entry.

```
k = parallel.gpu.CUDAKernel('test.ptx','test.cu');
```

4 Run the kernel with two numeric inputs. By default, a kernel runs on one thread.

```
result = feval(k,2,3)
```

```
result =
    5
```

Add Two Vectors

This example extends the previous one to add two vectors together. For simplicity, assume that there are exactly the same number of threads as elements in the vectors and that there is only one thread block.

1   The CU code is slightly different from the last example. Both inputs are pointers, and one is constant because you are not changing it. Each thread will simply add the elements at its thread index. The thread index must work out which element this thread should add. (Getting these thread- and block-specific values is a very common pattern in CUDA programming.)

```
__global__ void add2( double * v1, const double * v2 )
{
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

Save this code in the file `test.cu`.

2   Compile as before using `nvcc`.

```
nvcc -ptx test.cu
```

3   If this code was put in the same CU file along with the code of the first example, you need to specify the entry point name this time to distinguish it.

```
k = parallel.gpu.CUDAKernel('test.ptx','test.cu','add2');
```

4   Before you run the kernel, set the number of threads correctly for the vectors you want to add.

```
N = 128;
k.ThreadBlockSize = N;
in1 = ones(N,1,'gpuArray');
in2 = ones(N,1,'gpuArray');
result = feval(k,in1,in2);
```

Example with CU and PTX Files

For an example that shows how to work with CUDA, and provides CU and PTX files for you to experiment with, see "Illustrating Three Approaches to GPU Computing: The Mandelbrot Set" on page 9-182.

Run MEX-Functions Containing CUDA Code

In this section...
"Write a MEX-File Containing CUDA Code" on page 8-28
"Run the Resulting MEX-Functions" on page 8-28
"Comparison to a CUDA Kernel" on page 8-29
"Access Complex Data" on page 8-29
"Compile a GPU MEX-File" on page 8-30

Write a MEX-File Containing CUDA Code

As with any MEX-files, those containing CUDA code have a single entry point, known as `mexFunction`. The MEX-function contains the host-side code that interacts with gpuArray objects from MATLAB and launches the CUDA code. The CUDA code in the MEX-file must conform to the CUDA runtime API.

You should call the function `mxInitGPU` at the entry to your MEX-file. This ensures that the GPU device is properly initialized and known to MATLAB.

The interface you use to write a MEX-file for gpuArray objects is different from the MEX interface for standard MATLAB arrays.

You can see an example of a MEX-file containing CUDA code at:

matlabroot/toolbox/parallel/gpu/extern/src/mex/mexGPUExample.cu

This file contains the following CUDA device function:

```
void __global__ TimesTwo(double const * const A,
                         double * const B,
                         int const N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        B[i] = 2.0 * A[i];
}
```

It contains the following lines to determine the array size and launch a grid of the proper size:

```
N = (int)(mxGPUGetNumberOfElements(A));
blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
TimesTwo<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, N);
```

Run the Resulting MEX-Functions

The MEX-function in this example multiplies every element in the input array by 2 to get the values in the output array. To test it, start with a gpuArray in which every element is 1:

```
x = ones(4,4,'gpuArray');
y = mexGPUExample(x)

y =
```

```
2    2    2    2
2    2    2    2
2    2    2    2
2    2    2    2
```

Both the input and output arrays are gpuArray objects:

```
disp(['class(x) = ',class(x),', class(y) = ',class(y)])
```

```
class(x) = gpuArray, class(y) = gpuArray
```

Comparison to a CUDA Kernel

Parallel Computing Toolbox also supports `CUDAKernel` objects that can be used to integrate CUDA code with MATLAB. Consider the following when choosing the MEX-file approach versus the `CUDAKernel` approach:

- MEX-files can interact with host-side libraries, such as the NVIDIA Performance Primitives (NPP) or CUFFT libraries, and can also contain calls from the host to functions in the CUDA runtime library.
- MEX-files can analyze the size of the input and allocate memory of a different size, or launch grids of a different size, from C or C++ code. In comparison, MATLAB code that calls `CUDAKernel` objects must preallocate output memory and determine the grid size.

Access Complex Data

Complex data on a GPU device is stored in interleaved complex format. That is, for a complex gpuArray A, the real and imaginary parts of element `i` are stored in consecutive addresses. MATLAB uses CUDA built-in vector types to store complex data on the device (see the NVIDIA CUDA C Programming Guide).

Depending on the needs of your kernel, you can cast the pointer to complex data either as the real type or as the built-in vector type. For example, in MATLAB, suppose you create a matrix:

```
a = complex(ones(4,'gpuArray'),ones(4,'gpuArray'));
```

If you pass a gpuArray to a MEX-function as the first argument (prhs[0]), then you can get a pointer to the complex data by using the calls:

```
mxGPUArray const * A = mxGPUCreateFromMxArray(prhs[0]);
mwSize numel_complex = mxGPUGetNumberOfElements(A);
double2 * d_A = (double2 const *)(mxGPUGetDataReadOnly(A));
```

To treat the array as a real double-precision array of twice the length, you could do it this way:

```
mxGPUArray const * A = mxGPUCreateFromMxArray(prhs[0]);
mwSize numel_real =2*mxGPUGetNumberOfElements(A);
double * d_A = (double const *)(mxGPUGetDataReadOnly(A));
```

Various functions exist to convert data between complex and real formats on the GPU. These operations require a copy to interleave the data. The function `mxGPUCreateComplexGPUArray` takes two real mxGPUArrays and interleaves their elements to produce a single complex mxGPUArray of the same length. The functions `mxGPUCopyReal` and `mxGPUCopyImag` each copy either the real or the imaginary elements into a new real mxGPUArray. (There is no equivalent of the `mxGetImagData` function for mxGPUArray objects.)

Compile a GPU MEX-File

To compile CUDA code you must have installed the CUDA toolkit version consistent with the `ToolkitVersion` property of the `gpuDevice` object.

Use the `mexcuda` command in MATLAB to compile a MEX-file containing the CUDA code. You can compile the example file using the command:

```
mexcuda mexGPUExample.cu
```

If `mexcuda` has trouble locating the NVIDIA compiler (`nvcc`), it might be installed in a non-default location. You can specify the location of `nvcc` on your system by storing it in the environment variable `MW_NVCC_PATH`. You can set this variable using the MATLAB `setenv` command. For example,

```
setenv('MW_NVCC_PATH','/usr/local/CUDA/bin')
```

Only a subset of Visual Studio® compilers is supported for mexcuda. For details, consult the NVIDIA toolkit documentation.

See Also

Related Examples
- "Accessing Advanced CUDA Features Using MEX" on page 9-203

Measure and Improve GPU Performance

Getting Started with GPU Benchmarking

You can use various benchmark tests in MATLAB to measure the performance of your GPU:

- Use `gpuBench` in MATLAB Central File Exchange to do various tests, including both memory and compute intensive tasks in both single and double precision. Compare the performance of a display card with a compute card. For more information, see https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.
- Use the `paralleldemo_gpu_bench` script in "Measuring GPU Performance" on page 9-171 to obtain information on your PCI bus speed, GPU memory read/write and peak calculation performances for double precision matrix calculations.

Improve Performance Using Single Precision Calculations

You can improve the performance of your GPU by doing your calculations in single precision instead of double precision. In CPU computations, on the other hand, you do not get this improvement when switching from double to single precision. The reason is that most GPU cards are designed for graphic display, demanding high single precision performance.

Typical examples of calculations suitable for single-precision computation on the GPU include image processing and machine learning, see e.g. https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/d/Deep_Learning_in_Cloud_Whitepaper.pdf. However, other types of calculations, such as linear algebra problems, typically require double precision processing.

You can get a performance improvement of up to a factor of 50 for single compared to double precision calculations, depending on the GPU card and total number of cores. High end compute cards typically show a smaller improvement. You can determine the performance improvement of your particular GPU by using `gpuBench`, see https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.

For a comprehensive performance overview of NVIDIA GPU cards, see https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units. You can calculate the performance improvement factor between single precision and double precision as follows:

- Find the GPU on the wiki page above.
- Get the stated single and double precision performance values from the table. If there is no double precision GFLOPS value, assume the ratio is 24-32x slower for double precision.
- Divide the stated single precision GFLOPS value by the double precision GFLOPS value.

Note If you have a mobile graphics card in your laptop, you can use this card for GPU computing. However, the laptop GPU is likely to be much less powerful than the desktop machine equivalent and so performance is reduced.

Basic Workflow for Improving Performance

The purpose of GPU computing in MATLAB is to speed up your applications. This topic discusses fundamental concepts and practices that can help you achieve better performance on the GPU, such

as the configuration of the GPU hardware and best practices within your code. It discusses the trade-off between implementation difficulty and performance, and describes the criteria you might use to choose between using gpuArray functions, `arrayfun`, MEX-files, or CUDA kernels. Finally, it describes how to accurately measure performance on the GPU.

When converting MATLAB code to run on the GPU, it is best to start with MATLAB code that already performs well. While the GPU and CPU have different performance characteristics, the general guidelines for writing good MATLAB code also help you write good MATLAB code for the GPU. The first step is almost always to profile your CPU code. The lines of code that the profiler shows taking the most time on the CPU will likely be ones that you must concentrate on when you code for the GPU.

It is easiest to start converting your code using MATLAB built-in functions that support gpuArray data. These functions take gpuArray inputs, perform calculations on the GPU, and return gpuArray outputs. A list of the MATLAB functions that support gpuArray data is found in "Run MATLAB Functions on a GPU" on page 8-9. In general, these functions support the same arguments and data types as standard MATLAB functions that are calculated on the CPU.

If all the functions that you want to use are supported on the GPU, running code on the GPU may be as simple as calling `gpuArray` to transfer input data to the GPU, and calling `gather` to retrieve the output data from the GPU when finished. In many cases, you might need to vectorize your code, replacing looped scalar operations with MATLAB matrix and vector operations. While vectorizing is generally a good practice on the CPU, it is usually critical for achieving high performance on the GPU. For more information, see "Vectorize for Improved GPU Performance" on page 8-35.

Advanced Tools for Improving Performance

It is possible that even after converting inputs to gpuArrays and vectorizing your code, there are operations in your algorithm that are either not built-in functions, or that are not fast enough to meet your application's requirements. In such situations you have three main options: use `arrayfun` to precompile element-wise parts of your application, make use of GPU library functions, or write a custom CUDA kernel.

If you have a purely element-wise function, you can improve its performance by calling it with `arrayfun`. The `arrayfun` function on the GPU turns an element-wise MATLAB function into a custom CUDA kernel, thus reducing the overhead of performing the operation. Often, there is a subset of your application that can be used with `arrayfun` even if the entire application cannot be. The example "Improve Performance of Element-wise MATLAB® Functions on the GPU using ARRAYFUN" on page 9-169 shows the basic concepts of this approach; and the example "Using GPU ARRAYFUN for Monte-Carlo Simulations" on page 9-192 shows how this can be done in simulations for a finance application.

MATLAB provides an extensive library of GPU-enabled functions in Parallel Computing Toolbox, Image Processing Toolbox, Signal Processing Toolbox, and other products. However, there are many libraries of additional functions that do not have direct built-in analogs in MATLAB's GPU support. Examples include the NVIDIA Performance Primitives library and the CURAND library, which are included in the CUDA toolkit that ships with MATLAB. If you need to call a function in one of these libraries, you can do so using the GPU MEX interface. This interface allows you to extract the pointers to the device data from MATLAB gpuArrays so that you can pass these pointers to GPU functions. You can convert the returned values into gpuArrays for return to MATLAB. For more information see "Run MEX-Functions Containing CUDA Code" on page 8-28.

Finally, you have the option of writing a custom CUDA kernel for the operation that you need. Such kernels can be directly integrated into MATLAB using the CUDAKernel object.

The example "Illustrating Three Approaches to GPU Computing: The Mandelbrot Set" on page 9-182 shows how to implement a simple calculation using three of the approaches mentioned in this section. This example begins with MATLAB code that is easily converted to run on the GPU, rewrites the code to use `arrayfun` for element-wise operations, and finally shows how to integrate a custom CUDA kernel for the same operation.

Alternately, you can write a CUDA kernel as part of a MEX-file and call it using the CUDA Runtime API inside the MEX-file. Either of these approaches might let you work with low-level features of the GPU, such as shared memory and texture memory, that are not directly available in MATLAB code. For more details, see the example "Accessing Advanced CUDA Features Using MEX" on page 9-203.

Best Practices for Improving Performance

Hardware Configuration

In general you can achieve the best performance when your GPU is dedicated to computing. It is usually not practical to use the same GPU device for both computations and graphics, because of the amount of memory taken up for problems of reasonable size and the constant use of the device by the system for graphics. If possible, obtain a separate device for graphics. Details of configuring your device for compute or graphics depend on the operating system and driver version.

On Windows systems, a GPU device can be in one of two modes: Windows Display Driver Model (WDDM) or Tesla Compute Cluster (TCC) mode. For best performance, any devices used for computing should be in TCC mode. Consult NVIDIA documentation for more details.

NVIDIA's highest-performance compute devices, the Tesla line, support error correcting codes (ECC) when reading and writing GPU memory. The purpose of ECC is to correct for occasional bit-errors that occur normally when reading or writing dynamic memory. One technique to improve performance is to turn off ECC to increase the achievable memory bandwidth. While the hardware can be configured this way, MathWorks does not recommend this practice. The potential loss of accuracy due to silent errors can be more harmful than the performance benefit.

MATLAB Coding Practices

This topic describes general techniques that help you achieve better performance on the GPU. Some of these tips apply when writing MATLAB code for the CPU as well.

Data in MATLAB arrays is stored in column-major order. Therefore, it is beneficial to operate along the first or column dimension of your array. If one dimension of your data is significantly longer than others, you might achieve better performance if you make that the first dimension. Similarly, if you frequently operate along a particular dimension, it is usually best to have it as the first dimension. In some cases, if consecutive operations target different dimensions of an array, it might be beneficial to transpose or permute the array between these operations.

GPUs achieve high performance by calculating many results in parallel. Thus, matrix and higher-dimensional array operations typically perform much better than operations on vectors or scalars. You can achieve better performance by rewriting your loops to make use of higher-dimensional operations. The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called vectorization. For more details, see "Using Vectorization".

By default, all operations in MATLAB are performed in double-precision floating-point arithmetic. However, most operations support a variety of data types, including integer and single-precision floating-point. Today's GPUs and CPUs typically have much higher throughput when performing single-precision operations, and single-precision floating-point data occupies less memory. If your application's accuracy requirements allow the use of single-precision floating-point, it can greatly improve the performance of your MATLAB code.

The GPU sits at the end of a data transfer mechanism known as the PCI bus. While this bus is an efficient, high-bandwidth way to transfer data from the PC host memory to various extension cards, it is still much slower than the overall bandwidth to the global memory of the GPU device or of the CPU (for more details, see the example "Measuring GPU Performance" on page 9-171). In addition, transfers from the GPU device to MATLAB host memory cause MATLAB to wait for all pending operations on the device to complete before executing any other statements. This can significantly hurt the performance of your application. In general, you should limit the number of times you transfer data between the MATLAB workspace and the GPU. If you can transfer data to the GPU once at the start of your application, perform all the calculations you can on the GPU, and then transfer the results back into MATLAB at the end, that generally results in the best performance. Similarly, when possible it helps to create arrays directly on the GPU, using either the `'gpuArray'` or the `'like'` option for functions such as `zeros` (e.g., `Z = zeros(___,'gpuArray')` or `Z = zeros(N,'like',g)` for existing gpuArray g).

Measure Performance on the GPU

The best way to measure performance on the GPU is to use `gputimeit`. This function takes as input a function handle with no input arguments, and returns the measured execution time of that function. It takes care of such benchmarking considerations as repeating the timed operation to get better resolution, executing the function before measurement to avoid initialization overhead, and subtracting out the overhead of the timing function. Also, `gputimeit` ensures that all operations on the GPU have completed before the final timing.

For example, consider measuring the time taken to compute the `lu` factorization of a random matrix A of size N-by-N. You can do this by defining a function that does the `lu` factorization and passing the function handle to `gputimeit`:

```
A = rand(N,'gpuArray');
fh = @() lu(A);
gputimeit(fh,2); % 2nd arg indicates number of outputs
```

You can also measure performance with `tic` and `toc`. However, to get accurate timing on the GPU, you must wait for operations to complete before calling `toc`. There are two ways to do this. You can call `gather` on the final GPU output before calling `toc`: this forces all computations to complete before the time measurement is taken. Alternately, you can use the `wait` function with a `gpuDevice` object as its input. For example, if you wanted to measure the time taken to compute the `lu` factorization of matrix A using `tic`, `toc`, and `wait`, you can do it as follows:

```
gd = gpuDevice();
tic();
[l,u] = lu(A);
wait(gd);
tLU = toc();
```

You can also use the MATLAB profiler to show how computation time is distributed in your GPU code. Note, that to accomplish timing measurements, the profiler runs each line of code independently, so it cannot account for overlapping (asynchronous) execution such as might occur during normal

operation. For timing whole algorithms, you should use `tic` and `toc`, or `gputimeit`, as described above. Also, the profile might not yield correct results for user-defined MEX functions if they run asynchronously.

Vectorize for Improved GPU Performance

This example shows you how to improve performance by running a function on the GPU instead of the CPU, and by vectorizing the calculations.

Consider a function that performs fast convolution on the columns of a matrix. Fast convolution, which is a common operation in signal processing applications, transforms each column of data from the time domain to the frequency domain, multiplies it by the transform of a filter vector, transforms back to the time domain, and stores the result in an output matrix.

```matlab
function y = fastConvolution(data,filter)
[m,n] = size(data);
% Zero-pad filter to the column length of data, and transform
filter_f = fft(filter,m);

% Create an array of zeros of the same size and class as data
y = zeros(m,n,'like',data);

% Transform each column of data
for ix = 1:n
    af = fft(data(:,ix));
    y(:,ix) = ifft(af .* filter_f);
end
end
```

Execute this function in the CPU on data of a particular size, and measure the execution time using the MATLAB `timeit` function. The `timeit` function takes care of common benchmarking considerations, such as accounting for startup and overhead.

```matlab
a = complex(randn(4096,100),randn(4096,100));  % Data input
b = randn(16,1);                               % Filter input
c = fastConvolution(a,b);                      % Calculate output
ctime = timeit(@()fastConvolution(a,b));       % Measure CPU time
disp(['Execution time on CPU = ',num2str(ctime)]);
```

On a sample machine, this code displays the output:

```
Execution time on CPU = 0.019335
```

Now execute this function on the GPU. You can do this easily by changing the input data to be gpuArrays rather than normal MATLAB arrays. The `'like'` syntax used when creating the output inside the function ensures that `y` will be a gpuArray if `data` is a gpuArray.

```matlab
ga = gpuArray(a);                              % Move array to GPU
gb = gpuArray(b);                              % Move filter to GPU
gc = fastConvolution(ga,gb);                   % Calculate on GPU
gtime = gputimeit(@()fastConvolution(ga,gb));  % Measure GPU time
gerr = max(max(abs(gather(gc)-c)));            % Calculate error
disp(['Execution time on GPU = ',num2str(gtime)]);
disp(['Maximum absolute error = ',num2str(gerr)]);
```

On the same machine, this code displays the output:

```
Execution time on CPU = 0.019335
Execution time on GPU = 0.027235
Maximum absolute error = 1.1374e-14
```

Unfortunately, the GPU is slower than the CPU for this problem. The reason is that the `for`-loop is executing the FFT, multiplication, and inverse FFT operations on individual columns of length 4096. The best way to increase the performance is to vectorize the code, so that a single MATLAB function call performs more calculation. The FFT and IFFT operations are easy to vectorize: `fft(A)` computes the FFT of each column of a matrix A. You can perform a multiply of the filter with every column in a matrix at once using the MATLAB binary scalar expansion function `bsxfun`. The vectorized function looks like this:

```
function y = fastConvolution_v2(data,filter)
m = size(data,1);
% Zero-pad filter to the length of data, and transform
filter_f = fft(filter,m);

% Transform each column of the input
af = fft(data);

% Multiply each column by filter and compute inverse transform
y = ifft(bsxfun(@times,af,filter_f));
end
```

Perform the same experiment using the vectorized function:

```
a = complex(randn(4096,100),randn(4096,100));   % Data input
b = randn(16,1);                                 % Filter input
c = fastConvolution_v2(a,b);                     % Calculate output
ctime = timeit(@()fastConvolution_v2(a,b));      % Measure CPU time
disp(['Execution time on CPU = ',num2str(ctime)]);

ga = gpuArray(a);                                % Move data to GPU
gb = gpuArray(b);                                % Move filter to GPU
gc = fastConvolution_v2(ga, gb);                 % Calculate on GPU
gtime = gputimeit(@()fastConvolution_v2(ga,gb));% Measure GPU time
gerr = max(max(abs(gather(gc)-c)));              % Calculate error
disp(['Execution time on GPU = ',num2str(gtime)]);
disp(['Maximum absolute error = ',num2str(gerr)]);
```

```
Execution time on CPU = 0.010393
Execution time on GPU = 0.0020537
Maximum absolute error = 1.1374e-14
```

In conclusion, vectorizing the code helps both the CPU and GPU versions to run faster. However, vectorization helps the GPU version much more than the CPU. The improved CPU version is nearly twice as fast as the original; the improved GPU version is 13 times faster than the original. The GPU code went from being 40% slower than the CPU in the original version, to about five times faster in the revised version.

Troubleshooting GPUs

If you only have one GPU in your machine, then it is likely that your graphics card is also acting as your display card. In this case, your GPU is probably subject to timeout imposed by the operating system (OS). You can examine this for your GPU as follows:

```
gpuDevice
```

```
ans =

...

KernelExecutionTimeout: 1
```

If `KernelExecutionTimeout = 1`, then your GPU is subject to timeout imposed by the OS, ensuring that the OS is always able to print updates to the screen. If your GPU calculation takes too much time, then the operation is killed. In this case, you must restart MATLAB to resume GPU calculations successfully.

See Also
`gpuDevice`

More About
- "GPU Capabilities and Performance" on page 8-2
- "Establish Arrays on a GPU" on page 8-3
- "Run MATLAB Functions on a GPU" on page 8-9
- "Identify and Select a GPU Device" on page 8-19

GPU Support by Release

To use your GPU with MATLAB, you must install a recent graphics driver. Best practice is to ensure you have the latest driver for your device. Installing the driver is sufficient for most uses of GPUs in MATLAB, including `gpuArray` and GPU-enabled MATLAB functions. You can download the latest drivers for your GPU device at NVIDIA Driver Downloads.

Supported GPUs

To see support for NVIDIA GPU architectures by MATLAB release, consult the following table.

The *cc* numbers show the compute capability of the GPU architecture. To check your GPU compute capability, see `ComputeCapability` in the output of the `gpuDevice` function. Alternatively, see CUDA GPUs (NVIDIA).

MATLAB Release	Ampere (cc8.x)	Turing (cc7.5)	Volta (cc7.0, cc7.2)	Pascal (cc6.x)	Maxwell (cc5.x)	Kepler (cc3.x)	Fermi (cc2.x)	Tesla (cc1.3)	CUDA Toolkit Version
R2020b	⚠	✓	✓	✓	✓†	✓†			10.2
R2020a	✓∗	✓	✓	✓	✓†	✓†			10.1
R2019b	✓∗	✓	✓	✓	✓	✓			10.1
R2019a	✓∗	✓	✓	✓	✓	✓			10.0
R2018b	✓∗	✓	✓	✓	✓	✓			9.1
R2018a	✓∗	✓	✓	✓	✓	✓			9.0
R2017b	✓∗	✓∗	✓∗	✓	✓	✓	✓		8.0
R2017a	✓∗	✓∗	✓∗	✓	✓	✓	✓		8.0
R2016b	✓∗	✓∗	✓∗	✓∗	✓	✓	✓		7.5
R2016a	✓∗	✓∗	✓∗	✓∗	✓	✓	✓		7.5
R2015b	✓∗	✓∗	✓∗	✓∗	✓	✓	✓		7.0
R2015a	✓∗	✓∗	✓∗	✓∗	✓	✓	✓		6.5
R2014b	✓∗	✓∗	✓∗	✓∗	✓	✓	✓		6.0
R2014a	✓∗	✓∗	✓∗	✓∗	✓∗	✓	✓	✓	5.5
R2013b	✓∗	✓∗	✓∗	✓∗	✓∗	✓	✓	✓	5.0
R2013a	✓∗	✓∗	✓∗	✓∗	✓∗	✓	✓	✓	5.0

MATLAB Release	Ampere (cc8.x)	Turing (cc7.5)	Volta (cc7.0, cc7.2)	Pascal (cc6.x)	Maxwell (cc5.x)	Kepler (cc3.x)	Fermi (cc2.x)	Tesla (cc1.3)	CUDA Toolkit Version
R2012b	✔✳	✔✳	✔✳	✔✳	✔✳	✔	✔	✔	4.2
R2012a	✔✳	✔✳	✔✳	✔✳	✔✳	✔✳	✔	✔	4.0
R2011b	✔✳	✔✳	✔✳	✔✳	✔✳	✔✳	✔	✔	4.0
R2011a	✔✳	✔✳	✔✳	✔✳	✔✳	✔✳	✔	✔	3.2
R2010b	✔✳	✔✳	✔✳	✔✳	✔✳	✔✳	✔	✔	3.1

- ✔ – Built-in binary support.

- ✔† – Support for Kepler and Maxwell GPU architectures will be removed in a future release. At that time, using a GPU with MATLAB will require a GPU device with compute capability 6.0 or greater. MATLAB generates a warning the first time you use a Kepler or Maxwell GPU.

- ✔✳ – Supported via forward compatibility. Optimized device libraries must be compiled at runtime from an unoptimized version. Support can be limited and you might see errors and unexpected behaviour. For more information, see "Forward Compatibility for GPU Devices" on page 8-40.

- ⚠ – By default, this architecture is not supported. You can enable support by enabling forward compatibility for GPU devices. You might see errors and unexpected behaviour. For more information, see "Forward Compatibility for GPU Devices" on page 8-40.

CUDA Toolkit

If you want to use CUDA kernel objects or use GPU Coder, you must install a CUDA Toolkit. The CUDA Toolkit contains CUDA libraries and tools for compilation.

Task	Requirements
Use `gpuArray` and GPU-enabled MATLAB functions, or create CUDA enabled MEX-functions.	Get the latest graphics driver at NVIDIA Driver Downloads.

You do not need the CUDA Toolkit as well. |
| Create and use CUDA kernel objects or use GPU Coder. | Install the version of the CUDA Toolkit supported by your MATLAB release. |

For more information about generating CUDA code in MATLAB, see "Run MEX-Functions Containing CUDA Code" on page 8-28 and "Run CUDA or PTX Code on GPU" on page 8-20. Not all compilers supported by the CUDA Toolkit are supported in MATLAB.

For more information about the CUDA Toolkit and to download your supported version, see CUDA Toolkit Archive (NVIDIA).

Forward Compatibility for GPU Devices

Note Starting in R2020b, forward compatibility for GPU devices is disabled by default.

In R2020a and earlier releases, you cannot disable forward compatibility for GPU devices.

Forward compatibility allows you to use a GPU device with an architecture that was released after your version of MATLAB was built, by recompiling the device libraries at runtime.

When forward compatibility is enabled, the CUDA driver recompiles the GPU libraries the first time you access a device with an architecture newer than your MATLAB version. Recompilation can take up to an hour. Increase the CUDA cache size to prevent a recurrence of this delay. For instructions, see "Increase the CUDA Cache Size" on page 8-40.

When forward compatibility is disabled, you cannot perform computations using a GPU device with an architecture that was released after the version of MATLAB you are using was built. You must enable forward compatibility if you want to use this GPU device in MATLAB.

Caution Enabling forward compatibility can result in wrong answers and unexpected behavior during GPU computations.

The degree of success of recompilation of device libraries can vary depending on the device architecture and the CUDA version used by MATLAB. In some cases, forward compatibility does not work as expected and recompilation of the libraries results in errors.

For example, forward compatibility from CUDA version 10.0–10.2 (MATLAB versions R2019a, R2019b, R2020a, and R2020b) to Ampere (compute capability 8.x) has only limited functionality.

You can enable forward compatibility for GPU devices using the following methods.

- Use the function `parallel.gpu.enableCUDAForwardCompatiblity`. Enabling forward compatibility using this method is not persistent between MATLAB sessions.
- Set the environment variable `MW_CUDA_FORWARD_COMPATIBILITY` to 1. This can preserve the forward compatibility between MATLAB sessions. If you change the environment variable while MATLAB is running, you must restart MATLAB to see the effect. For help setting an environment variable, see this example: "Set the MATLABPATH Environment Variable"

Increase the CUDA Cache Size

If your GPU architecture does not have built-in binary support in your MATLAB release, the graphics driver must compile and cache the GPU libraries. This process can take up to an hour the first time you access the GPU from MATLAB. To increase the CUDA cache size to prevent a recurrence of this delay, set the environment variable `CUDA_CACHE_MAXSIZE` to a minimum of `536870912` (512 MB). For help setting an environment variable, see this example: "Set the MATLABPATH Environment Variable".

See Also

Related Examples
- "Identify and Select a GPU Device" on page 8-19

External Websites
- Deep Learning with GPUs and MATLAB

9

Parallel Computing Toolbox Examples

Profile Parallel Code

This example shows how to profile parallel code using the parallel profiler on workers in a parallel pool.

Create a parallel pool.

```
numberOfWorkers = 3;
pool = parpool(numberOfWorkers);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 3).
```

Collect parallel profile data by enabling `mpiprofile.`

```
mpiprofile on
```

Run your parallel code. For the purposes of this example, use a simple `parfor` loop that iterates over a series of values.

```
values = [5 12 13 1 12 5];
tic;
parfor idx = 1:numel(values)
    u = rand(values(idx)*3e4,1);
    out(idx) = max(conv(u,u));
end
toc

Elapsed time is 31.228931 seconds.
```

After the code completes, view the results from the parallel profiler by calling `mpiprofile viewer`. This action also stops profile data collection.

```
mpiprofile viewer
```

The report shows execution time information for each function that runs on the workers. You can explore which functions take the most time in each worker.

Generally, comparing the workers with the minimum and maximum total execution times is useful. To do so, click **Compare (max vs. min TotalTime)** in the report. In this example, observe that `conv` executes multiple times and takes significantly longer in one worker than in the other. This observation suggests that the load might not be distributed evenly across the workers.

Parallel Profile Summary *Generated 23-Aug-2019 16:29:24 using real time.*

Showing **all functions** called in worker **2** compared with worker **1** (worker 1 has min total time)

Automatic Comparison Selection	Manual Comparison Selection		
Compare (max vs. min TotalTime)	Go to worker: max Total Time	Show Figures (all workers):	No Plot
Compare (max vs. min CommTime)	Compare with: min Total Time		Plot Time Histograms
			Plot All Per Worker Communication
			Plot Communication Time Per Worker

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

Function Name comparison with worker 1	Calls	Total Time	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Inter-worker Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
remoteParallelFunction	3	31.200 s	0.008 s	0 s	0 s	0 b	100.0%	
remoteParallelFunction	3	3.579 s	0.008 s	0 s	0 s	0 b	100.0%	
...>make_general_channel/channel_general	2	31.172 s	0.017 s	0 s	0 s	0 b	100.0%	
...>make_general_channel/channel_general	2	3.548 s	0.007 s	0 s	0 s	0 b	100.0%	
conv	2	31.155 s	31.155 s	0 s	0 s	0 b	100.0%	
conv	2	3.541 s	3.541 s	0 s	0 s	0 b	100.0%	

- If you do not know the workload of each iteration, then a good practice is to randomize the iterations, such as in the following sample code.

```
values = values(randperm(numel(values)));
```

- If you do know the workload of each iteration in your `parfor` loop, then you can use `parforOptions` to control the partitioning of iterations into subranges for the workers. For more information, see `parforOptions`.

In this example, the greater `values(idx)` is, the more computationally intensive the iteration is. Create a set of `parfor` options to divide the `parfor` iterations into subranges of size 2 so that the workload is better distributed.

```
opts = parforOptions(pool,"RangePartitionMethod","fixed","SubrangeSize",2);
```

Enable the parallel profiler.

```
mpiprofile on
```

Run the same code as before. To use the `parfor` options, pass them to the second input argument of `parfor`.

```
values = [5 12 13 1 12 5];
tic;
parfor (idx = 1:numel(values),opts)
    u = rand(values(idx)*3e4,1);
    out(idx) = max(conv(u,u));
end
toc
```

```
Elapsed time is 21.077027 seconds.
```

Visualize the parallel profiler results.

```
mpiprofile viewer
```

In the report, select **Compare (max vs. min TotalTime)** to compare the workers with the minimum and maximum total execution times. Observe that this time, the multiple executions of `conv` take a similar amount of time in all workers. The workload is now better distributed.

Parallel Profile Summary *Generated 23-Aug-2019 16:31:29 using real time.*

Showing **all functions** called in worker **1** compared with worker **3** (worker 3 has min total time)

Automatic Comparison Selection	Manual Comparison Selection	Show Figures (all workers):	
Compare (max vs. min TotalTime)	Go to worker: max Total Time		No Plot
Compare (max vs. min CommTime)	Compare with: min Total Time		Plot Time Histograms
			Plot All Per Worker Communication
			Plot Communication Time Per Worker

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

Function Name comparison with worker 3	Calls	Total Time	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Inter-worker Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
remoteParallelFunction	2	21.052 s	0.012 s	0 s	0 s	0 b	100.0%	
remoteParallelFunction	*2*	*20.493 s*	*0.011 s*	*0 s*	*0 s*	*0 b*	*100.0%*	
...>make_general_channel/channel_general	1	21.021 s	0.015 s	0 s	0 s	0 b	100.0%	
...>make_general_channel/channel_general	*1*	*20.462 s*	*0.013 s*	*0 s*	*0 s*	*0 b*	*100.0%*	
conv	2	21.006 s	21.006 s	0 s	0 s	0 b	100.0%	
conv	*2*	*20.449 s*	*20.449 s*	*0 s*	*0 s*	*0 b*	*100.0%*	

See Also
`mpiprofile` | `parforOptions` | `parpool`

Train Network in Parallel with Custom Training Loop

This example shows how to set up a custom training loop to train a network in parallel. In this example, parallel workers train on portions of the overall mini-batch. If you have a GPU, then training happens on the GPU. During training, a `DataQueue` object sends training progress information back to the MATLAB client.

Load Data Set

Load the digit data set and create an image datastore for the data set. Split the datastore into training and test datastores in a randomized way.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsTest] = splitEachLabel(imds,0.9,"randomized");
```

Determine the different classes in the training set.

```
classes = categories(imdsTrain.Labels);
numClasses = numel(classes);
```

Define Network

Define your network architecture and make it into a layer graph by using the `layerGraph` function. This network architecture includes batch normalization layers, which track the mean and variance statistics of the data set. When training in parallel, combine the statistics from all of the workers at the end of each iteration step, to ensure the network state reflects the whole mini-batch. Otherwise, the network state can diverge across the workers. If you are training stateful recurrent neural networks (RNNs), for example, using sequence data that has been split into smaller sequences to train networks containing LSTM or GRU layers, you must also manage the state between the workers.

```
layers = [
    imageInputLayer([28 28 1],'Name','input','Normalization','none')
    convolution2dLayer(5,20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')];

lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph. `dlnetwork` objects allow for training with custom loops.

```
dlnet = dlnetwork(lgraph)

dlnet =
  dlnetwork with properties:
```

```
        Layers: [11×1 nnet.cnn.layer.Layer]
   Connections: [10×2 table]
    Learnables: [14×3 table]
         State: [6×3 table]
    InputNames: {'input'}
   OutputNames: {'fc'}
```

Set Up Parallel Environment

Determine if GPUs are available for MATLAB to use with the `canUseGPU` function.

- If there are GPUs available, then train on the GPUs. Create a parallel pool with as many workers as GPUs.

- If there are no GPUs available, then train on the CPUs. Create a parallel pool with the default number of workers.

```
if canUseGPU
    executionEnvironment = "gpu";
    numberOfGPUs = gpuDeviceCount;
    pool = parpool(numberOfGPUs);
else
    executionEnvironment = "cpu";
    pool = parpool;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

Get the number of workers in the parallel pool. Later in this example, you divide the workload according to this number.

```
N = pool.NumWorkers;
```

To send data back from the workers during training, create a `DataQueue` object. Use `afterEach` to set up a function, `displayTrainingProgress`, to call each time a worker sends data. `displayTrainingProgress` is a supporting function, defined at the end of this example, that displays the training progress information that comes from the workers.

```
Q = parallel.pool.DataQueue;
afterEach(Q,@displayTrainingProgress);
```

Train Model

Specify the training options.

```
numEpochs = 20;
miniBatchSize = 128;
velocity = [];
```

For GPU training, a recommended practice is to scale up the mini-batch size linearly with the number of GPUs, in order to keep the workload on each GPU constant. For more related advice, see "Training with Multiple GPUs" (Deep Learning Toolbox).

```
if executionEnvironment == "gpu"
    miniBatchSize = miniBatchSize .* N
end
```

```
miniBatchSize = 256
```

Calculate the mini-batch size for each worker by dividing the overall mini-batch size evenly among the workers. Distribute the remainder across the first workers.

```
workerMiniBatchSize = floor(miniBatchSize ./ repmat(N,1,N));
remainder = miniBatchSize - sum(workerMiniBatchSize);
workerMiniBatchSize = workerMiniBatchSize + [ones(1,remainder) zeros(1,N-remainder)]
```

workerMiniBatchSize = *1×2*

```
   128    128
```

Train the model using a custom parallel training loop, as detailed in the following steps. To execute the code simultaneously on all the workers, use an `spmd` block. Within the `spmd` block, `labindex` gives the index of the worker currently executing the code.

Before training, partition the datastore for each worker by using the `partition` function, and set `ReadSize` to the mini-batch size of the worker.

For each epoch, reset and shuffle the datastore with the `reset` and `shuffle` functions.

For each iteration in the epoch:

- Ensure that all workers have data available before beginning processing it in parallel, by performing a global `and` operation (`gop`) on the result of the `hasdata` function.
- Read a mini-batch from the datastore by using the `read` function, and concatenate the retrieved images into a four-dimensional array of images. Normalize the images so that the pixels take values between `0` and `1`.
- Convert the labels to a matrix of dummy variables that puts labels against observations. Dummy variables contain `1` for the label of the observation and `0` otherwise.
- Convert the mini-batch of data to a `dlarray` object with the underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch). For GPU training, convert the data to `gpuArray`.
- Compute the gradients and the loss of the network on each worker by calling `dlfeval` on the `modelGradients` function. The `dlfeval` function evaluates the helper function `modelGradients` with automatic differentiation enabled, so `modelGradients` can compute the gradients with respect to the loss in an automatic way. `modelGradients` is defined at the end of the example and returns loss and gradients given a network, mini-batch of data, and true labels.
- To obtain the overall loss, aggregate the losses on all workers. This example uses cross entropy for the loss function, and the aggregated loss is the sum of all losses. Before aggregating, normalize each loss by multiplying by the proportion of the overall mini-batch that the worker is working on. Use `gplus` to add all losses together and replicate the results across workers.
- To aggregate and update the gradients of all workers, use the `dlupdate` function with the `aggregateGradients` function. `aggregateGradients` is a supporting function defined at the end of this example. This function uses `gplus` to add together and replicate gradients across workers, following normalization according to the proportion of the overall mini-batch that each worker is working on.
- To aggregate and update the network state, use the `dlupdate` function with the `aggregateState` function. `aggregateState` is a supporting function defined at the end of this example. The function uses `gplus` to add together and replicate the state across all workers,

following normalization according to the proportion of the overall mini-batch that each worker is working on.

- Aggregate the state of the network on all workers. The batchnormalization layers in the network track the mean and variance of the data. Since the complete mini-batch is spread across multiple workers, aggregate the network state after each iteration to compute the mean and variance of the whole minibatch.

- After computing the final gradients, update the network learnable parameters with the `sgdmupdate` function.

- Send training progress information back to the client by using the `send` function with the `DataQueue`. Use only one worker to send data, because all workers have the same loss information. To ensure that data is on the CPU, so that a client machine without a GPU can access it, use `gather` on the `dlarray` before sending it.

```matlab
spmd
    % Partition datastore.
    workerImds = partition(imdsTrain,N,labindex);
    workerImds.ReadSize = workerMiniBatchSize(labindex);

    workerVelocity = velocity;

    iteration = 0;

    for epoch = 1:numEpochs
        % Reset and shuffle the datastore.
        reset(workerImds);
        workerImds = shuffle(workerImds);

        % Loop over mini-batches.
        while gop(@and,hasdata(workerImds))
            iteration = iteration + 1;

            % Read a mini-batch of data.
            [workerXBatch,workerTBatch] = read(workerImds);
            workerXBatch = cat(4,workerXBatch{:});
            workerNumObservations = numel(workerTBatch.Label);

            % Normalize the images.
            workerXBatch =  single(workerXBatch) ./ 255;

            % Convert the labels to dummy variables.
            workerY = zeros(numClasses,workerNumObservations,'single');
            for c = 1:numClasses
                workerY(c,workerTBatch.Label==classes(c)) = 1;
            end

            % Convert the mini-batch of data to dlarray.
            dlworkerX = dlarray(workerXBatch,'SSCB');

            % If training on GPU, then convert data to gpuArray.
            if executionEnvironment == "gpu"
                dlworkerX = gpuArray(dlworkerX);
            end

            % Evaluate the model gradients and loss on the worker.
            [workerGradients,dlworkerLoss,workerState] = dlfeval(@modelGradients,dlnet,dlworkerX,
```

```
                % Aggregate the losses on all workers.
                workerNormalizationFactor = workerMiniBatchSize(labindex)./miniBatchSize;
                loss = gplus(workerNormalizationFactor*extractdata(dlworkerLoss));

                % Aggregate the network state on all workers
                workerState.Value = dlupdate(@aggregateState,workerState.Value,{workerNormalizationF
                dlnet.State = workerState;

                % Aggregate the gradients on all workers.
                workerGradients.Value = dlupdate(@aggregateGradients,workerGradients.Value,{workerNor

                % Update the network parameters using the SGDM optimizer.
                [dlnet.Learnables,workerVelocity] = sgdmupdate(dlnet.Learnables,workerGradients,worke
            end

        % Display training progress information.
        if labindex == 1
            data = [epoch loss];
            send(Q,gather(data));
        end
    end
  end
end
```

Test Model

After you train the network, you can test its accuracy.

Load the test images into memory by using `readall` on the test datastore, concatenate them, and normalize them.

```
XTest = readall(imdsTest);
XTest = cat(4,XTest{:});
XTest = single(XTest) ./ 255;
YTest = imdsTest.Labels;
```

After the training is complete, all workers have the same complete trained network. Retrieve any of them.

```
dlnetFinal = dlnet{1};
```

To classify images using a `dlnetwork` object, use the `predict` function on a `dlarray`.

```
dlYPredScores = predict(dlnetFinal,dlarray(XTest,'SSCB'));
```

From the predicted scores, find the class with the highest score with the `max` function. Before you do that, extract the data from the `dlarray` with the `extractdata` function.

```
[~,idx] = max(extractdata(dlYPredScores),[],1);
YPred = classes(idx);
```

To obtain the classification accuracy of the model, compare the predictions on the test set against the true labels.

```
accuracy = mean(YPred==YTest)
```

```
accuracy = 0.9990
```

Define Helper Functions

Define a function, `modelGradients`, to compute the gradients of the loss with respect to the learnable parameters of the network. This function computes the network outputs for a mini-batch X with `forward` and `softmax` and calculates the loss, given the true outputs, using cross entropy. When you call this function with `dlfeval`, automatic differentiation is enabled, and `dlgradient` can compute the gradients of the loss with respect to the learnables automatically.

```
function [dlgradients,dlloss,state] = modelGradients(dlnet,dlX,dlY)
    [dlYPred,state] = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);

    dlloss = crossentropy(dlYPred,dlY);
    dlgradients = dlgradient(dlloss,dlnet.Learnables);
end
```

Define a function to display training progress information that comes from the workers. The `DataQueue` in this example calls this function every time a worker sends data.

```
function displayTrainingProgress (data)
    disp("Epoch: " + data(1) + ", Loss: " + data(2));
end
```

Define a function that aggregates the gradients on all workers by adding them together. `gplus` adds together and replicates all the gradients on the workers. Before adding them together, normalize them by multiplying them by a factor that represents the proportion of the overall mini-batch that the worker is working on. To retrieve the contents of a `dlarray`, use `extractdata`.

```
function gradients = aggregateGradients(dlgradients,factor)
    gradients = extractdata(dlgradients);
    gradients = gplus(factor*gradients);
end
```

Define a function that aggregates the network state on all workers. The network state contains the batch normalization statistics of the data set that are calculated as a weighted average of the mean and variance across the training iterations. Since each worker only sees a portion of the mini-batch, aggregate the network state so that the statistics are representative of the statistics across all the data.

```
function state = aggregateState(state, factor)
    state = gplus(factor*state);
end
```

See Also

dlarray | dlfeval | dlgradient | dlnetwork | gplus | gpuArray | parallel.pool.DataQueue | parpool | partition | read | spmd

Related Examples

- "Run Single Programs on Multiple Data Sets" on page 3-2
- "Train Network Using Custom Training Loop" (Deep Learning Toolbox)

More About

- "Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

Solve Differential Equation Using Multigrid Preconditioner on Distributed Discretization

This example shows how to solve Poisson's equation using a preconditioned iterative solver and distributed arrays. By using distributed arrays, you can scale up the calculation using the memory of a cluster of machines, not just the memory of a single machine. You can scale up without changing your code.

This example continues the topics covered in "Use Distributed Arrays to Solve Systems of Linear Equations with Iterative Methods" on page 9-112. Based on [1], the example models heat distribution in a room by using Poisson's equation, in a form known as the homogeneous steady-state heat equation. Steady state means that the heat does not vary with time and homogeneous means that there is no external heat source.

$$-\Delta u = -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) = 0$$

In the equation, u represents the temperature at every point (x, y, z) of the room. To solve the equation, you first approximate it by a system of linear equations using a finite difference discretization method. Then, you use the preconditioned conjugate gradients (`pcg`) method to solve the system. Preconditioning transforms the problem to improve the performance of the numerical solver. By using distributed arrays, you can leverage the combined memory of a cluster of machines and allow finer discretizations.

Learn how to:

- Set up a discrete 3-D grid and boundary conditions.
- Define a discretization and a multigrid preconditioner.
- Apply a preconditioned numerical solver to solve the heat equation across the 3-D volume.

Discretize Spatial Dimensions

In this example, a cube of side 1 models the room. The first step is to discretize it using a 3-D grid.

The preconditioning method in this example uses several grids with different levels of granularity. Each level coarsens the grid by a factor of 2 in each dimension. Define the number of multigrid levels.

```
multigridLevels = 2;
```

Define the number of points in each dimension, X, Y, and Z, of the finest grid. The preconditioning method requires that the number of points in this grid be divisible by `2^multigridLevels`. In this case, the number of points must be divisible by 4, because the number of multigrid levels is 2.

```
numPoints.X = 32;
numPoints.Y = 32;
numPoints.Z = 32;
```

Discretize the spatial dimensions with a 3-D grid by using the `meshgrid` function. Divide each dimension uniformly according to the number of points by using `linspace`. Note that, to include the boundaries of the cube, you must add two additional points.

```
[X,Y,Z] = meshgrid(linspace(0,1,numPoints.X+2), ...
    linspace(0,1,numPoints.Y+2), ...
    linspace(0,1,numPoints.Z+2));
```

Define Boundary Conditions

Suppose the room has a window and a door. The walls and ceiling have a constant temperature of 0 degrees, the window has a constant temperature of 16 degrees, and the door has a constant temperature of 15 degrees. The floor is at 0.5 degrees. The goal is to determine the temperature distribution across the interior of the room.

Define the coordinates of the floor, window, and door using relational operators, and define the temperature on these boundary elements. The boundaries are the facets of the cube and, therefore, one of X, Y, or Z must be 0 or 1. Set the rest of the boundary and the interior of the cube to 0.

```
floor  = (0.0 <= X & X <= 1.0) & (0.0 <= Y & Y <= 1)   & (Z == 0.0);
window = (X == 1)              & (0.2 <= Y & Y <= 0.8) & (0.4 <= Z & Z <= 0.6);
door   = (0.4 <= X & X <= 0.6) & (Y == 1.0)            & (0.0 <= Z & Z <= 0.6);

u = zeros(size(X));
u(floor)  = 0.5;
u(window) = 16;
u(door)   = 15;
```

These boundary conditions specify the constant values that a solution must take along the boundary of the domain. This type of boundary condition is known as the Dirichlet boundary condition.

Visualize the boundary conditions using the `slice` function. Use slices positioned at the boundary of the cube that show the nonzero boundary conditions.

```
xSlices = 1;
ySlices = 1;
zSlices = 0;
f = slice(X,Y,Z,u,xSlices,ySlices,zSlices,'nearest');
title('Constant nonzero boundary conditions'), xlabel('x'), ylabel('y'), zlabel('z');
colorbar, colormap cool;
shading interp;
set(f,'EdgeColor',[0 0 0]);
```

Constant nonzero boundary conditions
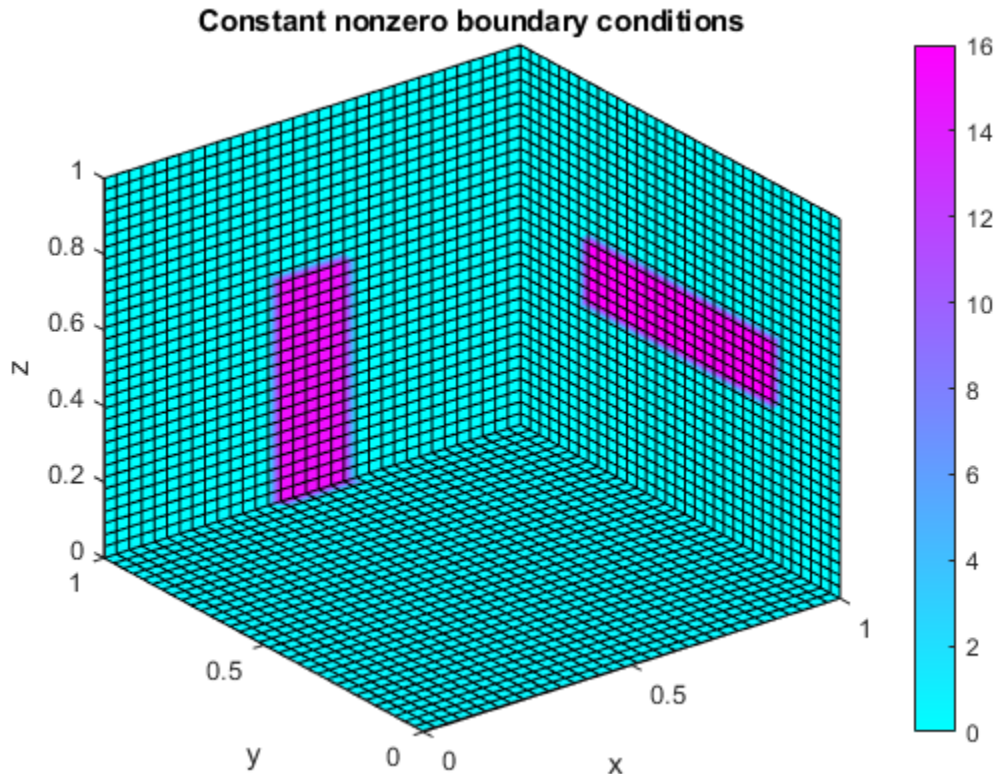
Discretize and Solve Differential Equation

This example discretizes the differential equation into a linear system using a finite differences approximation method, and uses a multigrid preconditioner to improve the performance of the iterative solver. For this example, use the discretization and the preconditioner in the supporting functions `discretizePoissonEquation` and `multigridPreconditioner`. For other problems, choose a discretization and a preconditioner that are appropriate for your application.

In this example, `discretizePoissonEquation` discretizes Poisson's equation with a seven-point-stencil finite differences method into multiple grids with different levels of granularity. The function creates a multigrid structure of discretizations, with precomputed triangular factorizations and operators that map between coarse and fine levels. The preconditioner uses this multigrid information to approximate the solution in an efficient way.

Among other techniques, this preconditioner applies smoothing to minimize errors with a series of approximations. Define the number of smoothing steps. Using a greater number of steps makes approximations more accurate, but also more computationally intensive. Then, discretize the differential equation and set up the preconditioner.

```
numberOfSmootherSteps = 1;
[A,b,multigridData] = discretizePoissonEquation(numPoints,multigridLevels,numberOfSmootherSteps,
```

```
Level 0: The problem is of dimension 32768 with 223232 nonzeros.
Level 1: The problem is of dimension 4096 with 27136 nonzeros.
Level 2: The problem is of dimension 512 with 3200 nonzeros.
```

```
preconditioner = setupPreconditioner(multigridData);
```

9-15

Solve the linear system using preconditioned conjugate gradients.

```
tol = 1e-12;
maxit = numel(A);
pcg(A,b,tol,maxit,preconditioner);
```

```
pcg converged at iteration 45 to a solution with relative residual 5.4e-13.
```

Scale Up with Distributed Arrays

If you need more computational resources, such as memory, you can scale up using distributed arrays without needing to change your code. Distributed arrays distribute your data across multiple workers and they can leverage the computational performance and memory of a cluster of machines.

Start a pool of parallel workers. By default, `parpool` uses your default cluster. Check your default cluster profile on the MATLAB **Home** tab, in the **Environment** area, in **Parallel > Select a Default Cluster**.

```
parpool;
```

```
Starting parallel pool (parpool) using the 'MyCluster' profile ...
Connected to the parallel pool (number of workers: 12).
```

Distribute the temperature variable u across the memory of the workers in your cluster by using the `distributed` function.

```
distU = distributed(u);
```

You can use the same code as before; no changes are required because the discretization and preconditioner functions create distributed arrays if the input is a distributed array. Many MATLAB functions are enhanced for distributed arrays, so you can work with them in the same way you work with in-memory arrays.

Note that `discretizePoissonEquation` returns a structure containing distributed data. To use distributed data inside a structure in a distributed manner, you must create the structure inside an `spmd` block. You must also call any function that uses it inside an `spmd` block.

```
spmd
    [A,b,multigridData] = discretizePoissonEquation(numPoints,multigridLevels,numberOfSmootherSte
    preconditioner = setupPreconditioner(multigridData);
end
```

```
Analyzing and transferring files to the workers ...done.
Lab  1:
  Level 0: The problem is of dimension 32768 with 223232 nonzeros.
  Level 1: The problem is of dimension 4096 with 27136 nonzeros.
  Level 2: The problem is of dimension 512 with 3200 nonzeros.
```

Use `pcg` inside an `spmd` block to solve the linear system in a distributed manner.

```
spmd
    x = pcg(A,b,tol,maxit,preconditioner);
end
```

```
Lab  1:
  pcg converged at iteration 45 to a solution with relative residual 5.4e-13.
```

Plot Results

The solution from the solver is a vector that fits in memory. Send the data from the workers to the client by using `gather`. Reshape the data back into a 3-D array and reorder the dimensions to produce the final solution. Set the inner part of `u` to this solution. The outer part, the boundary, already contains the value of the boundary conditions.

```
x3D = reshape(gather(x),numPoints.X,numPoints.Y,numPoints.Z);
u(2:end-1,2:end-1,2:end-1) = permute(x3D, [2, 1, 3]);
```

Visualize the solution using the `slice` function. Add additional slices to plot the temperature inside the cube. You can use the `Rotate` tool, or vary the position of the slices, to explore the solution.

```
xSlices = [.5,1];
ySlices = [.5,1];
zSlices = [0,.5];
f = slice(X,Y,Z,u,xSlices,ySlices,zSlices,'nearest');
title('Heat distribution'), xlabel('x'), ylabel('y'), zlabel('z');
colorbar, colormap cool;
shading interp;
```
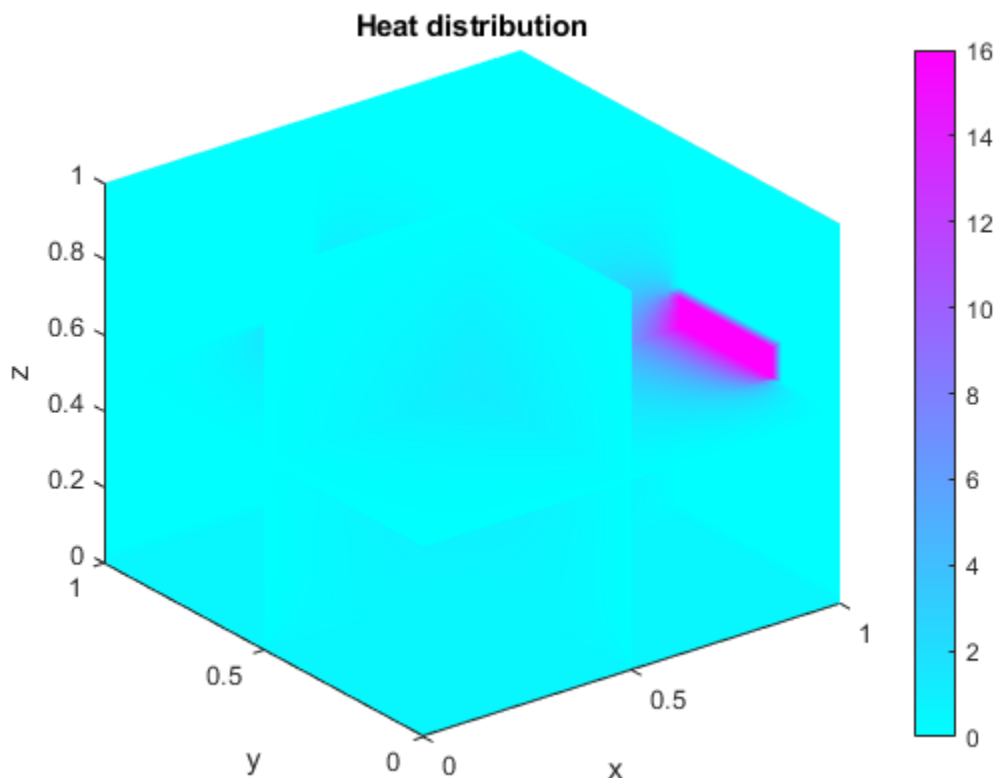

You can try different values of `numPoints` in this example to test different levels of discretization. Using a larger value increases the resolution, but requires more memory. In addition, the larger `multigridLevels` is, the more memory efficient the preconditioner is. However, a larger `multigridLevels` implies a less accurate preconditioner, since coarsening reduces accuracy at every level. As a result, the solver might need more iterations to achieve the same level of accuracy.

Define Preconditioner

Define a multigrid preconditioner for use with the preconditioned conjugate gradients method. This type of preconditioner uses several discretization grids with different levels of granularity to approximate the solution of a system of linear equations more efficiently. The preconditioning method in this example is based on [2], and follows these main stages:

- Presmooth using the Gauss-Seidel approximation method.
- Compute the residual solution on a coarser level.
- Recursively precondition on a coarser level, or solve directly if on the coarsest level.
- Update the solution with a coarser grid solution.
- Postsmooth using the Gauss-Seidel approximation method.

```matlab
function x = multigridPreconditioner(mgData,r,level)

if(level < mgData(level).MaxLevel)
    x = zeros(size(r),'like',r);

    % Presmooth using Gauss-Seidel
    for i=1:mgData(level).NumberOfSmootherSteps
        x = mgData(level).Matrices.LD \ (-mgData(level).Matrices.U*x + r);
        x = mgData(level).Matrices.DU \ (-mgData(level).Matrices.L*x + r);
    end

    % Compute residual on a coarser level
    Axf = mgData(level).Matrices.A*x;
    rc = r(mgData(level).Fine2CoarseOperator)- Axf(mgData(level).Fine2CoarseOperator);

    % Recursive call until coarsest level is reached
    xc = multigridPreconditioner(mgData, rc, level+1);

    % Update solution with prolonged coarse grid solution
    x(mgData(level).Fine2CoarseOperator) = x(mgData(level).Fine2CoarseOperator)+xc;

    % Postsmooth using Gauss-Seidel
    for i = 1:mgData(level).NumberOfSmootherSteps
        x = mgData(level).Matrices.LD \ (-mgData(level).Matrices.U*x + r);
        x = mgData(level).Matrices.DU \ (-mgData(level).Matrices.L*x + r);
    end
else
    % Obtain exact solution on the coarsest level
    x = mgData(level).Matrices.A \ r;
end

end
```

Create a function that takes the multigrid data and returns a function handle that applies the preconditioner to input data. In this example, this function handle is the preconditioner input to `pcg`. You must create this function because it is not possible to define anonymous functions inside `spmd` blocks.

```matlab
function preconditioner = setupPreconditioner(multigridData)

if ~isempty(multigridData)
    preconditioner = @(x,varargin) multigridPreconditioner(multigridData,x,1);
else
```

```
    preconditioner = [];
end

end
```

References

[1] Dongarra, J., M. A. Heroux, and P. Luszczek. "HPCG Benchmark: A New Metric for Ranking High Performance Computing Systems." Knoxville, TN: University of Tennessee, 2015.

[2] Elman, H. C., D. J. Silvester, and A. J. Wathen. *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics.* Oxford, UK: Oxford University Press, 2005, Section 2.5.

See Also
distributed | gather | pcg | spmd

Related Examples
- "Use Distributed Arrays to Solve Systems of Linear Equations with Iterative Methods" on page 9-112
- "Use Distributed Arrays to Solve Systems of Linear Equations with Direct Methods" on page 9-107

Plot During Parameter Sweep with parfeval

This example shows how to perform a parallel parameter sweep with `parfeval` and send results back during computations with a `DataQueue` object. `parfeval` does not block MATLAB, so you can continue working while computations take place.

The example performs a parameter sweep on the Lorenz system of ordinary differential equations, on the parameters σ and ρ, and shows the chaotic nature of this system.

$$\frac{d}{dt}x = \sigma(y - z)$$

$$\frac{d}{dt}y = x(\rho - z) - y$$

$$\frac{d}{dt}z = xy - \beta x$$

Create Parameter Grid

Define the range of parameters that you want to explore in the parameter sweep.

```
gridSize = 40;
sigma = linspace(5, 45, gridSize);
rho = linspace(50, 100, gridSize);
beta = 8/3;
```

Create a 2-D grid of parameters by using the `meshgrid` function.

```
[rho,sigma] = meshgrid(rho,sigma);
```

Create a figure object, and set `'Visible'` to `true` so that it opens in a new window, outside of the live script. To visualize the results of the parameter sweep, create a surface plot. Note that initializing the Z component of the surface with `NaN` creates an empty plot.

```
figure('Visible',true);
surface = surf(rho,sigma,NaN(size(sigma)));
xlabel('\rho','Interpreter','Tex')
ylabel('\sigma','Interpreter','Tex')
```

Set Up Parallel Environment

Create a pool of parallel workers by using the `parpool` function.

```
parpool;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

To send data from the workers, create a `DataQueue` object. Set up a function that updates the surface plot each time a worker sends data by using the `afterEach` function. The `updatePlot` function is a supporting function defined at the end of the example.

```
Q = parallel.pool.DataQueue;
afterEach(Q,@(data) updatePlot(surface,data));
```

Perform Parallel Parameter Sweep

After you define the parameters, you can perform the parallel parameter sweep.

`parfeval` works more efficiently when you distribute the workload. To distribute the workload, group the parameters to explore into partitions. For this example, split into uniform partitions of size `step` by using the colon operator (`:`). The resulting array `partitions` contains the boundaries of the partitions. Note that you must add the end point of the last partition.

```
step = 100;
partitions = [1:step:numel(sigma), numel(sigma)+1]
```

```
partitions = 1×17

         1         101         201         301         401         501         601         701
```

For best performance, try to split into partitions that are:

- Large enough that the computation time is large compared to the overhead of scheduling the partition.
- Small enough that there are enough partitions to keep all workers busy.

To represent function executions on parallel workers and hold their results, use future objects.

```
f(1:numel(partitions)-1) = parallel.FevalFuture;
```

Offload computations to parallel workers by using the `parfeval` function. `parameterSweep` is a helper function defined at the end of this script that solves the Lorenz system on a partition of the parameters to explore. It has one output argument, so you must specify `1` as the number of outputs in `parfeval`.

```
for ii = 1:numel(partitions)-1
    f(ii) = parfeval(@parameterSweep,1,partitions(ii),partitions(ii+1),sigma,rho,beta,Q);
end
```

`parfeval` does not block MATLAB, so you can continue working while computations take place. The workers compute in parallel and send intermediate results through the `DataQueue` as soon as they become available.

If you want to block MATLAB until `parfeval` completes, use the `wait` function on the future objects. Using the `wait` function is useful when subsequent code depends on the completion of `parfeval`.

```
wait(f);
```

After `parfeval` finishes the computations, `wait` finishes and you can execute more code. For example, plot the contour of the resulting surface. Use the `fetchOutputs` function to retrieve the results stored in the future objects.

```
results = reshape(fetchOutputs(f),gridSize,[]);
contourf(rho,sigma,results)
xlabel('\rho','Interpreter','Tex')
ylabel('\sigma','Interpreter','Tex')
```

If your parameter sweep needs more computational resources and you have access to a cluster, you can scale up your `parfeval` computations. For more information, see "Scale Up from Desktop to Cluster" on page 9-88.

Define Helper Functions

Define a helper function that solves the Lorenz system on a partition of the parameters to explore. Send intermediate results to the MATLAB client by using the `send` function on the `DataQueue` object.

```
function results = parameterSweep(first,last,sigma,rho,beta,Q)
    results = zeros(last-first,1);
    for ii = first:last-1
        lorenzSystem = @(t,a) [sigma(ii)*(a(2) - a(1)); a(1)*(rho(ii) - a(3)) - a(2); a(1)*a(2)
        [t,a] = ode45(lorenzSystem,[0 100],[1 1 1]);
        result = a(end,3);
        send(Q,[ii,result]);
        results(ii-first+1) = result;
    end
end
```

Define another helper function that updates the surface plot when new data arrives.

```
function updatePlot(surface,data)
    surface.ZData(data(1)) = data(2);
```

```
    drawnow('limitrate');
end
```

See Also

afterEach | parallel.pool.DataQueue | parfeval | parpool

Related Examples

Perform Webcam Image Acquisition in Parallel with Postprocessing

This example shows how to perform frame acquisition from a webcam in parallel with data postprocessing.

In the example, you use a parallel worker to perform image acquisition and then stream the data back to the client for postprocessing by using a `DataQueue` object.

To perform postprocessing using workers instead of your MATLAB client, see "Perform Image Acquisition and Parallel Image Processing" on page 9-28.

Set Up Parallel Environment

Start a parallel pool with one worker on the local cluster.

```
parpool('local',1);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 1).
```

To send information back from the worker to the MATLAB client, create a `DataQueue` object.

```
D = parallel.pool.DataQueue;
```

Create a figure object, and set `'Visible'` to `'on'` so that it opens outside of the live script. To display images every time they arrive from the `DataQueue` object, use `afterEach`.

```
fig = figure('Visible','on');
afterEach(D,@processDisp);
```

Fetch Data and Perform Postprocessing in Parallel

Define the frequency of acquisition, that is, how many frames per second you want to pull out from the camera.

```
freq = 5;
```

Select a value that takes into account how long postprocessing takes. Otherwise the video stream can significantly lag over time.

To start data acquisition on the parallel worker, call `parfeval` and pass the acquisition function, the `DataQueue` object, and the acquisition rate as arguments.

```
f = parfeval(@getFrameFromCamera,0,D,freq);
```

Acquire frames for a period of 30 seconds. This example applies a blurring filter as the postprocessing step and shows the original and processed frames side by side.

```
pause(30);
```

To stop the video feed, cancel the acquisition.

```
cancel(f);
```

For a more detailed example showing postprocessing on workers, see "Perform Image Acquisition and Parallel Image Processing" on page 9-28.

Define Helper Functions

The `getFrameFromCamera` function connects to the webcam, then acquires image frames and sends them to the `DataQueue` object in an infinite loop.

```
function getFrameFromCamera(D,freq)
    cam = webcam;
    while true
        img = snapshot(cam);

        send(D,img);
        pause(1/freq);
    end
end
```

The `processDisp` function postprocesses frames and displays the original and processed frames each time data arrives to the `DataQueue` object.

```
function processDisp(img)
    imgBlur = imgaussfilt(img,3);
    imshow([img, imgBlur],'Parent',gca)
end
```

See Also

`afterEach` | `imshow` | `parallel.pool.DataQueue` | `parfeval` | `parpool`

Related Examples

- "Perform Image Acquisition and Parallel Image Processing" on page 9-28

Perform Image Acquisition and Parallel Image Processing

This example shows how to perform image acquisition from a webcam and postprocess data in parallel.

In this example, the MATLAB client acquires frames from the video device and then offloads the postprocessing to parallel workers, which filter off the noise from each frame using a denoising neural network. The frames are then written into a video.

In this example, you use `parfeval` to perform postprocessing in the workers and `parallel.pool.Constant` to instantiate the denoising network in the workers to be used during postprocessing. To send the frames back from the workers and ensure that they are written in order, this example uses an `OrderedDataQueue` object.

Extract Device Information and Set Up Video Output

Clear previous image acquisition objects and extract information about the video device currently connected to the machine.

```
objects = imaqfind;
delete(objects);
imaqreset;
deviceInfo = imaqhwinfo('winvideo')

deviceInfo = struct with fields:
        AdaptorDllName: 'adaptor.dll'
     AdaptorDllVersion: '6.1 (R2019b)'
           AdaptorName: 'winvideo'
             DeviceIDs: {[1]}
            DeviceInfo: [1×1 struct]
```

Check if a folder for the output video already exists in the current directory. If no folder for output video exists, create one.

```
if ~isfolder('OutputFolder')
    mkdir OutputFolder
end
```

To write video data to an AVI file in the output folder, create a `VideoWriter` object.

```
videoOut = VideoWriter('OutputFolder/myVideo.avi');
```

Set Up Parallel Environment

To enable the offloading of postprocessing to the workers, first start a parallel pool.

```
p = parpool('local');

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Create a `parallel.pool.Constant` object to create a denoising network only once in the workers and use it to filter the noise out from the frames.

```
C = parallel.pool.Constant(@() denoisingNetwork('dncnn'));
```

To send the postprocessed frames back from the workers and write them in order, use an `OrderedDataQueue`. Set a callback to write the frames to disk by using `afterEach`.

```
Q = OrderedDataQueue;
afterEach(Q,@(frame) writeVideo(videoOut,frame));
```

The `OrderedDataQueue` object is defined in a supporting file to this example. If you want to use it in your own code, copy and place it with the rest of your files.

Set Up Video Input Object

Create a video input object. Set the object to perform acquisition in the client frame by frame.

```
videoIn = videoinput('winvideo',1,'YUY2_800x600')

Summary of Video Input Object Using 'Microsoft® LifeCam Cinema(TM)'.

   Acquisition Source(s):  input1 is available.

  Acquisition Parameters:  'input1' is the current selected source.
                           10 frames per trigger using the selected source.
                           'YUY2_800x600' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

     Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.

videoIn.ReturnedColorSpace = 'RGB';
videoIn.FramesPerTrigger = Inf;
videoIn.FramesAcquiredFcnCount = 1;
```

Set the video writing frame rate to the same rate as for video reading, and open the video output object.

```
src = videoIn.Source;
videoOut.FrameRate = str2double(src.FrameRate);
open(videoOut);
```

To start postprocessing operations after each frame is acquired, define a `FramesAcquiredFcn` callback for the video input object and start the acquisition.

```
videoIn.FramesAcquiredFcn = {@postProcessAndWrite,C,Q};
start(videoIn);
```

Create a preview window. You can stop the video as soon as the preview is manually closed by using `waitfor` on the figure handle `hPreviewFig`. For this example, stop video acquisition after 2 seconds.

```
hPreviewImg = preview(videoIn);
hPreviewFig = ancestor(hPreviewImg,'figure');
pause(2);
stop(videoIn);
```

The postprocessing function stores a future variable in the `UserData` property of the video object. This variable represents a future execution of the video write operations. To close the video writer after all the data is written to the output file, use `afterAll` on this future variable.

```
postProcessFutures = videoIn.UserData;
closeVideoFuture = afterAll(postProcessFutures,@() close(videoOut),0);
```

The postprocessing operation in this example is can take a few minutes. On a Windows 10, Intel® Xeon® W-2133 3.60 GHz CPU, with 6 cores, postprocessing took 4 minutes.

You can use a waitbar to track the postprocessing progress. To update the waitbar after each postprocessing operation finishes, use `afterEach`. To close the waitbar after all operations finish, use `afterAll`. For more information, see "Update a User Interface Asynchronously Using afterEach and afterAll" on page 9-101.

```
h = waitbar(0,'Postprocessing...');
updateWaitbarFuture = afterEach(postProcessFutures, ...
    @(~) waitbar(sum(strcmp('finished',{postProcessFutures.State}))/numel(postProcessFutures),h)
afterAll(closeVideoFuture, @() close(h),0);
```

Block execution in the client session until the writing finishes by waiting for the future variable.

```
wait(closeVideoFuture);
```

Delete the video input object when finished.

```
delete(videoIn);
```

Visualize Results

After the video file has been created, you can visualize the results.

Use a `VideoReader` object to read the video file.

```
vidObj = VideoReader('OutputFolder/MyVideo.avi');
```

Read some frames by using the `readFrame` function.

```
images = cell(1,5);
times = .4:.4:2;
for ii = 1:numel(times)
    vidObj.CurrentTime = times(ii);
    images{ii} = readFrame(vidObj);
end
```

To visualize the frames, use the `montage` function.

```
montage(images,'Size',[1 5])
```
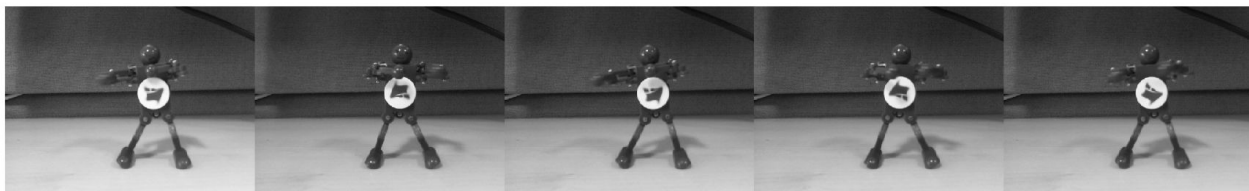
Define Helper Functions

Define the main postprocessing routine, which is executed after each frame acquisition. This function `postProcessAndWrite` fetches the data from the video input object and calls `parfeval` to start the frame denoising in a parallel worker.

```
function postProcessAndWrite(videoIn,~,C,Q)
    [frame,~,metadata] = getdata(videoIn,1);
    postProcessFuture = parfeval(@postProcess,0,frame,C,Q,metadata.FrameNumber);
    videoIn.UserData = [videoIn.UserData postProcessFuture];
end
```

Define the postprocessing function to be executed in the worker. For this example, to simplify computation, convert each frame to gray, and then denoise it by using the `denoiseImage` function. The function `postProcess` takes the frame and the denoising network object stored in the `Value` field of the `parallel.pool.Constant` object as inputs. For more information on denoising images with a denoising neural network, see "Get Pretrained Image Denoising Network" (Image Processing Toolbox).

```
function postProcess(frame,C,Q,frameNumber)
    grayFrame = im2double(rgb2gray(frame));
    denoisedGrayFrame = denoiseImage(grayFrame,C.Value);
    denoisedGrayFrame = im2uint8(denoisedGrayFrame);
    send(Q,frameNumber,denoisedGrayFrame)
end
```

See Also

FramesAcquiredFcn | VideoWriter | afterAll | afterEach | denoiseImage | imaqfind | parallel.pool.Constant | parfeval | videoinput

Related Examples

- "Perform Webcam Image Acquisition in Parallel with Postprocessing" on page 9-26

More About

- "Get Started with Image Processing Toolbox" (Image Processing Toolbox)
- "Get Started with Image Acquisition Toolbox" (Image Acquisition Toolbox)

Run Script as Batch Job

Use `batch` to offload work to a MATLAB worker session that runs in the background. You can continue using MATLAB while computations take place.

Run a script as a batch job by using the `batch` function. By default, `batch` uses your default cluster profile. Check your default cluster profile on the MATLAB **Home** tab, in the **Environment** section, in **Parallel > Select a Default Cluster**. Alternatively, you can specify a cluster profile with the `'Profile'` name-value pair argument.

```
job = batch('myScript');
```

`batch` does not block MATLAB and you can continue working while computations take place.

If you want to block MATLAB until the job finishes, use the `wait` function on the job object.

```
wait(job);
```

By default, MATLAB saves the Command Window output from the batch job to the diary of the job. To retrieve it, use the `diary` function.

```
diary(job)

--- Start Diary ---
n = 100

--- End Diary ---
```

After the job finishes, fetch the results by using the `load` function.

```
load(job,'x');
plot(x)
```

If you want to load all the variables in the batch job, use the `load` function without arguments.

When you have fetched all the required variables, delete the job object to clean up its data and avoid consuming resources unnecessarily.

```
delete(job);
clear job
```

Note that if you send a script file using `batch`, MATLAB transfers all the workspace variables to the cluster, even if your script does not use them. The data transfer time for a large workspace can be substantial. As a best practice, convert your script to a function file to avoid this communication overhead. For an example that uses a function, see "Run Batch Job and Access Files from Workers" on page 11-26.

For more advanced options with `batch`, see "Run Batch Job and Access Files from Workers" on page 11-26.

See Also
`batch` | `load` | `wait`

Related Examples

-

Run Batch Job and Access Files from Workers

You can offload your computations to run in the background by using `batch`. If your code needs access to files, you can use additional options, such as `'AttachedFiles'` or `'AdditionalPaths'`, to make the data accessible. You can close or continue working in MATLAB while computations take place and recover the results later.

Prepare Example

Prepare and copy the supporting files for this example by using the following command.

```
prepareSupportingFiles;
```

Run Batch Job

Create a cluster object using `parcluster`. By default, `parcluster` uses your default cluster profile. Check your default cluster profile on the MATLAB **Home** tab, in the **Environment** section, in **Parallel > Select a Default Cluster**.

```
c = parcluster();
```

Place your code inside a function and submit it as a batch job by using `batch`. For an example of a custom function, see the supporting function `myFunction`. Specify the expected number of output arguments and a cell array with inputs to the function.

If your code uses a parallel pool, use the `'Pool'` name-value pair argument to create a parallel pool with the number of workers that you specify. `batch` uses an additional worker to run the function itself.

By default, `batch` changes the initial working directory of the workers to the current folder of the MATLAB client. It can be useful to control the initial working directory in the workers. For example, you might want to control it if your cluster uses a different filesystem, and therefore the paths are different, such as when you submit from a Windows client machine to a Linux cluster.

- To keep the initial working directory of the workers and use their default, set `'CurrentFolder'` to `'.'`.
- To change the initial working directory, set `'CurrentFolder'` to a folder of your choice.

This example uses a parallel pool with three workers and chooses a temporary location for the initial working directory.

```
job = batch(c,@myFunction,1,{}, ...
    'Pool',3, ...
    'CurrentFolder',tempdir);
```

`batch` offloads the computations in your function to a parallel worker, so you can continue working in MATLAB while computations take place.

If you want to block MATLAB until the job completes, use the `wait` function on the job object.

```
wait(job);
```

To retrieve the results, use `fetchOutputs` on the job object.

If your code has an error, then `fetchOutputs` throws an error. You can access error information by checking the `Error` property of `Task` objects in the job. In this example, the code depends on a file that the workers cannot find.

```
getReport(job.Tasks(1).Error)
```

```
ans =
    'Error using myFunction (line 4)
     Unable to read file 'mydata2.dat'. No such file or directory.'
```

Access Files from Workers

By default, `batch` automatically analyzes your code and transfers required files to the workers. In some cases, you must explicitly transfer those files--for example, when you determine the name of a file at runtime.

In this example, `myFunction` accesses the supporting file `mydata.dat`, which `batch` automatically detects and transfers. The function also accesses `mydata1.dat`, but it resolves the name of the file at runtime, so the automatic dependency analysis does not detect it.

```
type myFunction.m

function X = myFunction()
    A = load("mydata.dat");
    X = zeros(flip(size(A)));
    parfor i = 1:3
        B = load("mydata"+i+".dat");
        X = X + A\B;
    end
end
```

If the data is in a location that the workers can access, you can use the name-value pair argument `'AdditionalPaths'` to specify the location. `'AdditionalPaths'` adds this path to the MATLAB search path of the workers and makes the data visible to them.

```
pathToData = pwd;
job(2) = batch(c,@myFunction,1,{}, ...
    'Pool',3, ...
    'CurrentFolder',tempdir, ...
    'AdditionalPaths',pathToData);
wait(job(2));
```

If the data is in a location that the workers cannot access, you can transfer files to the workers by using the `'AttachedFiles'` name-value pair argument.

```
job(3) = batch(c,@myFunction,1,{}, ...
    'Pool',3, ...
    'CurrentFolder',tempdir, ...
    'AttachedFiles',"mydata"+string(1:3)+".dat");
```

Find Existing Job

You can close MATLAB after job submission and retrieve the results later. Before you close MATLAB, make a note of the job ID.

```
job3ID = job(3).ID
```

```
job3ID = 19
```

When you open MATLAB again, you can find the job by using the `findJob` function.

```
job(3) = findJob(c,'ID',job3ID);
wait(job(3));
```

Alternatively, you can use the Job Monitor to track your job. You can open it from the MATLAB **Home** tab, in the **Environment** section, in **Parallel > Monitor Jobs**.

Retrieve Results and Clean Up Data

To retrieve the results of a batch job, use the `fetchOutputs` function. `fetchOutputs` returns a cell array with the outputs of the function run with `batch`.

```
X = fetchOutputs(job(3))
```

```
X = 1×1 cell array
    {40×207 double}
```

When you have retrieved all the required outputs and do not need the job object anymore, delete it to clean up its data and avoid consuming resources unnecessarily.

```
delete(job)
clear job
```

See Also
`batch` | `fetchOutputs (Job)` | `findJob` | `parcluster`

Benchmark Cluster Workers

This example shows how to run the MATLAB benchmark on your cluster workers. The benchmark measures the execution speed of several MATLAB computations. You can plot these results and compare the performance of the client and workers.

This example uses `pbench`, a function that runs a subset of the tests in `bench`, the MATLAB benchmark. The tests in this subset are LU, FFT, ODE, and Sparse. For details on these tests, see `bench`.

Run the MATLAB benchmark on the client.

```
tClient = pbench
```

```
tClient = 1×4

    0.0766    0.0725    0.0194    0.1311
```

Create a parallel pool `p` using the `parpool` function. By default, `parpool` starts a parallel pool with workers on your default cluster. Select your default cluster on the MATLAB **Home** tab, in the **Environment** area, in **Parallel > Select a Default Cluster**.

```
p = parpool();
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Run the MATLAB benchmark on the workers using the `parfevalOnAll` function. `parfevalOnAll` offloads the execution of a function to all the workers in the pool, and returns a `parallel.FevalOnAllFuture` object to hold the results when they are ready. To obtain the results from the workers, use `fetchOutputs` on the future object.

```
f = parfevalOnAll(@pbench,1);
tWorkers = fetchOutputs(f);
```

Combine the results of the client and workers, and plot them using a bar plot. Compare the relative performances of the workers and client.

```
tClientAndWorkers = [tClient;tWorkers];
bar(tClientAndWorkers');
xticklabels({'LU','FFT','ODE','Sparse'});
xlabel("Benchmark type");
ylabel("Benchmark execution time (seconds)");
workerNames = strcat("Worker ",string(1:size(tWorkers,1)));
legend(["Client",workerNames],'Location','bestoutside');
```
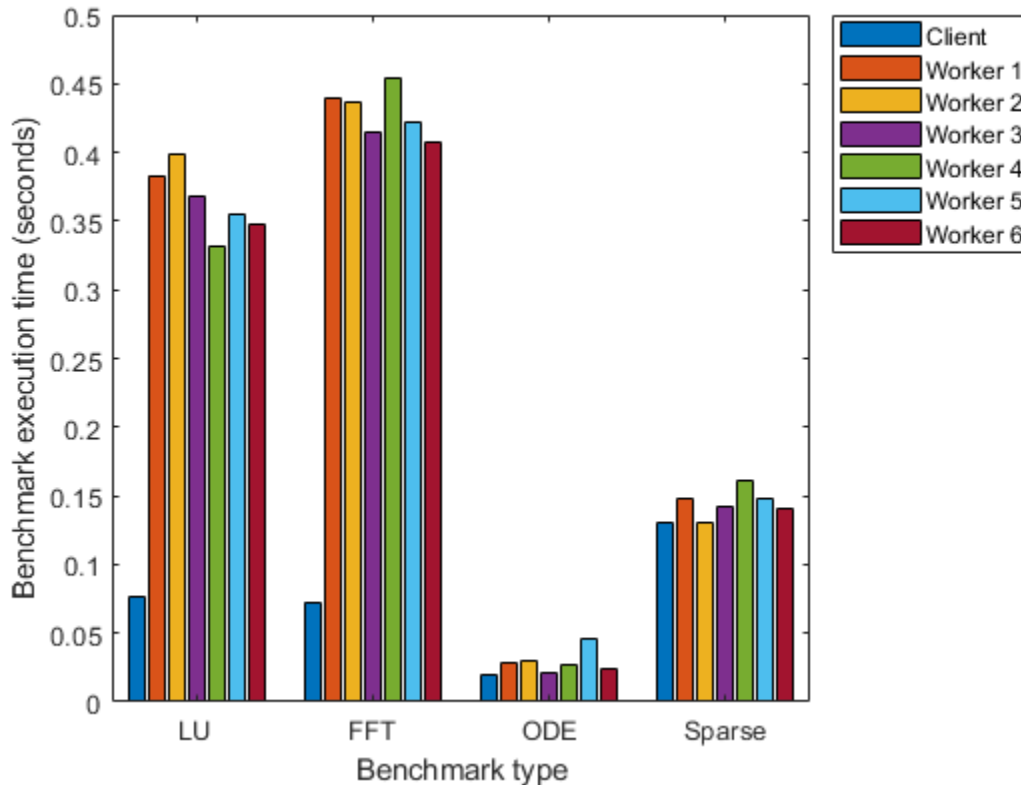
By default, the MATLAB client is enabled for multithreading. Multithreading enables MATLAB numerical functions, such as `lu` or `fft`, to run on multiple cores using multiple computational threads. The workers use a single computational thread by default, because they are typically associated with a single core. Therefore, the LU test, for example, runs faster on the MATLAB client than on the workers. Other problems, such as ODEs, cannot benefit from multithreading, so they perform the same on the MATLAB client and workers. Consider this difference when deciding whether to distribute computations to MATLAB parallel workers, such as with `parfor`. For more details, see "Deciding When to Use parfor" on page 2-2. For more information on multithreading, see Run MATLAB on multicore and multiprocessor machines.

See Also
bench | fetchOutputs (FevalFuture) | parfevalOnAll | parpool

Related Examples

- "Benchmark Your Cluster with the HPC Challenge" on page 9-39
- "Scale Up from Desktop to Cluster" on page 9-88

Benchmark Your Cluster with the HPC Challenge

This example shows how to evaluate the performance of a compute cluster with the HPC Challenge Benchmark. The benchmark consists of several tests that measure different memory access patterns. For more information, see HPC Challenge Benchmark.

Prepare the HPC Challenge

Start a parallel pool of workers in your cluster using the `parpool` function. By default, `parpool` creates a parallel pool using your default cluster profile. Check your default cluster profile on the **Home** tab, in **Parallel > Select a Default Cluster**. In this benchmark, the workers communicate with each other. To ensure that interworker communication is optimized, set `'SpmdEnabled'` to `true`.

```
pool = parpool('SpmdEnabled',true);
```

```
Starting parallel pool (parpool) using the 'MyCluster' profile ...
connected to 64 workers.
```

Use the `hpccDataSizes` function to compute a problem size for each individual benchmark that fulfils the requirements of the HPC Challenge. This size depends on the number of workers and the amount of memory available to each worker. For example, allow use of 1 GB per worker.

```
gbPerWorker = 1;
dataSizes = hpccDataSizes(pool.NumWorkers,gbPerWorker);
```

Run the HPC Challenge

The HPC Challenge benchmark consists of several pieces, each of which explores the performance of different aspects of the system. In the following code, each function runs a single benchmark, and returns a row table that contains performance results. These functions test a variety of operations on distributed arrays. MATLAB partitions distributed arrays across multiple parallel workers, so they can use the combined memory and computational resources of your cluster. For more information on distributed arrays, see "Distributed Arrays".

HPL

`hpccHPL(m)`, known as the Linpack Benchmark, measures the execution rate for solving a linear system of equations. It creates a random distributed real matrix A of size m-by-m and a real random distributed vector b of length m, and measures the time to solve the system x = A\b in parallel. The performance is returned in gigaflops (billions of floating-point operations per second).

```
hplResult = hpccHPL(dataSizes.HPL);
```

```
Starting HPCC benchmark: HPL with data size: 27.8255 GB. Running on a pool of 64 workers
Analyzing and transferring files to the workers ...done.
Finished HPCC benchmark: HPL in 196.816 seconds.
```

DGEMM

`hpccDGEMM(m)` measures the execution rate of real matrix-matrix multiplication. It creates random distributed real matrices A, B, and C, of size m-by-m, and measures the time to perform the matrix multiplication C = beta*C + alpha*A*B in parallel, where alpha and beta are random scalars. The performance is returned in gigaflops.

```
dgemmResult = hpccDGEMM(dataSizes.DGEMM);
```

```
Starting HPCC benchmark: DGEMM with data size: 9.27515 GB. Running on a pool of 64 workers
Analyzing and transferring files to the workers ...done.
Finished HPCC benchmark: DGEMM in 69.3654 seconds.
```

STREAM

hpccSTREAM(m) assesses the memory bandwidth of the cluster. It creates random distributed vectors b and c of length m, and a random scalar k, and computes a = b + c*k. This benchmark does not use interworker communication. The performance is returned in gigabytes per second.

```
streamResult = hpccSTREAM(dataSizes.STREAM);
```

```
Starting HPCC benchmark: STREAM with data size: 10.6667 GB. Running on a pool of 64 workers
Analyzing and transferring files to the workers ...done.
Finished HPCC benchmark: STREAM in 0.0796962 seconds.
```

PTRANS

hpccPTRANS(m) measures the interprocess communication speed of the system. It creates two random distributed matrices A and B of size m-by-m, and computes A' + B. The result is returned in gigabytes per second.

```
ptransResult = hpccPTRANS(dataSizes.PTRANS);
```

```
Starting HPCC benchmark: PTRANS with data size: 9.27515 GB. Running on a pool of 64 workers
Analyzing and transferring files to the workers ...done.
Finished HPCC benchmark: PTRANS in 6.43994 seconds.
```

RandomAccess

hpccRandomAccess(m) measures the number of memory locations in a distributed vector that can be randomly updated per second. The result is retuned in GUPS, giga updates per second. In this test, the workers use a random number generator compiled into a MEX function. Attach a version of this MEX function for each operating system architecture to the parallel pool, so the workers can access the one that corresponds to their operating system.

```
addAttachedFiles(pool,{'hpccRandomNumberGeneratorKernel.mexa64','hpccRandomNumberGeneratorKernel
randomAccessResult = hpccRandomAccess(dataSizes.RandomAccess);
```

```
Starting HPCC benchmark: RandomAccess with data size: 16 GB. Running on a pool of 64 workers
Analyzing and transferring files to the workers ...done.
Finished HPCC benchmark: RandomAccess in 208.103 seconds.
```

FFT

hpccFFT(m) measures the execution rate of a parallel fast Fourier transform (FFT) computation on a distributed vector of length m. This test measures both the arithmetic capability of the system and the communication performance. The performance is returned in gigaflops.

```
fftResult = hpccFFT(dataSizes.FFT);
```

```
Starting HPCC benchmark: FFT with data size: 8 GB. Running on a pool of 64 workers
Analyzing and transferring files to the workers ...done.
Finished HPCC benchmark: FFT in 11.772 seconds.
```

Display the Results

Each benchmark results in a single table row with statistics. Concatenate these rows to provide a summary of the test results.

```
allResults = [hplResult; dgemmResult; streamResult; ...
              ptransResult; randomAccessResult; fftResult];
disp(allResults);
```

Benchmark	DataSizeGB	Time	Performance	PerformanceUnits
"HPL"	27.826	196.82	773.11	"GFlops"
"DGEMM"	9.2752	69.365	1266.4	"GFlops"
"STREAM"	10.667	0.079696	431.13	"GBperSec"
"PTRANS"	9.2752	6.4399	1.5465	"GBperSec"
"RandomAccess"	16	208.1	0.010319	"GUPS"
"FFT"	8	11.772	6.6129	"GFlops"

Offload Computations with `batch`

You can use the `batch` function to offload the computations in the HPC Challenge to your cluster and continue working in MATLAB.

Before using `batch`, delete the current parallel pool. A batch job cannot be processed if a parallel pool is already using all available workers.

```
delete(gcp);
```

Send the function `hpccBenchmark` as a batch job to the cluster by using `batch`. This function invokes the tests in the HPC Challenge and returns the results in a table. When you use `batch`, a worker takes the role of the MATLAB client and executes the function. In addition, specify these name-value pair arguments:

- `'Pool'`: Creates a parallel pool with workers for the job. In this case, specify 32 workers. `hpccBenchmark` runs the HPC Challenge on those workers.

- `'AttachedFiles'`: Transfers files to the workers in the pool. In this case, attach a version of the `hpccRandomNumberGeneratorKernel` for each operating system architecture. The workers access the one that corresponds to their operating system when they execute the `hpccRandomAccess` test.

- `'CurrentFolder'`: Sets the working directory of the workers. If you do not specify this argument, MATLAB changes the current directory of the workers to the current directory in the MATLAB client. Set it to `'.'` if you want to use the current folder of the workers instead. This is useful when the workers have a different file system.

```
gbPerWorker = 1;
job = batch(@hpccBenchmark,1,{gbPerWorker}, ...
    'Pool',32, ...
    'AttachedFiles',{'hpccRandomNumberGeneratorKernel.mexa64','hpccRandomNumberGeneratorKernel.me
    'CurrentFolder','.');
```

After you submit the job, you can continue working in MATLAB. You can check the state of the job by using the Job Monitor. On the **Home** tab, in the **Environment** area, select **Parallel > Monitor Jobs**.

In this case, wait for the job to finish. To retrieve the results back from the cluster, use the `fetchOutputs` function.

```
wait(job);
results = fetchOutputs(job);
disp(results{1})
```

9-41

Benchmark	DataSizeGB	Time	Performance	PerformanceUnits
"HPL"	13.913	113.34	474.69	"GFlops"
"DGEMM"	4.6376	41.915	740.99	"GFlops"
"STREAM"	5.3333	0.074617	230.24	"GBperSec"
"PTRANS"	4.6376	3.7058	1.3437	"GBperSec"
"RandomAccess"	8	189.05	0.0056796	"GUPS"
"FFT"	4	7.6457	4.9153	"GFlops"

When you use large clusters, you increase the available computational resources. If the time spent on calculations outweighs the time spent on interworker communication, then your problem can scale up well. The following figure shows the scaling of the HPL benchmark with the number of workers, on a cluster with 4 machines and 18 physical cores per machine. Note that in this benchmark, the size of the data increases with the number of workers.
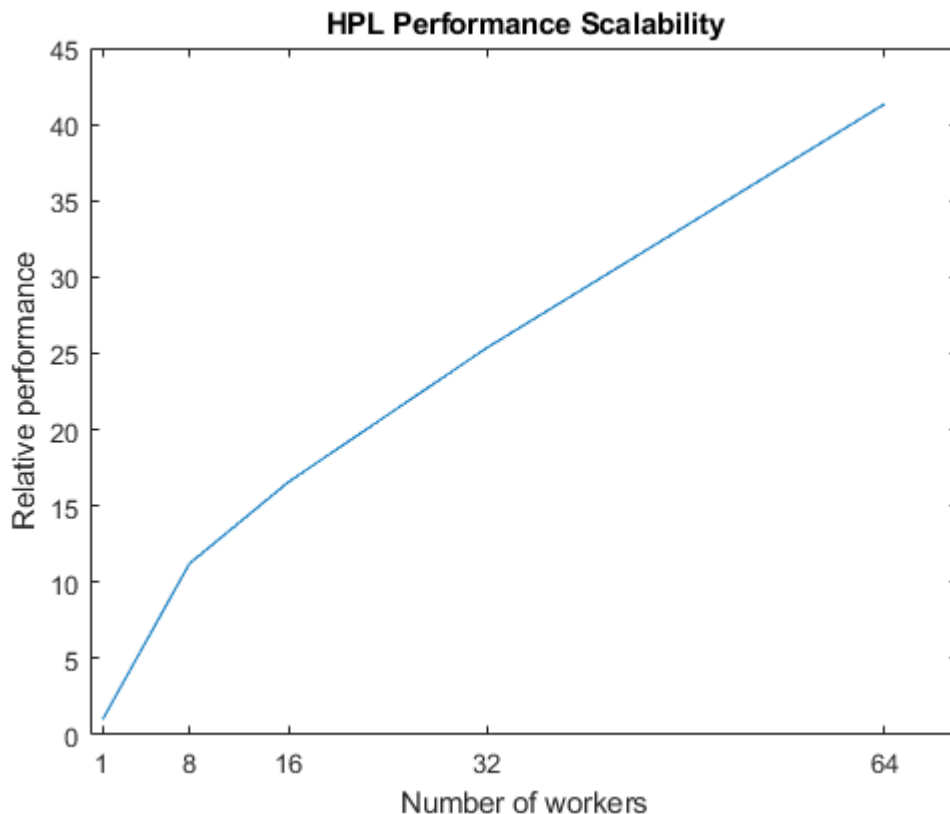
See Also
parpool | batch

Related Examples

Train Deep Learning Networks in Parallel

This example shows how to run multiple deep learning experiments on your local machine. Using this example as a template, you can modify the network layers and training options to suit your specific application needs. You can use this approach with a single or multiple GPUs. If you have a single GPU, the networks train one after the other in the background. The approach in this example enables you to continue using MATLAB® while deep learning experiments are in progress.

As an alternative, you can use Experiment Manager (Deep Learning Toolbox) to interactively train multiple deep networks in parallel. For more information, see "Use Experiment Manager to Train Networks in Parallel" (Deep Learning Toolbox).

Prepare Data Set

Before you can run the example, you must have access to a local copy of a deep learning data set. This example uses a data set with synthetic images of digits from 0 to 9. In the following code, change the location to point to your data set.

```
datasetLocation = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
```

If you want to run the experiments with more resources, you can run this example in a cluster in the cloud.

- Upload the data set to an Amazon S3 bucket. For an example, see "Upload Deep Learning Data to the Cloud" on page 9-77.
- Create a cloud cluster. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. For more information, see "Create Cloud Cluster" on page 5-14.
- Select your cloud cluster as the default, on the **Home** tab, in the **Environment** section, select **Parallel > Select a Default Cluster**.

Load Data Set

Load the data set by using an `imageDatastore` object. Split the data set into training, validation, and test sets.

```
imds = imageDatastore(datasetLocation, ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');

[imdsTrain,imdsValidation,imdsTest] = splitEachLabel(imds,0.8,0.1);
```

To train the network with augmented image data, create an `augmentedImageDatastore` (Deep Learning Toolbox). Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [28 28 1];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

Train Networks in Parallel

Start a parallel pool with as many workers as GPUs. You can check the number of available GPUs by using the `gpuDeviceCount` function. MATLAB assigns a different GPU to each worker. By default, `parpool` uses your default cluster profile. If you have not changed the default, it is `local`. This example was run using a machine with 2 GPUs.

```
numGPUs = 2;
parpool(numGPUs);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

To send training progress information from the workers during training, use a `parallel.pool.DataQueue` object. To learn more about how to use data queues to obtain feedback during training, see the example "Use parfeval to Train Multiple Deep Learning Networks" on page 9-59.

```
dataqueue = parallel.pool.DataQueue;
```

Define the network layers and training options. For code readability, you can define them in a separate function that returns several network architectures and training options. In this case, `networkLayersAndOptions` returns a cell array of network layers and an array of training options of the same length. Open this example in MATLAB and then click `networkLayersAndOptions` to open the supporting function `networkLayersAndOptions`. Paste in your own network layers and options. The file contains sample training options that show how to send information to the data queue using an output function.

```
[layersCell,options] = networkLayersAndOptions(augmentedImdsTrain,imdsValidation,dataqueue);
```

Prepare the training progress plots, and set a callback function to update these plots after each worker sends data to the queue. `preparePlots` and `updatePlots` are supporting functions for this example.

```
handles = preparePlots(numel(layersCell));
```
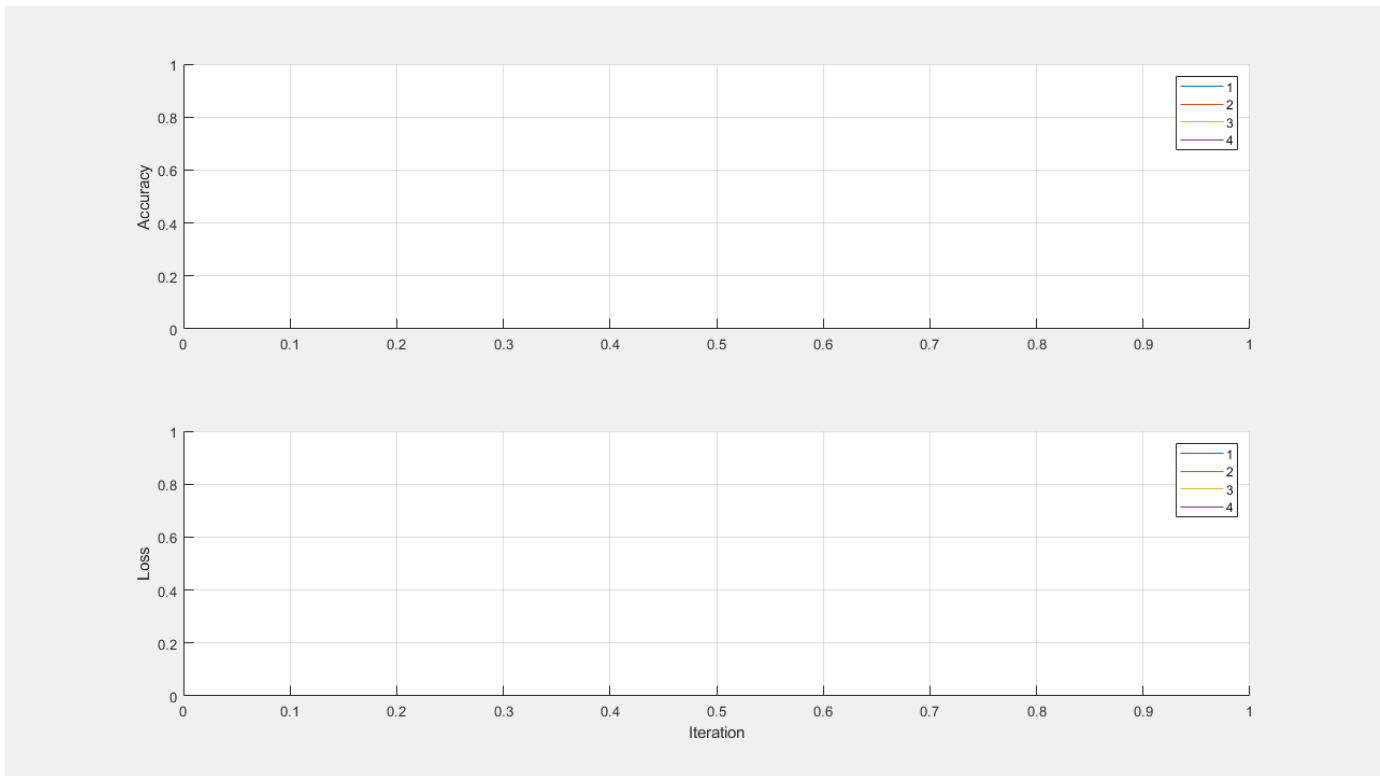
```
afterEach(dataqueue,@(data) updatePlots(handles,data));
```

To hold the computation results in parallel workers, use future objects. Preallocate an array of future objects for the result of each training.

```
trainingFuture(1:numel(layersCell)) = parallel.FevalFuture;
```

Loop through the network layers and options by using a `for` loop, and use `parfeval` to train the networks on a parallel worker. To request two output arguments from `trainNetwork`, specify 2 as the second input argument to `parfeval`.

```
for i=1:numel(layersCell)
    trainingFuture(i) = parfeval(@trainNetwork,2,augmentedImdsTrain,layersCell{i},options(i));
end
```

`parfeval` does not block MATLAB, so you can continue working while the computations take place.

To fetch results from future objects, use the `fetchOutputs` function. For this example, fetch the trained networks and their training information. `fetchOutputs` blocks MATLAB until the results are available. This step can take a few minutes.

```
[network,trainingInfo] = fetchOutputs(trainingFuture);
```

Save the results to disk using the `save` function. To load the results again later, use the `load` function.

```
save(['experiment-' datestr(now,'yyyymmddTHHMMSS')],'network','trainingInfo');
```

Plot Results

After the networks complete training, plot their training progress by using the information in `trainingInfo`.

Use subplots to distribute the different plots for each network. For this example, use the first row of subplots to plot the training accuracy against the number of epoch along with the validation accuracy.

```
figure('Units','normalized','Position',[0.1 0.1 0.6 0.6]);
title('Training Progress Plots');

for i=1:numel(layersCell)
    subplot(2,numel(layersCell),i);
    hold on; grid on;
    ylim([0 100]);
    iterationsPerEpoch = floor(augmentedImdsTrain.NumObservations/options(i).MiniBatchSize);
    epoch = (1:numel(trainingInfo(i).TrainingAccuracy))/iterationsPerEpoch;
    plot(epoch,trainingInfo(i).TrainingAccuracy);
    plot(epoch,trainingInfo(i).ValidationAccuracy,'.k','MarkerSize',10);
end
subplot(2,numel(layersCell),1), ylabel('Accuracy');
```

Then, use the second row of subplots to plot the training loss against the number of epoch along with the validation loss.

```
for i=1:numel(layersCell)
    subplot(2,numel(layersCell),numel(layersCell) + i);
    hold on; grid on;
    ylim([0 max([trainingInfo.TrainingLoss])]);
    iterationsPerEpoch = floor(augmentedImdsTrain.NumObservations/options(i).MiniBatchSize);
    epoch = (1:numel(trainingInfo(i).TrainingAccuracy))/iterationsPerEpoch;
    plot(epoch,trainingInfo(i).TrainingLoss);
    plot(epoch,trainingInfo(i).ValidationLoss,'.k','MarkerSize',10);
    xlabel('Epoch');
end
subplot(2,numel(layersCell),numel(layersCell)+1), ylabel('Loss');
```
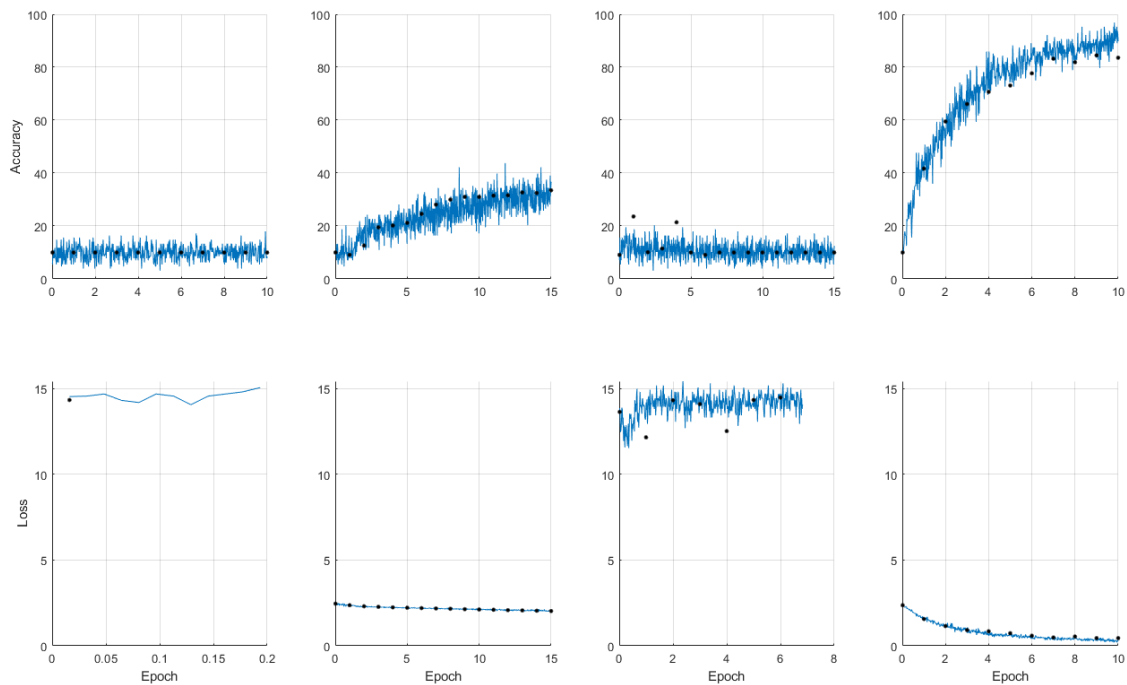

After you choose a network, you can use `classify` (Deep Learning Toolbox) and obtain its accuracy on the test data `imdsTest`.

See Also

`augmentedImageDatastore` (Deep Learning Toolbox) | `imageDatastore` | `parfeval` | `fetchOutputs` | `trainNetwork` (Deep Learning Toolbox) | `trainingOptions` (Deep Learning Toolbox)

Related Examples

- "Train Network Using Automatic Multi-GPU Support" on page 9-49
- "Use parfeval to Train Multiple Deep Learning Networks" on page 9-59

More About

- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

Train Network Using Automatic Multi-GPU Support

This example shows how to use multiple GPUs on your local machine for deep learning training using automatic parallel support. Training deep learning networks often takes hours or days. With parallel computing, you can speed up training using multiple GPUs. To learn more about options for parallel training, see "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox).

Requirements

Before you can run this example, you must download the CIFAR-10 data set to your local machine. The following code downloads the data set to your current directory. If you already have a local copy of CIFAR-10, then you can skip this section.

```
directory = pwd;
[locationCifar10Train,locationCifar10Test] = downloadCIFARToFolders(directory);

Downloading CIFAR-10 data set...done.
Copying CIFAR-10 to folders...done.
```

Load Data Set

Load the training and test data sets by using an `imageDatastore` object. In the following code, ensure that the location of the datastores points to CIFAR-10 in your local machine.

```
imdsTrain = imageDatastore(locationCifar10Train, ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');

imdsTest = imageDatastore(locationCifar10Test, ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');
```

To train the network with augmented image data, create an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

Define Network Architecture and Training Options

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
blockDepth = 4; % blockDepth controls the depth of a convolutional block.
netWidth = 32; % netWidth controls the number of filters in a convolutional block.

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,blockDepth)
```

```
        maxPooling2dLayer(2,'Stride',2)
        convolutionalBlock(2*netWidth,blockDepth)
        maxPooling2dLayer(2,'Stride',2)
        convolutionalBlock(4*netWidth,blockDepth)
        averagePooling2dLayer(8)

        fullyConnectedLayer(10)
        softmaxLayer
        classificationLayer
    ];
```

Define the training options. Train the network in parallel with multiple GPUs by setting the execution environment to `'multi-gpu'`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU constant. In this example, the number of GPUs is two. Scale the learning rate according to the mini-batch size. Use a learning rate schedule to drop the learning rate as the training progresses. Turn on the training progress plot to obtain visual feedback during training.

```
numGPUs = 2;
miniBatchSize = 256*numGPUs;
initialLearnRate = 1e-1*miniBatchSize/256;

options = trainingOptions('sgdm', ...
    'ExecutionEnvironment','multi-gpu', ... % Turn on automatic multi-gpu support.
    'InitialLearnRate',initialLearnRate, ... % Set the initial learning rate.
    'MiniBatchSize',miniBatchSize, ... % Set the MiniBatchSize.
    'Verbose',false, ... % Do not send command line output.
    'Plots','training-progress', ... % Turn on the training progress plot.
    'L2Regularization',1e-10, ...
    'MaxEpochs',60, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',floor(numel(imdsTrain.Files)/miniBatchSize), ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',50);
```

Train Network and Use for Classification

Train the network. During training, the plot displays the progress.

```
net = trainNetwork(augmentedImdsTrain,layers,options)

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

```
net =
  SeriesNetwork with properties:

    Layers: [43×1 nnet.cnn.layer.Layer]
```
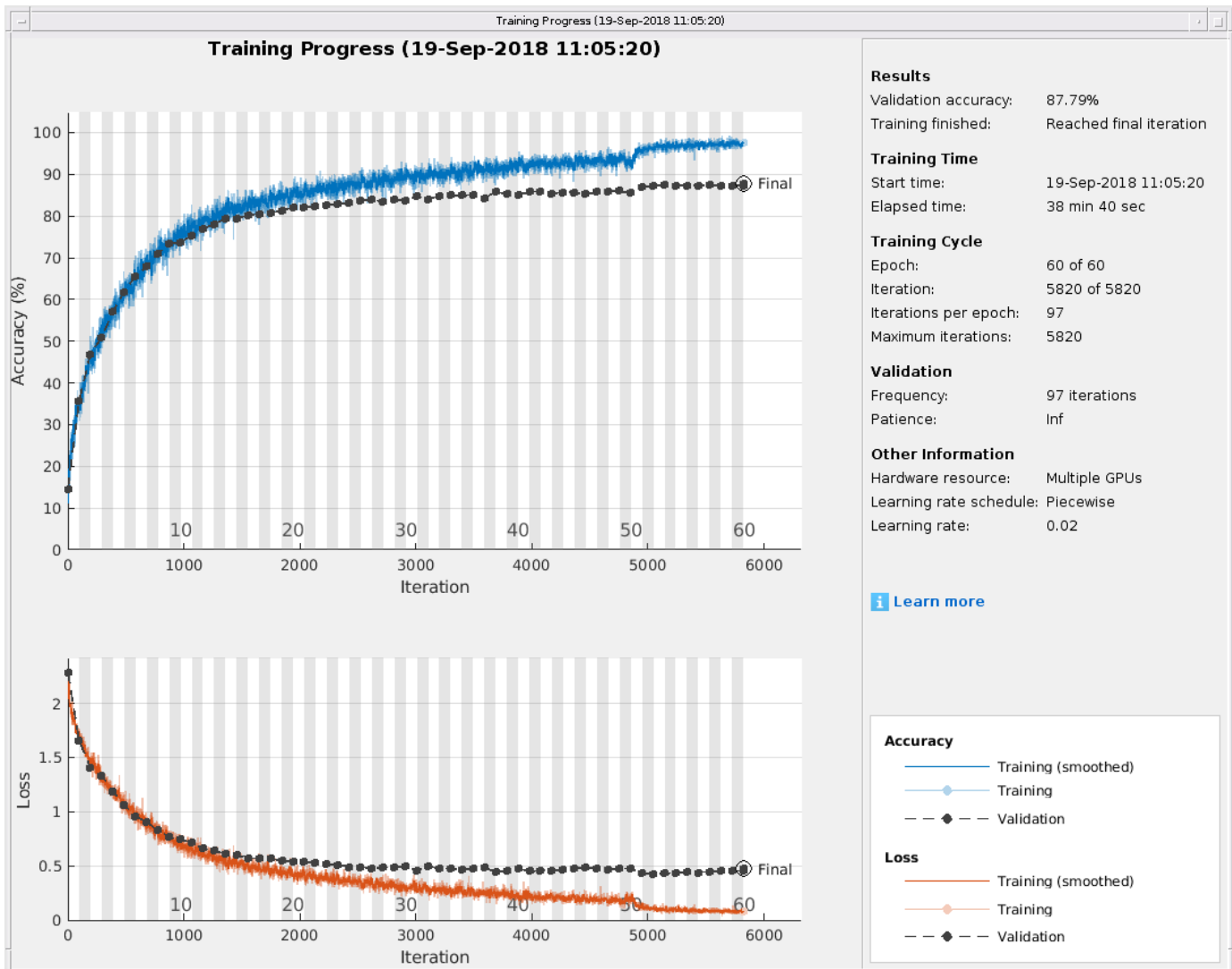
Determine the accuracy of the network by using the trained network to classify the test images on your local machine. Then compare the predicted labels to the actual labels.

```
YPredicted = classify(net,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

```
accuracy = 0.8779
```

Automatic multi-GPU support can speed up network training by taking advantage of several GPUs. The following plot shows the speedup in the overall training time with the number of GPUs on a Linux machine with four NVIDIA© TITAN Xp GPUs.
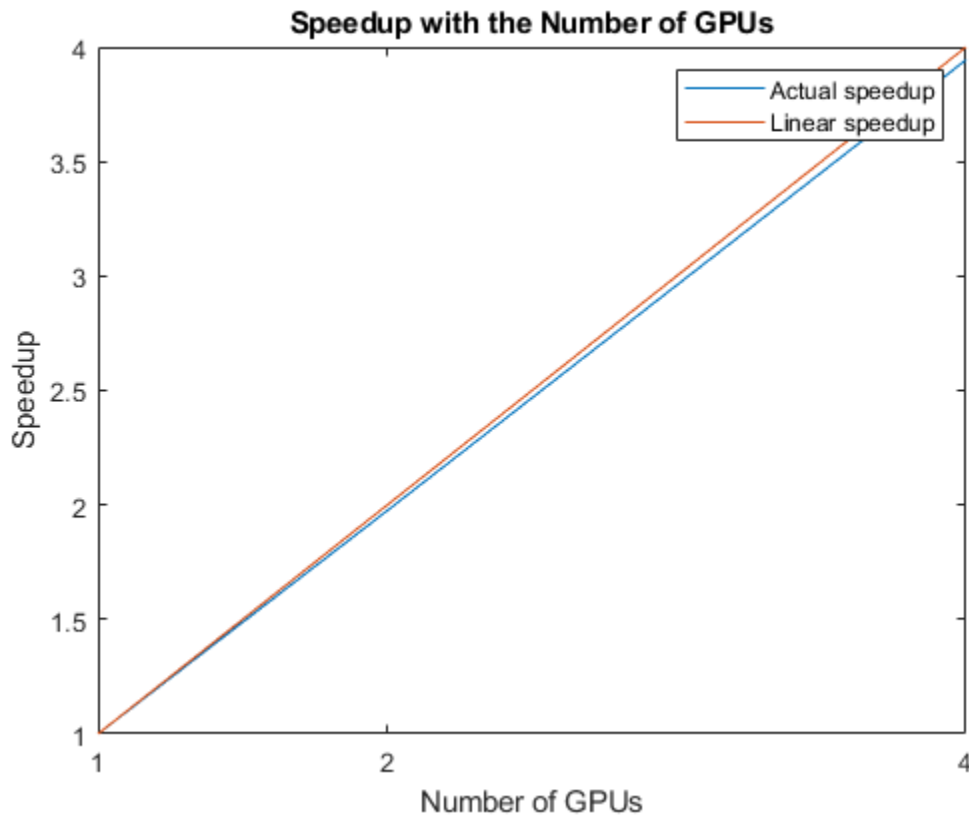
Define Helper Function

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
    layers = [
        convolution2dLayer(3,numFilters,'Padding','same')
        batchNormalizationLayer
        reluLayer];

    layers = repmat(layers,numConvLayers,1);
end
```

See Also
imageDatastore | trainNetwork | trainingOptions

Related Examples
- "Train Network in the Cloud Using Automatic Parallel Support" on page 9-66
- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

Process Big Data in the Cloud

This example shows how to access a large data set in the cloud and process it in a cloud cluster using MATLAB capabilities for big data.

Learn how to:

- Access a publicly available large data set on Amazon Cloud.
- Find and select an interesting subset of this data set.
- Use datastores, tall arrays, and Parallel Computing Toolbox to process this subset in less than 20 minutes.

The public data set in this example is part of the Wind Integration National Dataset Toolkit, or WIND Toolkit [1], [2], [3], [4]. For more information, see Wind Integration National Dataset Toolkit.

Requirements

To run this example, you must set up access to a cluster in Amazon AWS. In MATLAB, you can create clusters in Amazon AWS directly from the MATLAB desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters in Amazon AWS. For more information, see Getting Started with Cloud Center.

Set Up Access to Remote Data

The data set used in this example is the Techno-Economic WIND Toolkit. It contains 2 TB (terabyte) of data for wind power estimates and forecasts along with atmospheric variables from 2007 to 2013 within the continental U.S.

The Techno-Economic WIND Toolkit is available via Amazon Web Services, in the location *s3://pywtk-data.* It contains two data sets:

- *s3://pywtk-data/met_data* - Metrology Data
- *s3://pywtk-data/fcst_data* - Forecast Data

To work with remote data in Amazon S3, you must define environment variables for your AWS credentials. For more information on setting up access to remote data, see "Work with Remote Data". In the following code, replace YOUR_AWS_ACCESS_KEY_ID and YOUR_AWS_SECRET_ACCESS_KEY with your own Amazon AWS credentials.

```
setenv("AWS_ACCESS_KEY_ID","YOUR_AWS_ACCESS_KEY_ID");
setenv("AWS_SECRET_ACCESS_KEY","YOUR_AWS_SECRET_ACCESS_KEY");
```

This data set requires you to specify its geographic region, and so you must set the corresponding environment variable.

```
setenv("AWS_DEFAULT_REGION","us-west-2");
```

To give the workers in your cluster access to the remote data, add these environment variable names to the EnvironmentVariables property of your cluster profile. To edit the properties of your cluster profile, use the Cluster Profile Manager, in **Parallel > Create and Manage Clusters**.

Find Subset of Big Data

The 2 TB data set is quite large. This example shows you how to find a subset of the data set that you want to analyze. The example focuses on data for the state of Massachusetts.

First obtain the IDs that identify the metrological stations in Massachusetts, and determine the files that contain their metrological information. Metadata information for each station is in a file named `three_tier_site_metadata.csv`. Because this data is small and fits in memory, you can access it from the MATLAB client with `readtable`. You can use the `readtable` function to access open data in S3 buckets directly without needing to write special code.

```
tMetadata = readtable("s3://pywtk-data/three_tier_site_metadata.csv",...
    "ReadVariableNames",true,"TextType","string");
```

To find out which states are listed in this data set, use `unique`.

```
states = unique(tMetadata.state)
```

```
states = 50×1 string array
    ""
    "Alabama"
    "Arizona"
    "Arkansas"
    "California"
    "Colorado"
    "Connecticut"
    "Delaware"
    "District of Columbia"
    "Florida"
    "Georgia"
    "Idaho"
    "Illinois"
    "Indiana"
    "Iowa"
    "Kansas"
    "Kentucky"
    "Louisiana"
    "Maine"
    "Maryland"
    "Massachusetts"
    "Michigan"
    "Minnesota"
    "Mississippi"
    "Missouri"
    "Montana"
    "Nebraska"
    "Nevada"
    "New Hampshire"
    "New Jersey"
    "New Mexico"
    "New York"
    "North Carolina"
    "North Dakota"
    "Ohio"
    "Oklahoma"
    "Oregon"
    "Pennsylvania"
    "Rhode Island"
```

```
"South Carolina"
"South Dakota"
"Tennessee"
"Texas"
"Utah"
"Vermont"
"Virginia"
"Washington"
"West Virginia"
"Wisconsin"
"Wyoming"
```

Identify which stations are located in the state of Massachusetts.

```
index = tMetadata.state == "Massachusetts";
siteId = tMetadata{index,"site_id"};
```

The data for a given station is contained in a file that follows this naming convention: `s3://pywtk-data/met_data/folder/site_id.nc`, where `folder` is the nearest integer less than or equal to `site_id/500`. Using this convention, compose a file location for each station.

```
folder = floor(siteId/500);
fileLocations = compose("s3://pywtk-data/met_data/%d/%d.nc",folder,siteId);
```

Process Big Data

You can use datastores and tall arrays to access and process data that does not fit in memory. When performing big data computations, MATLAB accesses smaller portions of the remote data as needed, so you do not need to download the entire data set at once. With tall arrays, MATLAB automatically breaks the data into smaller blocks that fit in memory for processing.

If you have Parallel Computing Toolbox, MATLAB can process the many blocks in parallel. The parallelization enables you to run an analysis on a single desktop with local workers, or scale up to a cluster for more resources. When you use a cluster in the same cloud service as the data, the data stays in the cloud and you benefit from improved data transfer times. Keeping the data in the cloud is also more cost-effective. This example ran in less than 20 minutes using 18 workers on a c4.8xlarge machine in Amazon AWS.

If you use a parallel pool in a cluster, MATLAB processes this data using workers in the cluster. Create a parallel pool in the cluster. In the following code, use the name of your cluster profile instead. Attach the script to the pool, because the parallel workers need to access a helper function in it.

```
p = parpool("myAWSCluster");
```

```
Starting parallel pool (parpool) using the 'myAWSCluster' profile ...
connected to 18 workers.
```

```
addAttachedFiles(p,mfilename("fullpath"));
```

Create a datastore with the metrology data for the stations in Massachusetts. The data is in the form of Network Common Data Form (NetCDF) files, and you must use a custom read function to interpret them. In this example, this function is named `ncReader` and reads the NetCDF data into timetables. You can explore its contents at the end of this script.

```
dsMetrology = fileDatastore(fileLocations,"ReadFcn",@ncReader,"UniformRead",true);
```

Create a tall timetable with the metrology data from the datastore.

```
ttMetrology = tall(dsMetrology)

ttMetrology =

  M×6 tall timetable

            Time             wind_speed   wind_direction   power    density   temperature
    _____     _____   _____   _____   _____   _____

    01-Jan-2007 00:00:00       5.905          189.35        3.3254   1.2374      269.74
    01-Jan-2007 00:05:00       5.8898         188.77        3.2988   1.2376      269.73
    01-Jan-2007 00:10:00       5.9447         187.85         3.396   1.2376      269.71
    01-Jan-2007 00:15:00       6.0362         187.05        3.5574   1.2376      269.68
    01-Jan-2007 00:20:00       6.1156         186.49        3.6973   1.2375      269.83
    01-Jan-2007 00:25:00       6.2133         185.71        3.8698   1.2376      270.03
    01-Jan-2007 00:30:00       6.3232         184.29        4.0812   1.2379      270.19
    01-Jan-2007 00:35:00       6.4331         182.51        4.3382   1.2382       270.3
             :                   :              :             :        :           :
             :                   :              :             :        :           :
```

Get the mean temperature per month using `groupsummary`, and sort the resulting tall table. For performance, MATLAB defers most tall operations until the data is needed. In this case, plotting the data triggers evaluation of deferred calculations.

```
meanTemperature = groupsummary(ttMetrology,"Time","month","mean","temperature");
meanTemperature = sortrows(meanTemperature);
```

Plot the results.

```
figure;
plot(meanTemperature.mean_temperature,"*-");
ylim([260 300]);
xlim([1 12*7+1]);
xticks(1:12:12*7+1);
xticklabels(["2007","2008","2009","2010","2011","2012","2013","2014"]);
title("Average Temperature in Massachusetts 2007-2013");
xlabel("Year");
ylabel("Temperature (K)")
```

Average Temperature in Massachusetts 2007-2013
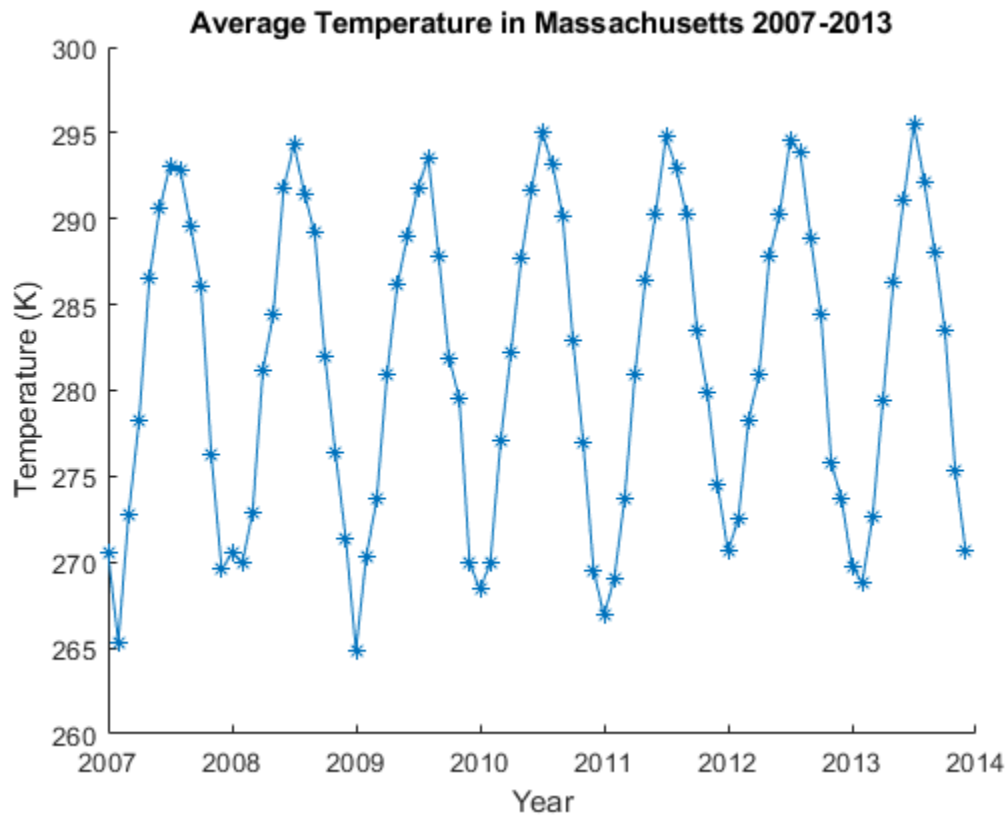
Many MATLAB functions support tall arrays, so you can perform a variety of calculations on big data sets using familiar syntax. For more information on supported functions, see "Supporting Functions".

Define Custom Read Function

The data in the Techno-Economic WIND Toolkit is saved in NetCDF files. Define a custom read function to read its data into a timetable. For more information on reading NetCDF files, see "NetCDF Files".

```
function t = ncReader(filename)
% NCREADER Read NetCDF File (.nc), extract data set and save as a timetable

% Get information about NetCDF data source
fileInfo = ncinfo(filename);

% Extract variable names and datatypes
varNames = string({fileInfo.Variables.Name});
varTypes = string({fileInfo.Variables.Datatype});

% Transform variable names into valid names for table variables
if any(startsWith(varNames,["4","6"]))
    strVarNames = replace(varNames,["4","6"],["four","six"]);
else
    strVarNames = varNames;
end

% Extract the length of each variable
```

```
fileLength = fileInfo.Dimensions.Length;

% Extract initial timestamp, sample period and create the time axis
tAttributes = struct2table(fileInfo.Attributes);
startTime = datetime(cell2mat(tAttributes.Value(contains(tAttributes.Name,"start_time"))),"Conver
samplePeriod = seconds(cell2mat(tAttributes.Value(contains(tAttributes.Name,"sample_period"))));

% Create the output timetable
numVars = numel(strVarNames);
tableSize = [fileLength numVars];
t = timetable('Size',tableSize,'VariableTypes',varTypes,'VariableNames',strVarNames,'TimeStep',sa

% Fill in the timetable with variable data
for k = 1:numVars
    t(:,k) = table(ncread(filename,varNames{k}));
end
end
```

References

[1] Draxl, C., B. M. Hodge, A. Clifton, and J. McCaa. *Overview and Meteorological Validation of the Wind Integration National Dataset Toolkit* (Technical Report, NREL/TP-5000-61740). Golden, CO: National Renewable Energy Laboratory, 2015.

[2] Draxl, C., B. M. Hodge, A. Clifton, and J. McCaa. "The Wind Integration National Dataset (WIND) Toolkit." *Applied Energy*. Vol. 151, 2015, pp. 355-366.

[3] King, J., A. Clifton, and B. M. Hodge. *Validation of Power Output for the WIND Toolkit* (Technical Report, NREL/TP-5D00-61714). Golden, CO: National Renewable Energy Laboratory, 2014.

[4] Lieberman-Cribbin, W., C. Draxl, and A. Clifton. *Guide to Using the WIND Toolkit Validation Code* (Technical Report, NREL/TP-5000-62595). Golden, CO: National Renewable Energy Laboratory, 2014.

See Also
datastore | parpool | readtable | tall

Related Examples
- "Work with Remote Data"
- "Discover Clusters and Use Cluster Profiles" on page 5-11

More About
- "Upload Deep Learning Data to the Cloud" on page 9-77
- "Deep Learning with Big Data on GPUs and in Parallel" (Deep Learning Toolbox)

Use parfeval to Train Multiple Deep Learning Networks

This example shows how to use `parfeval` to perform a parameter sweep on the depth of the network architecture for a deep learning network and retrieve data during training.

Deep learning training often takes hours or days, and searching for good architectures can be difficult. With parallel computing, you can speed up and automate your search for good models. If you have access to a machine with multiple graphical processing units (GPUs), you can complete this example on a local copy of the data set with a local parallel pool. If you want to use more resources, you can scale up deep learning training to the cloud. This example shows how to use `parfeval` to perform a parameter sweep on the depth of a network architecture in a cluster in the cloud. Using `parfeval` allows you to train in the background without blocking MATLAB, and provides options to stop early if results are satisfactory. You can modify the script to do a parameter sweep on any other parameter. Also, this example shows how to obtain feedback from the workers during computation by using `DataQueue`.

Requirements

Before you can run this example, you need to configure a cluster and upload your data to the Cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. For this example, ensure that your cluster is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. After that, upload your data to an Amazon S3 bucket and use it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

Load Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. Split the training data set into training and validation sets, and keep the test data set to test the best network from the parameter sweep. In this example, you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

```
imds = imageDatastore('s3://cifar10cloud/cifar10/train', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
```

Train the network with augmented image data by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
```

```
        'RandXTranslation',pixelRange, ...
        'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
        'DataAugmentation',imageAugmenter, ...
        'OutputSizeMode','randcrop');
```

Train Several Networks Simultaneously

Define the training options. Set the mini-batch size and scale the initial learning rate linearly according to the mini-batch size. Set the validation frequency so that `trainNetwork` validates the network once per epoch.

```
miniBatchSize = 128;
initialLearnRate = 1e-1 * miniBatchSize/256;
validationFrequency = floor(numel(imdsTrain.Labels)/miniBatchSize);
options = trainingOptions('sgdm', ...
        'MiniBatchSize',miniBatchSize, ... % Set the mini-batch size
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
        'Shuffle','every-epoch', ...
        'ValidationData',imdsValidation, ...
        'ValidationFrequency', validationFrequency);
```

Specify the depths for the network architecture on which to do a parameter sweep. Perform a parallel parameter sweep training several networks simultaneously using `parfeval`. Use a loop to iterate through the different network architectures in the sweep. Create the helper function `createNetworkArchitecture` at the end of the script, which takes an input argument to control the depth of the network and creates an architecture for CIFAR-10. Use `parfeval` to offload the computations performed by `trainNetwork` to a worker in the cluster. `parfeval` returns a future variable to hold the trained networks and training information when computations are done.

```
netDepths = 1:4;
for idx = 1:numel(netDepths)
    networksFuture(idx) = parfeval(@trainNetwork,2, ...
        augmentedImdsTrain,createNetworkArchitecture(netDepths(idx)),options);
end
```

```
Starting parallel pool (parpool) using the 'MyCluster' profile ...
Connected to the parallel pool (number of workers: 4).
```

`parfeval` does not block MATLAB, which means you can continue executing commands. In this case, obtain the trained networks and their training information by using `fetchOutputs` on `networksFuture`. The `fetchOutputs` function waits until the future variables finish.

```
[trainedNetworks,trainingInfo] = fetchOutputs(networksFuture);
```

Obtain the final validation accuracies of the networks by accessing the `trainingInfo` structure.

```
accuracies = [trainingInfo.FinalValidationAccuracy]
```

```
accuracies = 1×4

   72.5600   77.2600   79.4000   78.6800
```

Select the best network in terms of accuracy. Test its performance against the test data set.

```matlab
[~, I] = max(accuracies);
bestNetwork = trainedNetworks(I(1));
YPredicted = classify(bestNetwork,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

accuracy = 0.7840

Calculate the confusion matrix for the test data.

```matlab
figure('Units','normalized','Position',[0.2 0.2 0.4 0.4]);
confusionchart(imdsTest.Labels,YPredicted,'RowSummary','row-normalized','ColumnSummary','column-
```

True Class	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck		
airplane	861	19	19	8	14	2	11	12	23	31	86.1%	13.9%
automobile	8	922	2	1			4	2	6	55	92.2%	7.8%
bird	93	4	618	28	67	40	111	24	3	12	61.8%	38.2%
cat	36	7	41	571	55	104	116	43	9	18	57.1%	42.9%
deer	15	3	36	25	754	7	118	35	5	2	75.4%	24.6%
dog	11	3	37	102	48	649	61	73	6	10	64.9%	35.1%
frog	9	1	24	21	14	7	914	3	3	4	91.4%	8.6%
horse	28	6	11	15	51	28	22	830	1	8	83.0%	17.0%
ship	96	26	7	6	4	1	8	3	827	22	82.7%	17.3%
truck	17	62	2	7			6	2	10	894	89.4%	10.6%
	73.3%	87.6%	77.5%	72.8%	74.9%	77.4%	66.7%	80.8%	92.6%	84.7%		
	26.7%	12.4%	22.5%	27.2%	25.1%	22.6%	33.3%	19.2%	7.4%	15.3%		
	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck		

Predicted Class

Send Feedback Data During Training

Prepare and initialize plots that show the training progress in each of the workers. Use
`animatedLine` for a convenient way to show changing data.

```matlab
f = figure;
f.Visible = true;
for i=1:4
    subplot(2,2,i)
    xlabel('Iteration');
    ylabel('Training accuracy');
    lines(i) = animatedline;
end
```
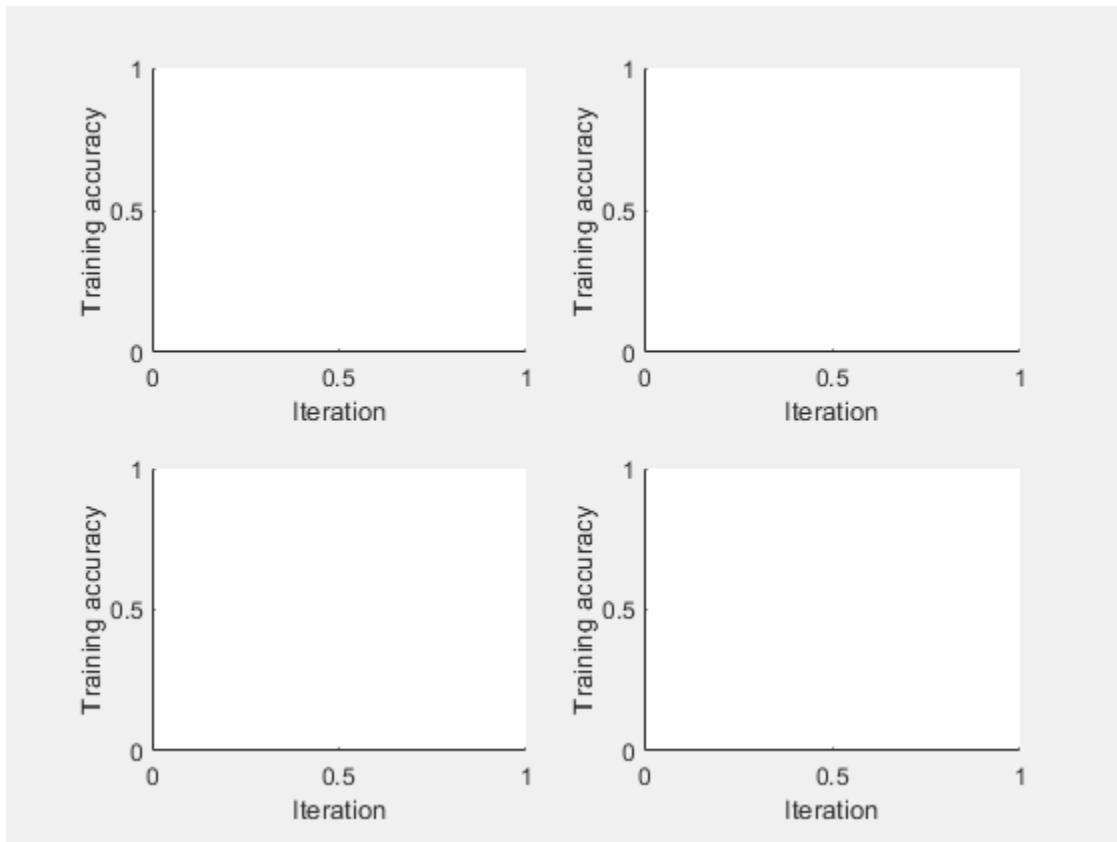
Send the training progress data from the workers to the client by using `DataQueue`, and then plot the data. Update the plots each time the workers send training progress feedback by using `afterEach`. The parameter `opts` contains information about the worker, training iteration, and training accuracy.

```
D = parallel.pool.DataQueue;
afterEach(D, @(opts) updatePlot(lines, opts{:}));
```

Specify the depths for the network architecture on which to do a parameter sweep, and perform the parallel parameter sweep using `parfeval`. Allow the workers to access any helper function in this script, by adding the script to the current pool as an attached file. Define an output function in the training options to send the training progress from the workers to the client. The training options depend on the index of the worker and must be included inside the `for` loop.

```
netDepths = 1:4;
addAttachedFiles(gcp,mfilename);
for idx = 1:numel(netDepths)

    miniBatchSize = 128;
    initialLearnRate = 1e-1 * miniBatchSize/256; % Scale the learning rate according to the mini
    validationFrequency = floor(numel(imdsTrain.Labels)/miniBatchSize);

    options = trainingOptions('sgdm', ...
        'OutputFcn',@(state) sendTrainingProgress(D,idx,state), ... % Set an output function to s
        'MiniBatchSize',miniBatchSize, ... % Set the corresponding MiniBatchSize in the sweep.
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
```

```
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
        'Shuffle','every-epoch', ...
        'ValidationData',imdsValidation, ...
        'ValidationFrequency', validationFrequency);

    networksFuture(idx) = parfeval(@trainNetwork,2, ...
        augmentedImdsTrain,createNetworkArchitecture(netDepths(idx)),options);
end
```

`parfeval` invokes `trainNetwork` on a worker in the cluster. Computations happen on the background, so you can continue working in MATLAB. If you want to stop a `parfeval` computation, you can call `cancel` on its corresponding future variable. For example, if you observe that a network is underperforming, you can cancel its future. When you do so, the next queued future variable starts its computations.

In this case, fetch the trained networks and their training information by invoking `fetchOutputs` on the future variables.

```
[trainedNetworks,trainingInfo] = fetchOutputs(networksFuture);
```
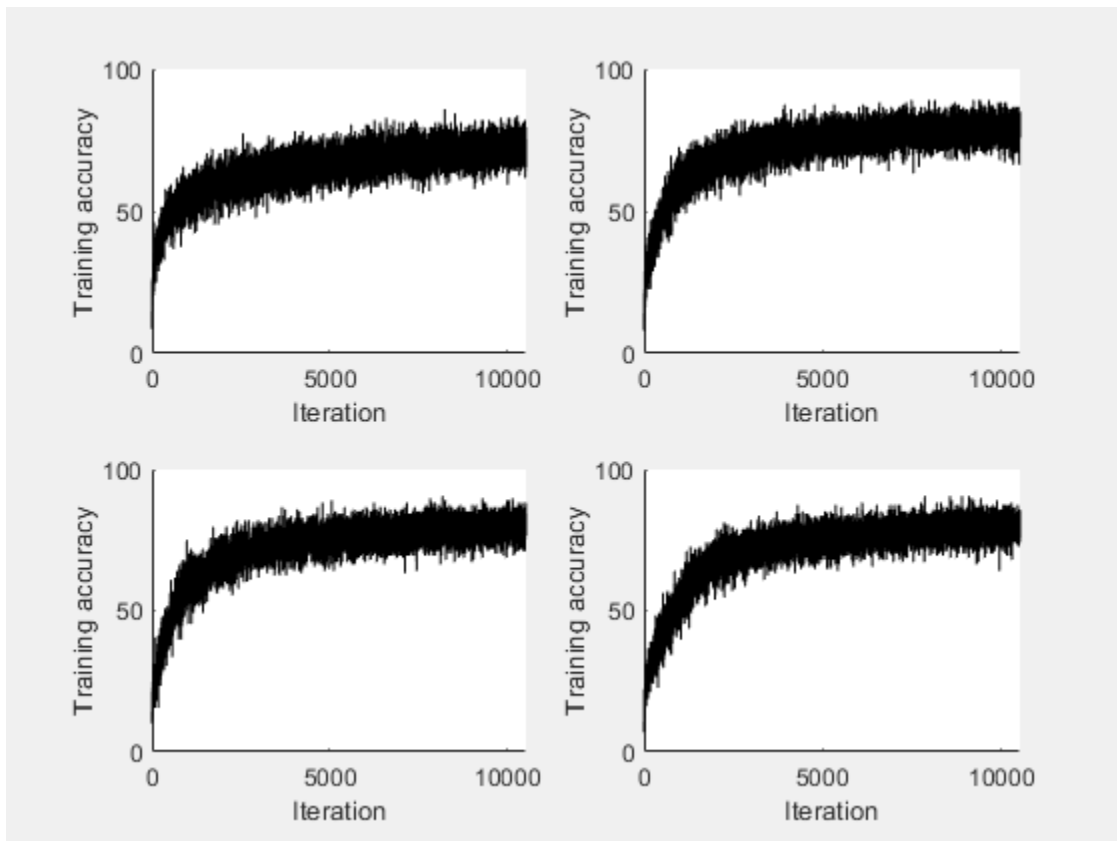

Obtain the final validation accuracy for each network.

```
accuracies = [trainingInfo.FinalValidationAccuracy]
```

`accuracies = 1×4`

```
   72.9200    77.4800    76.9200    77.0400
```

Helper Functions

Define a network architecture for the CIFAR-10 data set with a function, and use an input argument to adjust the depth of the network. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
function layers = createNetworkArchitecture(netDepth)
imageSize = [32 32 3];
netWidth = round(16/sqrt(netDepth)); % netWidth controls the number of filters in a convolutional

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,netDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
    ];
end
```

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
layers = [
    convolution2dLayer(3,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer
    ];

layers = repmat(layers,numConvLayers,1);
end
```

Define a function to send the training progress to the client through `DataQueue`.

```
function sendTrainingProgress(D,idx,info)
if info.State == "iteration"
    send(D,{idx,info.Iteration,info.TrainingAccuracy});
end
end
```

Define an update function to update the plots when a worker sends an intermediate result.

```
function updatePlot(lines,idx,iter,acc)
addpoints(lines(idx),iter,acc);
drawnow limitrate nocallbacks
end
```

See Also
afterEach | imageDatastore | parfeval | trainNetwork | trainingOptions

Related Examples

- "Train Network in the Cloud Using Automatic Parallel Support" on page 9-66
- "Use parfor to Train Multiple Deep Learning Networks" on page 9-70
- "Upload Deep Learning Data to the Cloud" on page 9-77
- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

Train Network in the Cloud Using Automatic Parallel Support

This example shows how to train a convolutional neural network using MATLAB automatic support for parallel training. Deep learning training often takes hours or days. With parallel computing, you can speed up training using multiple graphical processing units (GPUs) locally or in a cluster in the cloud. If you have access to a machine with multiple GPUs, then you can complete this example on a local copy of the data. If you want to use more resources, then you can scale up deep learning training to the cloud. To learn more about your options for parallel training, see "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox). This example guides you through the steps to train a deep learning network in a cluster in the cloud using MATLAB automatic parallel support.

Requirements

Before you can run the example, you need to configure a cluster and upload data to the cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. After that, upload your data to an Amazon S3 bucket and access it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

Set Up Parallel Pool

Start a parallel pool in the cluster and set the number of workers to the number of GPUs in your cluster. If you specify more workers than GPUs, then the remaining workers are idle. This example assumes that the cluster you are using is set as the default cluster profile. Check the default cluster profile on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**.

```
numberOfWorkers = 8;
parpool(numberOfWorkers);

Starting parallel pool (parpool) using the 'MyClusterInTheCloud' profile ...
connected to 8 workers.
```

Load Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. In this example, you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

```
imdsTrain = imageDatastore('s3://cifar10cloud/cifar10/train', ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');
```

Train the network with augmented image data by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```matlab
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Define Network Architecture and Training Options

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```matlab
blockDepth = 4; % blockDepth controls the depth of a convolutional block
netWidth = 32; % netWidth controls the number of filters in a convolutional block

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,blockDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
];
```

Define the training options. Train the network in parallel using the current cluster, by setting the execution environment to `parallel`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU constant. Scale the learning rate according to the mini-batch size. Use a learning rate schedule to drop the learning rate as the training progresses. Turn on the training progress plot to obtain visual feedback during training.

```matlab
miniBatchSize = 256 * numberOfWorkers;
initialLearnRate = 1e-1 * miniBatchSize/256;

options = trainingOptions('sgdm', ...
    'ExecutionEnvironment','parallel', ... % Turn on automatic parallel support.
    'InitialLearnRate',initialLearnRate, ... % Set the initial learning rate.
    'MiniBatchSize',miniBatchSize, ... % Set the MiniBatchSize.
    'Verbose',false, ... % Do not send command line output.
    'Plots','training-progress', ... % Turn on the training progress plot.
    'L2Regularization',1e-10, ...
    'MaxEpochs',50, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',floor(numel(imdsTrain.Files)/miniBatchSize), ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',45);
```

Train Network and Use for Classification

Train the network in the cluster. During training, the plot displays the progress.

```
net = trainNetwork(augmentedImdsTrain,layers,options)
```



```
net =
  SeriesNetwork with properties:

    Layers: [43×1 nnet.cnn.layer.Layer]
```
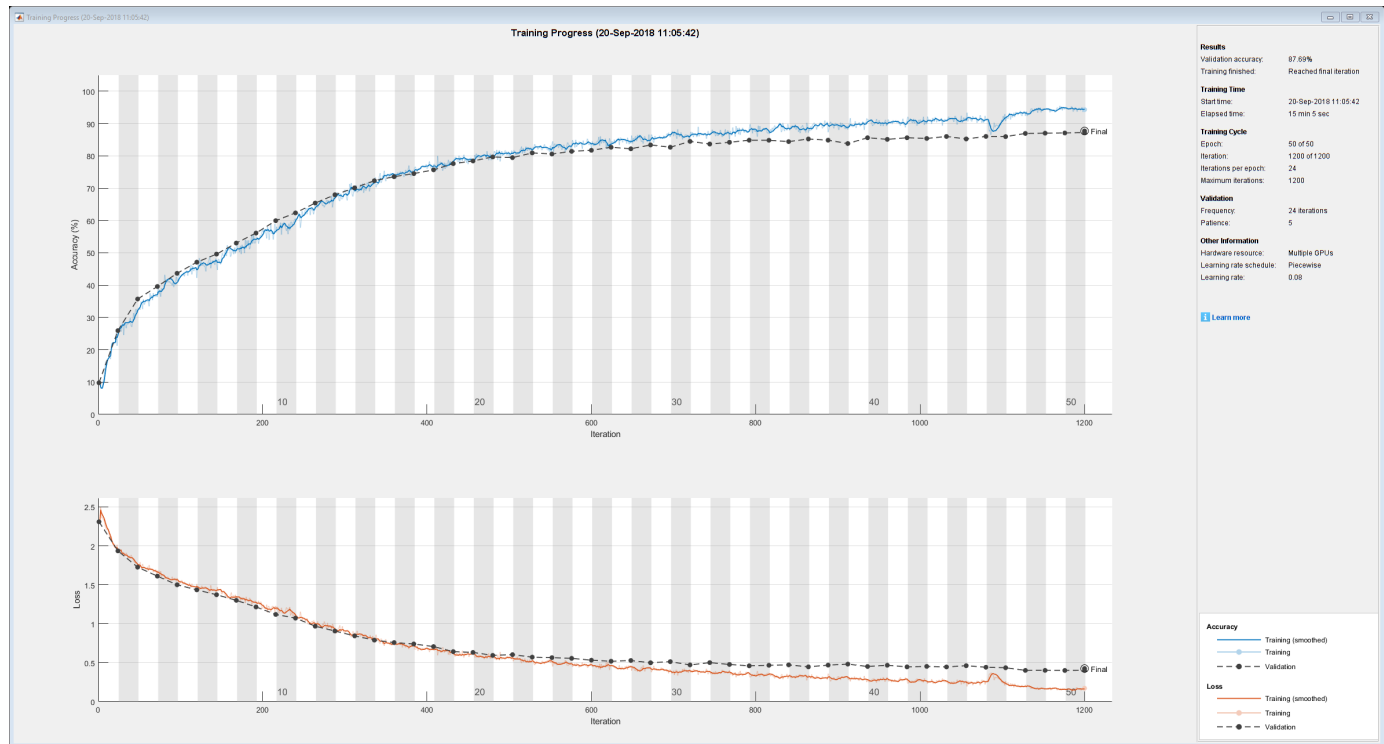
Determine the accuracy of the network, by using the trained network to classify the test images on your local machine. Then compare the predicted labels to the actual labels.

```
YPredicted = classify(net,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

Define Helper Function

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
    layers = [
        convolution2dLayer(3,numFilters,'Padding','same')
        batchNormalizationLayer
        reluLayer
    ];
```

```
    layers = repmat(layers,numConvLayers,1);
end
```

See Also
imageDatastore | trainNetwork | trainingOptions

Related Examples
- "Upload Deep Learning Data to the Cloud" on page 9-77
- "Use parfor to Train Multiple Deep Learning Networks" on page 9-70
- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

Use parfor to Train Multiple Deep Learning Networks

This example shows how to use a `parfor` loop to perform a parameter sweep on a training option.

Deep learning training often takes hours or days, and searching for good training options can be difficult. With parallel computing, you can speed up and automate your search for good models. If you have access to a machine with multiple graphical processing units (GPUs), you can complete this example on a local copy of the data set with a local parpool. If you want to use more resources, you can scale up deep learning training to the cloud. This example shows how to use a parfor loop to perform a parameter sweep on the training option `MiniBatchSize` in a cluster in the cloud. You can modify the script to do a parameter sweep on any other training option. Also, this example shows how to obtain feedback from the workers during computation using `DataQueue`. You can also send the script as a batch job to the cluster, so you can continue working or close MATLAB and fetch the results later. For more information, see "Send Deep Learning Batch Job to Cluster" (Deep Learning Toolbox).

Requirements

Before you can run this example, you need to configure a cluster and upload your data to the cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. For this example, ensure that your cluster is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. After that, upload your data to an Amazon S3 bucket and use it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

Load the Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. Split the training data set into training and validation sets, and keep the test data set to test the best network from the parameter sweep. In this example you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

```
imds = imageDatastore('s3://cifar10cloud/cifar10/train', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
```

Train the network with augmented image data, by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
```

```
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Define Network Architecture

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
imageSize = [32 32 3];
netDepth = 2; % netDepth controls the depth of a convolutional block
netWidth = 16; % netWidth controls the number of filters in a convolutional block

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,netDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
    ];
```

Train Several Networks Simultaneously

Specify the mini-batch sizes on which to do a parameter sweep. Allocate variables for the resulting networks and accuracy.

```
miniBatchSizes = [64 128 256 512];
numMiniBatchSizes = numel(miniBatchSizes);
trainedNetworks = cell(numMiniBatchSizes,1);
accuracies = zeros(numMiniBatchSizes,1);
```

Perform a parallel parameter sweep training several networks inside a `parfor` loop and varying the mini-batch size. The workers in the cluster train the networks simultaneously and send the trained networks and accuracies back when the training is complete. If you want to check that the training is working, set `Verbose` to `true` in the training options. Note that the workers compute independently, so the command line output is not in the same sequential order as the iterations.

```
parfor idx = 1:numMiniBatchSizes

    miniBatchSize = miniBatchSizes(idx);
    initialLearnRate = 1e-1 * miniBatchSize/256; % Scale the learning rate according to the mini

    % Define the training options. Set the mini-batch size.
    options = trainingOptions('sgdm', ...
        'MiniBatchSize',miniBatchSize, ... % Set the corresponding MiniBatchSize in the sweep.
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
```

```
        'Shuffle','every-epoch', ...
        'ValidationData',imdsValidation, ...
        'LearnRateSchedule','piecewise', ...
        'LearnRateDropFactor',0.1, ...
        'LearnRateDropPeriod',25);

    % Train the network in a worker in the cluster.
    net = trainNetwork(augmentedImdsTrain,layers,options);

    % To obtain the accuracy of this network, use the trained network to
    % classify the validation images on the worker and compare the predicted labels to the
    % actual labels.
    YPredicted = classify(net,imdsValidation);
    accuracies(idx) = sum(YPredicted == imdsValidation.Labels)/numel(imdsValidation.Labels);

    % Send the trained network back to the client.
    trainedNetworks{idx} = net;
end
```

```
Starting parallel pool (parpool) using the 'MyClusterInTheCloud' profile ...
Connected to the parallel pool (number of workers: 4).
```

After `parfor` finishes, `trainedNetworks` contains the resulting networks trained by the workers. Display the trained networks and their accuracies.

```
trainedNetworks
```

```
trainedNetworks = 4×1 cell array
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
```

```
accuracies
```

```
accuracies = 4×1

    0.8188
    0.8232
    0.8162
    0.8050
```

Select the best network in terms of accuracy. Test its performance against the test data set.

```
[~, I] = max(accuracies);
bestNetwork = trainedNetworks{I(1)};
YPredicted = classify(bestNetwork,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

```
accuracy = 0.8173
```

Send Feedback Data During Training

Prepare and initialize plots that show the training progress in each of the workers. Use `animatedLine` for a convenient way to show changing data.

```
f = figure;
f.Visible = true;
```

```
for i=1:4
    subplot(2,2,i)
    xlabel('Iteration');
    ylabel('Training accuracy');
    lines(i) = animatedline;
end
```
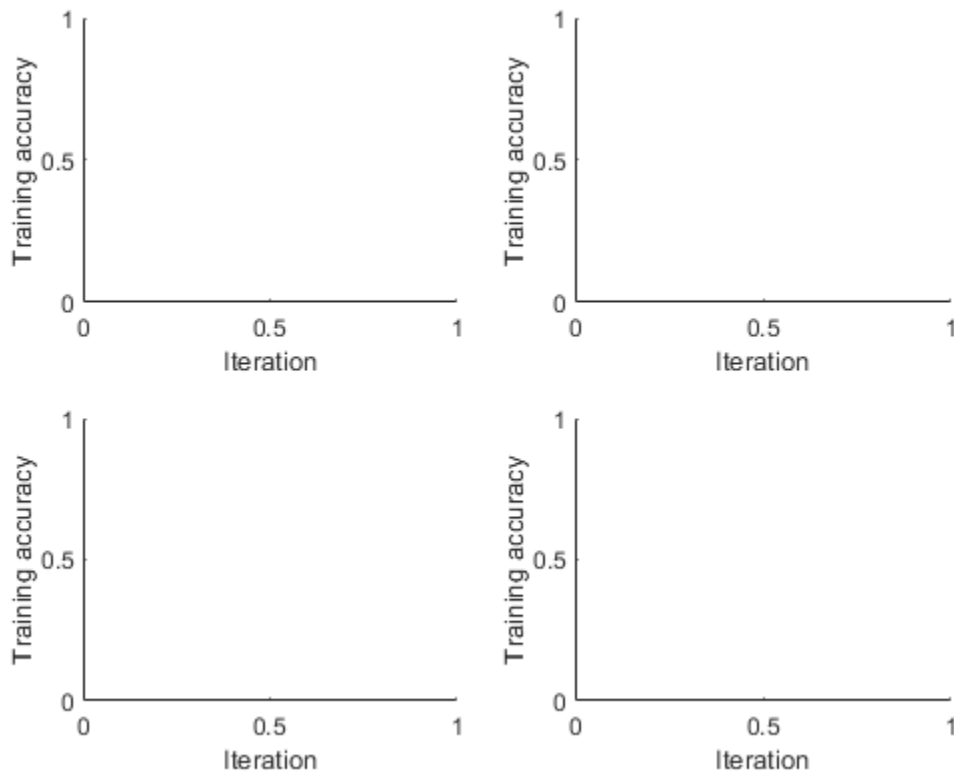
Send the training progress data from the workers to the client by using `DataQueue`, and then plot the data. Update the plots each time the workers send training progress feedback by using `afterEach`. The parameter `opts` contains information about the worker, training iteration, and training accuracy.

```
D = parallel.pool.DataQueue;
afterEach(D, @(opts) updatePlot(lines, opts{:}));
```

Perform a parallel parameter sweep training several networks inside a parfor loop with different mini-batch sizes. Note the use of `OutputFcn` in the training options to send the training progress to the client each iteration. This figure shows the training progress of four different workers during an execution of the following code.
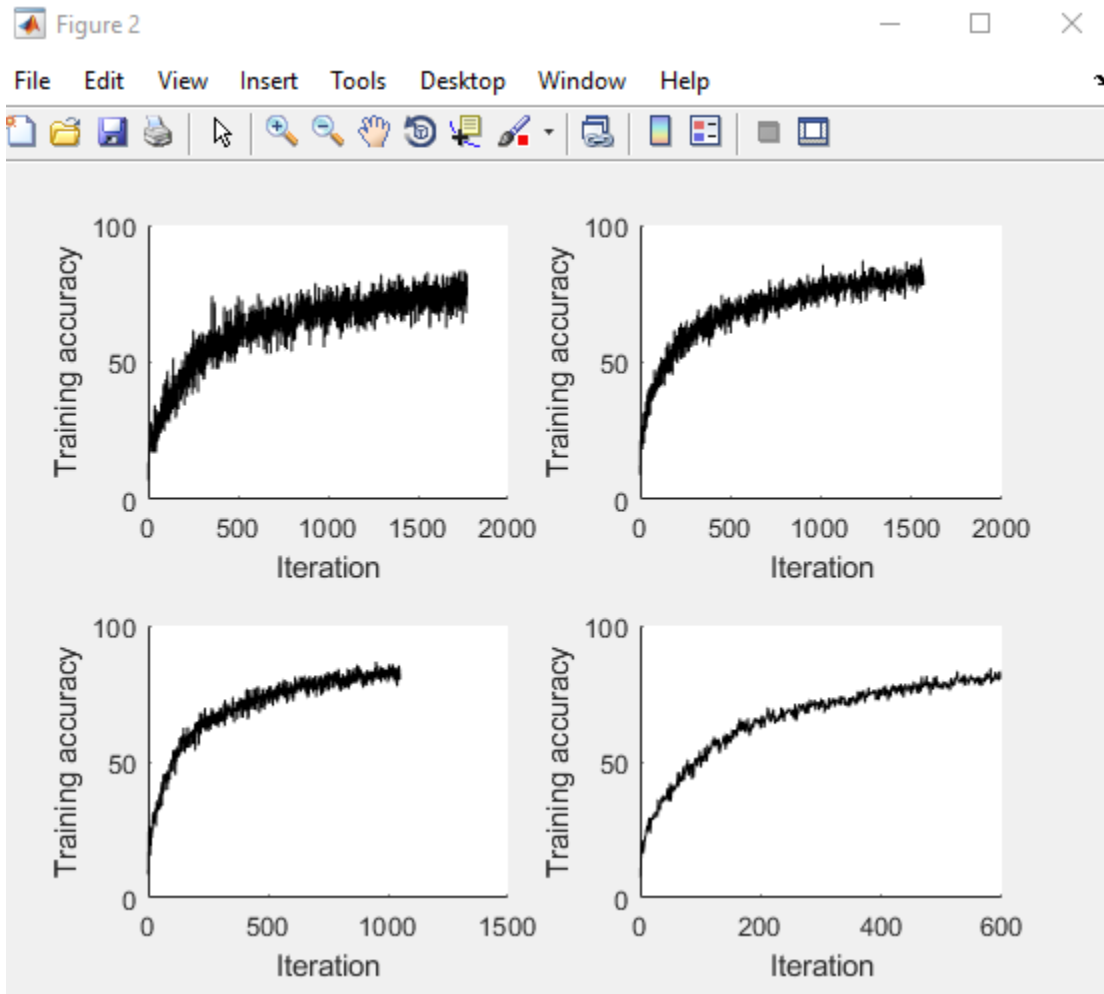
```matlab
parfor idx = 1:numel(miniBatchSizes)

    miniBatchSize = miniBatchSizes(idx);
    initialLearnRate = 1e-1 * miniBatchSize/256; % Scale the learning rate according to the miniB

    % Define the training options. Set an output function to send data back
    % to the client each iteration.
    options = trainingOptions('sgdm', ...
        'MiniBatchSize',miniBatchSize, ... % Set the corresponding MiniBatchSize in the sweep.
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
        'OutputFcn',@(state) sendTrainingProgress(D,idx,state), ... % Set an output function to s
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
        'Shuffle','every-epoch', ...
        'ValidationData',imdsValidation, ...
        'LearnRateSchedule','piecewise', ...
        'LearnRateDropFactor',0.1, ...
        'LearnRateDropPeriod',25);

    % Train the network in a worker in the cluster. The workers send
    % training progress information during training to the client.
```

```
    net = trainNetwork(augmentedImdsTrain,layers,options);

    % To obtain the accuracy of this network, use the trained network to
    % classify the validation images on the worker and compare the predicted labels to the
    % actual labels.
    YPredicted = classify(net,imdsValidation);
    accuracies(idx) = sum(YPredicted == imdsValidation.Labels)/numel(imdsValidation.Labels);

    % Send the trained network back to the client.
    trainedNetworks{idx} = net;
end
```

```
Analyzing and transferring files to the workers ...done.
```

After `parfor` finishes, `trainedNetworks` contains the resulting networks trained by the workers. Display the trained networks and their accuracies.

```
trainedNetworks
```

```
trainedNetworks = 4×1 cell array
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
```

```
accuracies
```

```
accuracies = 4×1

    0.8214
    0.8172
    0.8132
    0.8084
```

Select the best network in terms of accuracy. Test its performance against the test data set.

```
[~, I] = max(accuracies);
bestNetwork = trainedNetworks{I(1)};
YPredicted = classify(bestNetwork,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

```
accuracy = 0.8187
```

Helper Functions

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
layers = [
    convolution2dLayer(3,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer
    ];

layers = repmat(layers,numConvLayers,1);
end
```

Define a function to send the training progress to the client through `DataQueue`.

```
function sendTrainingProgress(D,idx,info)
if info.State == "iteration"
    send(D,{idx,info.Iteration,info.TrainingAccuracy});
end
end
```

Define an update function to update the plots when a worker sends an intermediate result.

```
function updatePlot(lines,idx,iter,acc)
addpoints(lines(idx),iter,acc);
drawnow limitrate nocallbacks
end
```

See Also
`imageDatastore` | `parallel.pool.DataQueue` | `trainNetwork`

Related Examples

- "Upload Deep Learning Data to the Cloud" on page 9-77
- "Send Deep Learning Batch Job to Cluster" on page 9-79
- "Use parfeval to Train Multiple Deep Learning Networks" on page 9-59
- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

More About

- "Parallel for-Loops (parfor)"

Upload Deep Learning Data to the Cloud

This example shows how to upload data to an Amazon S3 bucket.

Before you can perform deep learning training in the cloud, you need to upload your data to the cloud. The example shows how to download the CIFAR-10 data set to your computer, and then upload the data to an Amazon S3 bucket for later use in MATLAB. The CIFAR-10 data set is a labeled image data set commonly used for benchmarking image classification algorithms. Before running this example, you need access to an Amazon Web Services (AWS) account. After you upload the data set to Amazon S3, you can try any of the examples in "Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox).

Download CIFAR-10 to Local Machine

Specify a local directory in which to download the data set. The following code creates a folder in your current directory containing all the images in the data set.

```
directory = pwd;
[trainDirectory,testDirectory] = downloadCIFARToFolders(directory);

Downloading CIFAR-10 data set...done.
Copying CIFAR-10 to folders...done.
```

Upload Local Data Set to Amazon S3 Bucket

To work with data in the cloud, you can upload to Amazon S3 and then use datastores to access the data in S3 from the workers in your cluster. The following steps describe how to upload the CIFAR-10 data set from your local machine to an Amazon S3 bucket.

1. For efficient file transfers to and from Amazon S3, download and install the AWS Command Line Interface tool from `https://aws.amazon.com/cli/`.

2. Specify your AWS Access Key ID, Secret Access Key, and Region of the bucket as system environment variables. Contact your AWS account administrator to obtain your keys.

For example, on Linux, macOS, or Unix, specify these variables:

```
export AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
export AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
export AWS_DEFAULT_REGION="us-east-1"
```

On Windows, specify these variables:

```
set AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
set AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
set AWS_DEFAULT_REGION="us-east-1"
```

To specify these environment variables permanently, set them in your user or system environment.

3. Create a bucket for your data by using either the AWS S3 web page or a command such as the following:

```
aws s3 mb s3://mynewbucket
```

4. Upload your data using a command such as the following:

```
aws s3 cp mylocaldatapath s3://mynewbucket --recursive
```

For example:

```
aws s3 cp path/to/CIFAR10/in/the/local/machine s3://MyExampleCloudData/cifar10/ --recursive
```

5. Copy your AWS credentials to your cluster workers by completing these steps in MATLAB:

a. In the **Environment** section on the **Home** tab, select **Parallel > Create and Manage Clusters**.

b. In the **Cluster Profile** pane of the Cluster Profile Manager, select your cloud cluster profile.

c. In the **Properties** tab, select the **EnvironmentVariables** property, scrolling as necessary to find the property.

d. At the bottom right of the window, click **Edit**.

e. Click in the box to the right of **EnvironmentVariables**, and then type these three variables, each on its own line: AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_DEFAULT_REGION.

f. At the bottom right of the window, click **Done**.

For information on how to create a cloud cluster, see "Create Cloud Cluster" on page 5-14.

Use Data Set in MATLAB

After you store your data in Amazon S3, you can use datastores to access the data from your cluster workers. Simply create a datastore pointing to the URL of the S3 bucket. The following sample code shows how to use an `imageDatastore` to access an S3 bucket. Replace `'s3://MyExampleCloudData/cifar10/train'` with the URL of your S3 bucket.

```
imds = imageDatastore('s3://MyExampleCloudData/cifar10/train', ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');
```

With the CIFAR-10 data set now stored in Amazon S3, you can try any of the examples in "Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox) that show how to use CIFAR-10 in different use cases.

See Also
`imageDatastore`

Related Examples
- "Train Network in the Cloud Using Automatic Parallel Support" on page 9-66
- "Use parfor to Train Multiple Deep Learning Networks" on page 9-70
- "Use parfeval to Train Multiple Deep Learning Networks" on page 9-59
- "Send Deep Learning Batch Job to Cluster" on page 9-79
- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)
- "Work with Remote Data"

Send Deep Learning Batch Job to Cluster

This example shows how to send deep learning training batch jobs to a cluster so that you can continue working or close MATLAB during training.

Training deep neural networks often takes hours or days. To use time efficiently, you can train neural networks as batch jobs and fetch the results from the cluster when they are ready. You can continue working in MATLAB while computations take place or close MATLAB and obtain the results later using the Job Monitor. This example sends the parallel parameter sweep in "Use parfor to Train Multiple Deep Learning Networks" (Deep Learning Toolbox) as a batch job. After the job is complete, you can fetch the trained networks and compare their accuracies.

Requirements

Before you can run this example, you need to configure a cluster and upload your data to the Cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. For this example, ensure that your cluster is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. After that, upload your data to an Amazon S3 bucket and use it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see "Upload Deep Learning Data to the Cloud" (Deep Learning Toolbox).

Submit Batch Job

Send a script as a batch job to the cluster by using the `batch` function. The cluster allocates one worker to execute the contents of your script. If the parallel code in the script benefits from extra workers, for example, it includes automatic parallel support or a parfor loop, you need to request the workers explicitly. `batch` uses one worker for the client running the script. You can specify more workers by using the 'Pool' name-value pair argument.

In this case, send the `trainMultipleNetworks` script to the cluster. This script contains the parallel parameter sweep in "Use parfor to Train Multiple Deep Learning Networks" (Deep Learning Toolbox). Because the script contains a `parfor` loop, specify 4 extra workers with the `Pool` name-value pair argument.

```
totalNumberOfWorkers = 5;
job1 = batch('trainMultipleNetworks', ...
    'Pool',totalNumberOfWorkers-1);
```

You can see the current status of your job in the cluster by checking the Job Monitor. In the **Environment** section on the **Home** tab, select **Parallel > Monitor Jobs** to open the Job Monitor.
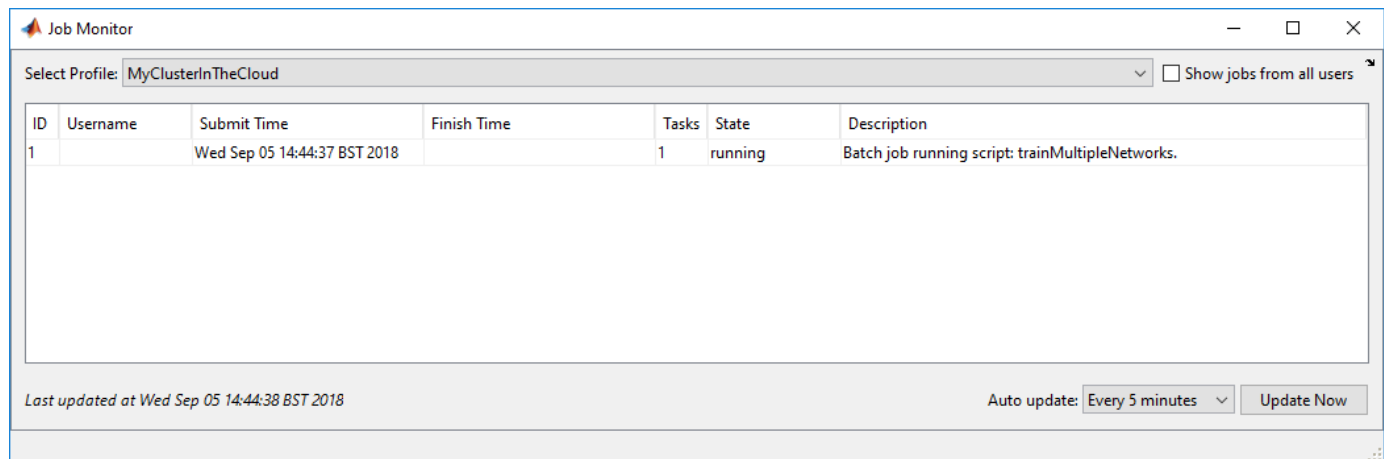
You can submit additional jobs to the cluster. If the cluster is not available because it is running other jobs, any new job you submit remains queued until the cluster becomes available.

Fetch Results Programmatically

After submitting jobs to the cluster, you can continue working in MATLAB while computations take place. If the rest of your code depends on completion of a job, block MATLAB by using the `wait` command. In this case, wait for the job to finish.

```
wait(job1);
```

After the job finishes, fetch the results by using the `load` function. In this case, fetch the trained networks from the parallel parameter sweep in the submitted script and their accuracies.

```
load(job1,'accuracies');
accuracies
```

```
accuracies = 4×1

    0.8312
    0.8276
    0.8288
    0.8258
```

```
load(job1,'trainedNetworks');
trainedNetworks
```

```
trainedNetworks = 4×1 cell array
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
```

To load all the variables in the batch job, use the `load` function without arguments.

```
load(job1);
```

If you close MATLAB, you can still recover the job in the cluster to fetch the results either while the computation is taking place or after the computation is complete. Before closing MATLAB, make a note of the job ID and then retrieve the job later by using the `findJob` function.

To retrieve a job, first create a cluster object for your cluster by using the `parcluster` function. Then, provide the job ID to `findJob`. In this case, the job ID is `1`.

```
c = parcluster('MyClusterInTheCloud');
job = findJob(c,'ID',1);
```

Delete the job when you are done. The job is removed from the Job Monitor.

```
delete(job1);
```

Use Job Monitor to Fetch Results

When you submit batch jobs, all the computations happen in the cluster and you can safely close MATLAB. You can check the status of your jobs by using the Job Monitor in another MATLAB session.

When a job is done, you can retrieve the results from the Job Monitor. In the **Environment** section on the **Home** tab, select **Parallel > Monitor Jobs** to open the Job Monitor. Then right-click a job to display the context menu. From this menu, you can:

• Load the job into the workspace by clicking **Show Details**

• Load all variables in the job by clicking **Load Variables**

• Delete the job when you are done by clicking **Delete**
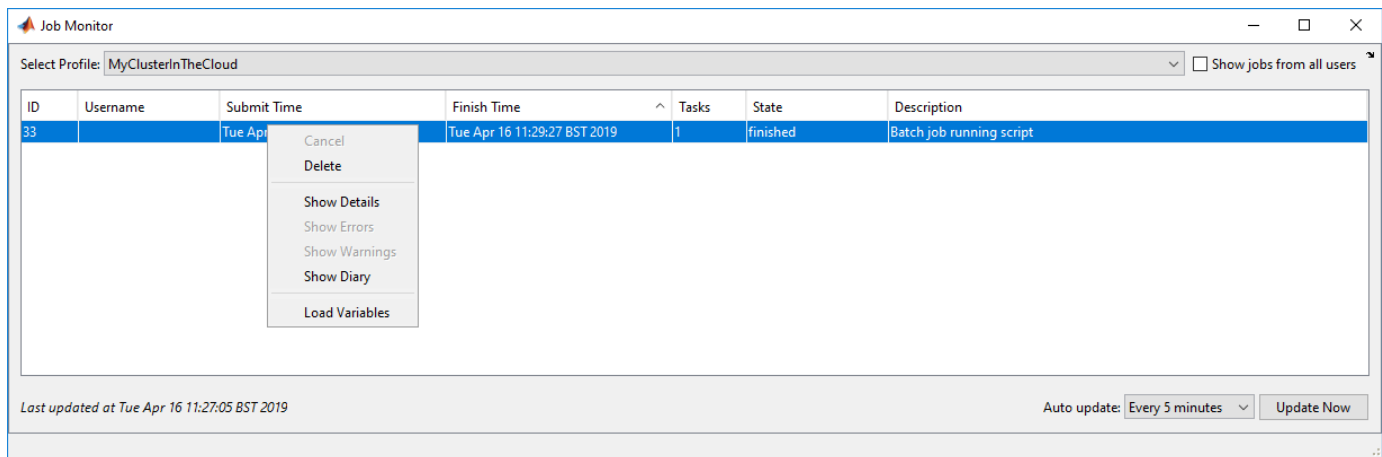
See Also
`batch`

Related Examples
• "Use parfor to Train Multiple Deep Learning Networks" on page 9-70
• "Upload Deep Learning Data to the Cloud" on page 9-77
• "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)
• "Run Batch Job and Access Files from Workers" on page 9-34

More About
• "Batch Processing"

Run MATLAB Functions on Multiple GPUs

This example shows how to run MATLAB code on multiple GPUs in parallel, first on your local machine, then scaling up to a cluster. As a sample problem, the example uses the logistic map, an equation that models the growth of a population.

A growing number of features in MATLAB offer automatic parallel support, including multi-gpu support, without requiring any extra coding. For details, see "Run MATLAB Functions with Automatic Parallel Support" on page 1-19. For example, the `trainNetwork` function offers multi-gpu support for training of neural networks and inference. For more information, see "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox).

Use a Single GPU

To run computations on a single GPU, use `gpuArray` objects as inputs to GPU-enabled MATLAB functions. To learn more about GPU-enabled functions, see "Run MATLAB Functions on a GPU" on page 8-9.

Create gpuArrays for the growth rate, `r`, and the population, `x`. For more information on creating gpuArrays, see "Establish Arrays on a GPU" on page 8-3.

```
N = 1000;
r = gpuArray.linspace(0,4,N);
x = rand(1,N,'gpuArray');
```

Use a simple algorithm to iterate the logistic map. Because the algorithm uses GPU-enabled operators on gpuArrays, the computations run on the GPU.

```
numIterations = 1000;
for n=1:numIterations
    x = r.*x.*(1-x);
end
```

When the computations are done, plot the growth rate against the population.

```
plot(r,x,'.');
```
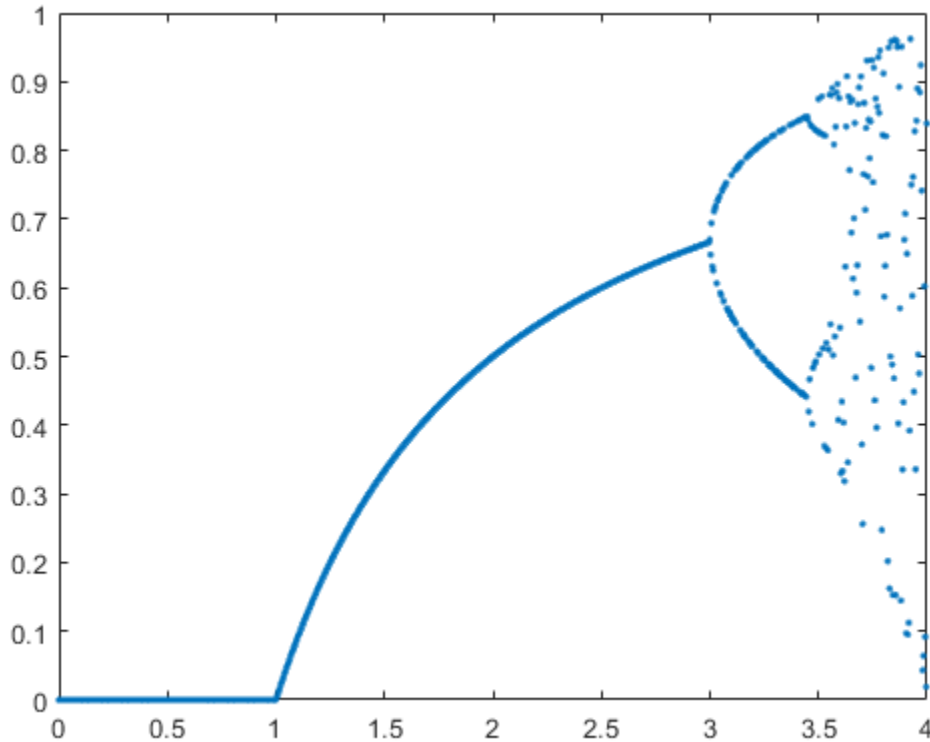
If you need more performance, gpuArrays supports several options. For a list, see the `gpuArray` function page. For example, the algorithm in this example only performs element-wise operations on gpuArrays, and so you can use the `arrayfun` function to precompile them for GPU.

Use Multiple GPUs with `parfor`

You can use `parfor` loops to distribute `for`-loop iterations among parallel workers. If your computations use GPU-enabled functions, then the computations run on the GPU of the worker. For example, if you use the Monte Carlo method to randomly simulate the evolution of populations, simulations are computed with multiple GPUs in parallel using a `parfor` loop.

Create a parallel pool with as many workers as GPUs available. To determine the number of GPUs available, use the `gpuDeviceCount` function. By default, MATLAB assigns a different GPU to each worker for best performance. For more information on selecting GPUs in a parallel pool, see "Use Multiple GPUs in a Parallel Pool" on page 10-33.

```
parpool(gpuDeviceCount);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 2 workers.
```

Define the number of simulations, and create an array in the GPU to store the population vector for each simulation.

```
numSimulations = 100;
X = zeros(numSimulations,N,'gpuArray');
```

Use a `parfor` loop to distribute simulations to workers in the pool. The code inside the loop creates a random gpuArray for the initial population, and iterates the logistic map on it. Because the code uses GPU-enabled operators on gpuArrays, the computations automatically run on the GPU of the worker.

```
parfor i = 1:numSimulations
    X(i,:) = rand(1,N,'gpuArray');
    for n=1:numIterations
        X(i,:) = r.*X(i,:).*(1-X(i,:));
    end
end
```

When the computations are done, plot the results of all simulations. Each color represents a different simulation.

```
figure
plot(r,X,'.');
```
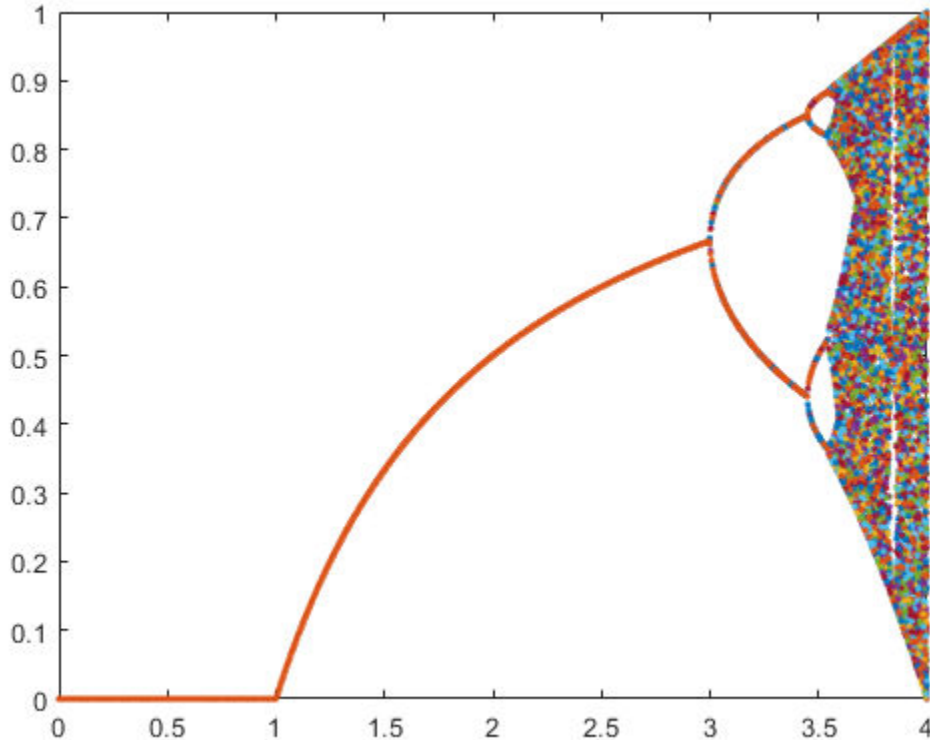

If you need greater control over your calculations, you can use more advanced parallel functionality. For example, you can use a `parallel.pool.DataQueue` to send data from the workers during computations. For an example, see "Plot During Parameter Sweep with parfor" on page 9-97.

If you want to generate a reproducible set of random numbers, you can control the random number generation on the worker GPU. For more information, see "Control Random Number Streams on Workers" on page 5-29.

Use Multiple GPUs Asynchronously with `parfeval`

You can use `parfeval` to run computations asynchronously on parallel pool workers. If your computations use GPU-enabled functions, then the computations run on the GPU of the worker. As an example, you run Monte Carlo simulations on multiple GPUs asynchronously.

To hold the results of computations after the workers complete them, use future objects. Preallocate an array of future objects for the result of each simulation.

```
f(numSimulations) = parallel.FevalFuture;
```

To run computations with `parfeval`, you must place them inside a function. For example, `myParallelFcn` contains the code of a single simulation.

```
type myParallelFcn

function x = myParallelFcn(r)
    N = 1000;
    x = gpuArray.rand(1,N);
    numIterations = 1000;
    for n=1:numIterations
        x = r.*x.*(1-x);
    end
end
```

Use a `for` loop to loop over simulations, and use `parfeval` to run them asynchronously on a worker in the parallel pool. `myParallelFcn` uses GPU-enabled functions on gpuArrays, so they run on the GPU of the worker. Because `parfeval` performs the computations asynchronously, it does not block MATLAB, and you can continue working while computations happen.

```
for i=1:numSimulations
    f(i) = parfeval(@myParallelFcn,1,r);
end
```

To collect the results from `parfeval` when they are ready, you can use `fetchOutputs` or `fetchNext` on the future objects. Also, you can use `afterEach` or `afterAll` to invoke functions on the results automatically when they are ready. For example, to plot the result of each simulation immediately after it completes, use `afterEach` on the future objects. Each color represents a different simulation.

```
figure
hold on
afterEach(f,@(x) plot(r,x,'.'),0);
```
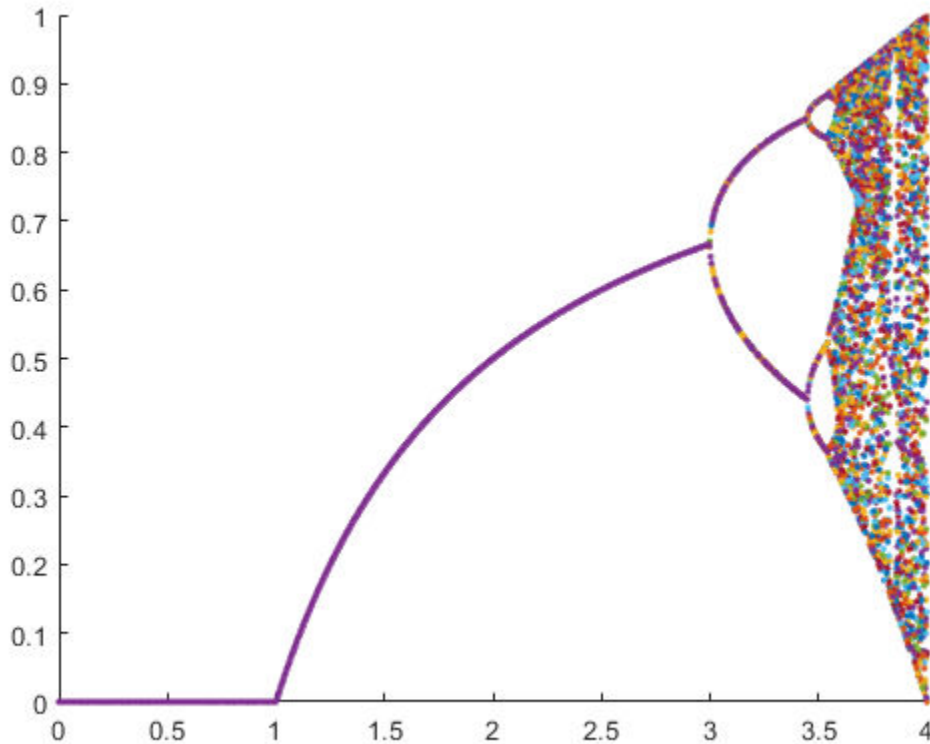
Use Multiple GPUs in a Cluster

If you have access to a cluster with multiple GPUs, then you can scale up your computations. Use the `parpool` function to start a parallel pool on the cluster. When you do so, parallel features, such as `parfor` loops or `parfeval`, run on the cluster workers. If your computations use GPU-enabled functions on gpuArrays, then those functions run on the GPU of the cluster worker. To learn more about running parallel features on a cluster, see "Scale Up from Desktop to Cluster" on page 9-88.

See Also

`afterEach` | `fetchOutputs (FevalFuture)` | `gpuArray` | `gpuDevice` | `parfeval` | `parfor` | `parpool`

Related Examples

- "Run MATLAB Functions on a GPU" on page 8-9
- "Scale Up from Desktop to Cluster" on page 9-88
- "Scale Up Deep Learning in Parallel and in the Cloud" (Deep Learning Toolbox)

More About

- "GPU Computing in MATLAB"

External Websites

- Running Monte Carlo Simulations on Multiple GPUs

Scale Up from Desktop to Cluster

This example shows how to develop your parallel MATLAB® code on your local machine and scale up to a cluster. Clusters provide more computational resources to speed up and distribute your computations. You can run your code interactively in parallel on your local machine, then on a cluster, without changing your code. When you are done prototyping your code on your local machine, you can offload your computations to the cluster using batch jobs. So, you can close MATLAB and retrieve the results later.

Develop Your Algorithm

Start by prototyping your algorithm on your local machine. The example uses integer factorization as a sample problem. It is a computationally intensive problem, where the complexity of the factorization increases with the magnitude of the number. You use a simple algorithm to factorize a sequence of integer numbers.

Create a vector of prime numbers in 64-bit precision, and multiply pairs of prime numbers randomly to obtain large composite numbers. Create an array to store the results of each factorization. The code in each of the following sections in this example can take more than 20 min. To make it faster, reduce the workload by using fewer prime numbers, such as 2^19. Run with 2^21 to see the optimum final plots.

```
primeNumbers = primes(uint64(2^21));
compositeNumbers = primeNumbers.*primeNumbers(randperm(numel(primeNumbers)));
factors = zeros(numel(primeNumbers),2);
```

Use a loop to factor each composite number, and measure the time that the computation takes.

```
tic;
for idx = 1:numel(compositeNumbers)
    factors(idx,:) = factor(compositeNumbers(idx));
end
toc
```

Elapsed time is 684.464556 seconds.

Run Your Code on a Local Parallel Pool

Parallel Computing Toolbox™ enables you to scale up your workflow by running on multiple workers in a parallel pool. The iterations in the previous for loop are independent, and so you can use a parfor loop to distribute iterations to multiple workers. Simply transform your for loop into a parfor loop. Then, run the code and measure the overall computation time. The code runs in a parallel pool with no further changes, and the workers send your computations back to the local workspace. Because the workload is distributed across several workers, the computation time is lower.

```
tic;
parfor idx = 1:numel(compositeNumbers)
    factors(idx,:) = factor(compositeNumbers(idx));
end
toc
```

Elapsed time is 144.550358 seconds.

When you use `parfor` and you have Parallel Computing Toolbox, MATLAB automatically starts a parallel pool of workers. The parallel pool takes some time to start. This example shows a second run with the pool already started.

The default cluster profile is `'local'`. You can check that this profile is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. With this profile enabled, MATLAB creates workers on your machine for the parallel pool. When you use the `'local'` profile, MATLAB, by default, starts as many workers as physical cores in your machine, up to your preferred number of workers. You can control parallel behavior using the parallel preferences. On the MATLAB **Home** tab, select **Parallel > Parallel Preferences**.

To measure the speedup with the number of workers, run the same code several times, limiting the maximum number of workers. First, define the number of workers for each run, up to the number of workers in the pool, and create an array to store the result of each test.

```
numWorkers = [1 2 4 6];
tLocal = zeros(size(numWorkers));
```

Use a loop to iterate through the maximum number of workers, and run the previous code. To limit the number of workers, use the second input argument of `parfor`.

```
for w = 1:numel(numWorkers)
    tic;
    parfor (idx = 1:numel(compositeNumbers), numWorkers(w))
        factors(idx,:) = factor(compositeNumbers(idx));
    end
    tLocal(w) = toc;
end
```

Calculate the speedup by computing the ratio between the computation time of a single worker and the computation time of each maximum number of workers. To visualize how the computations scale up with the number of workers, plot the speedup against the number of workers. Observe that the speedup increases with the number of workers. However, the scaling is not perfect due to overhead associated with parallelization.

```
f = figure;
speedup = tLocal(1)./tLocal;
plot(numWorkers, speedup);
title('Speedup with the number of workers');
xlabel('Number of workers');
xticks(numWorkers);
ylabel('Speedup');
```
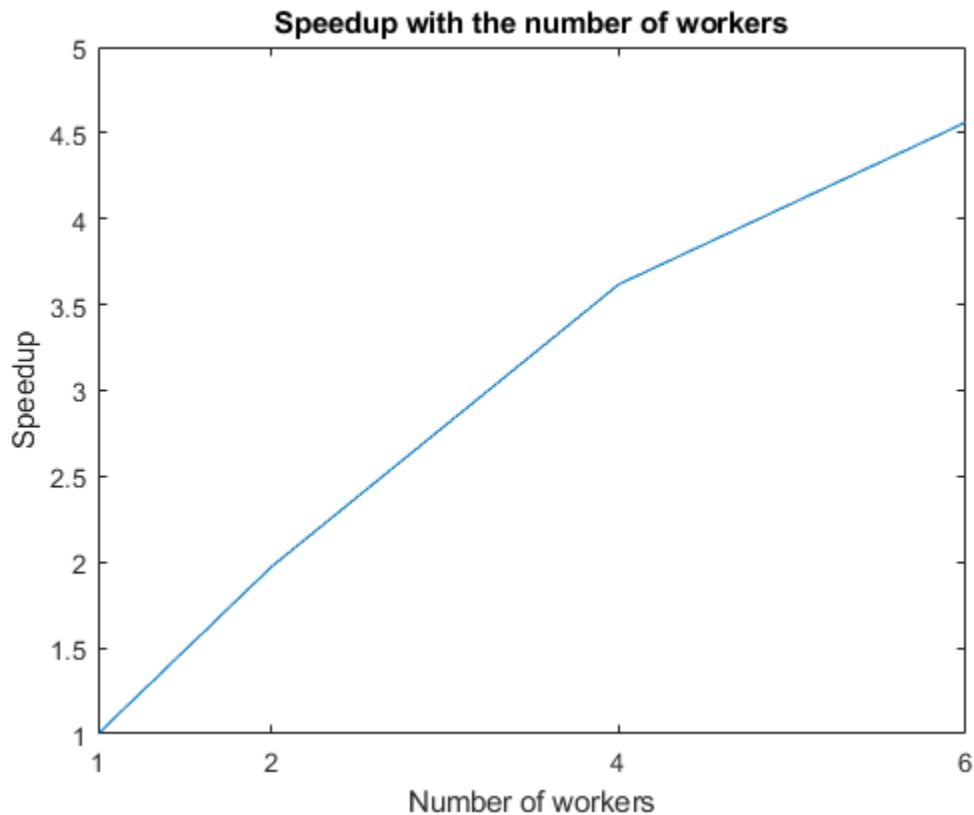
When you are done with your computation, delete the current parallel pool so you can create a new one for your cluster. You can obtain the current parallel pool with the `gcp` function.

```
delete(gcp);
```

Set up Your Cluster

If your computing task is too big or too slow for your local computer, you can offload your calculation to a cluster onsite or in the cloud. Before you can run the next sections, you must get access to a cluster. On the MATLAB **Home** tab, go to **Parallel > Discover Clusters** to find out if you already have access to a cluster with MATLAB Parallel Server™. For more information, see "Discover Clusters" on page 5-12.
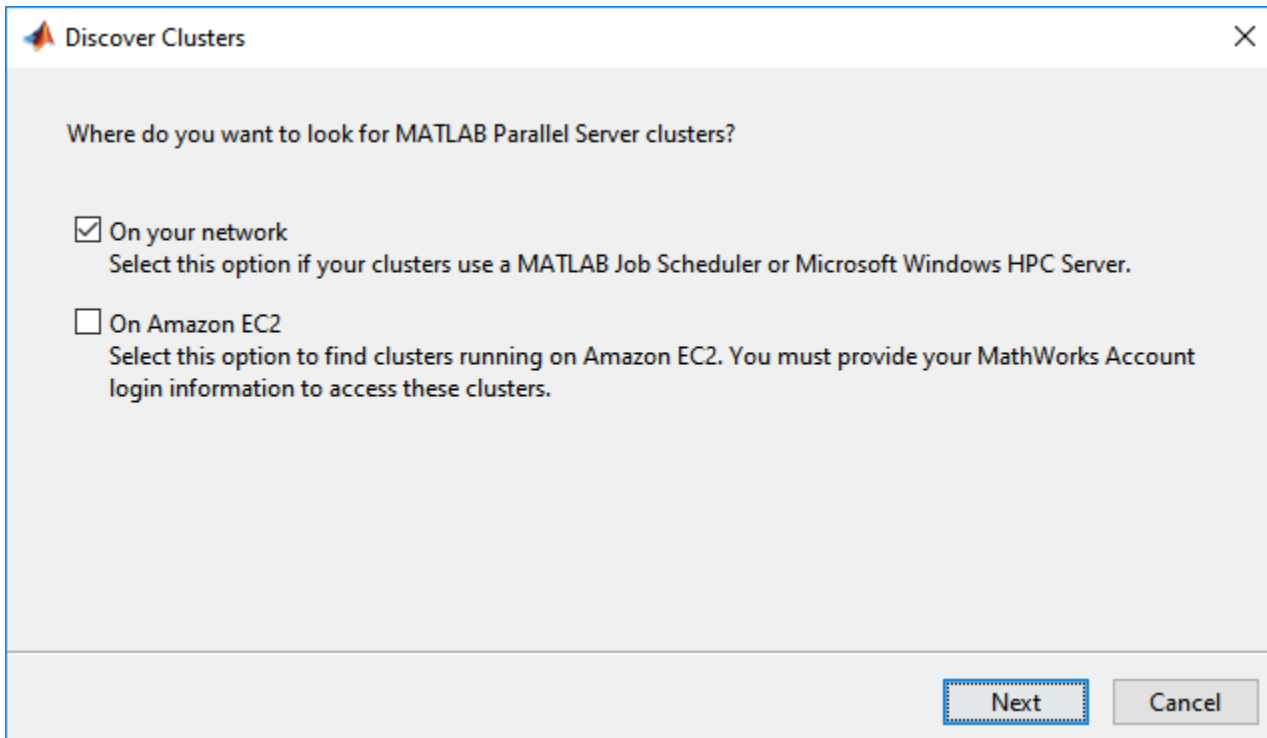
If you do not have access to a cluster, you must configure access to one before you can run the next sections. In MATLAB, you can create clusters in a cloud service, such as Amazon AWS, directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. To learn more about scaling up to the cloud, see Getting Started with Cloud Center. To learn more about your options for scaling to a cluster in your network, see "Get Started with MATLAB Parallel Server" (MATLAB Parallel Server).

After you set up a cluster profile, you can modify its properties in **Parallel > Create and Manage Clusters**. For more information, see "Discover Clusters and Use Cluster Profiles" on page 5-11. The following image shows a cluster profile in the *Cluster Profile Manager*:
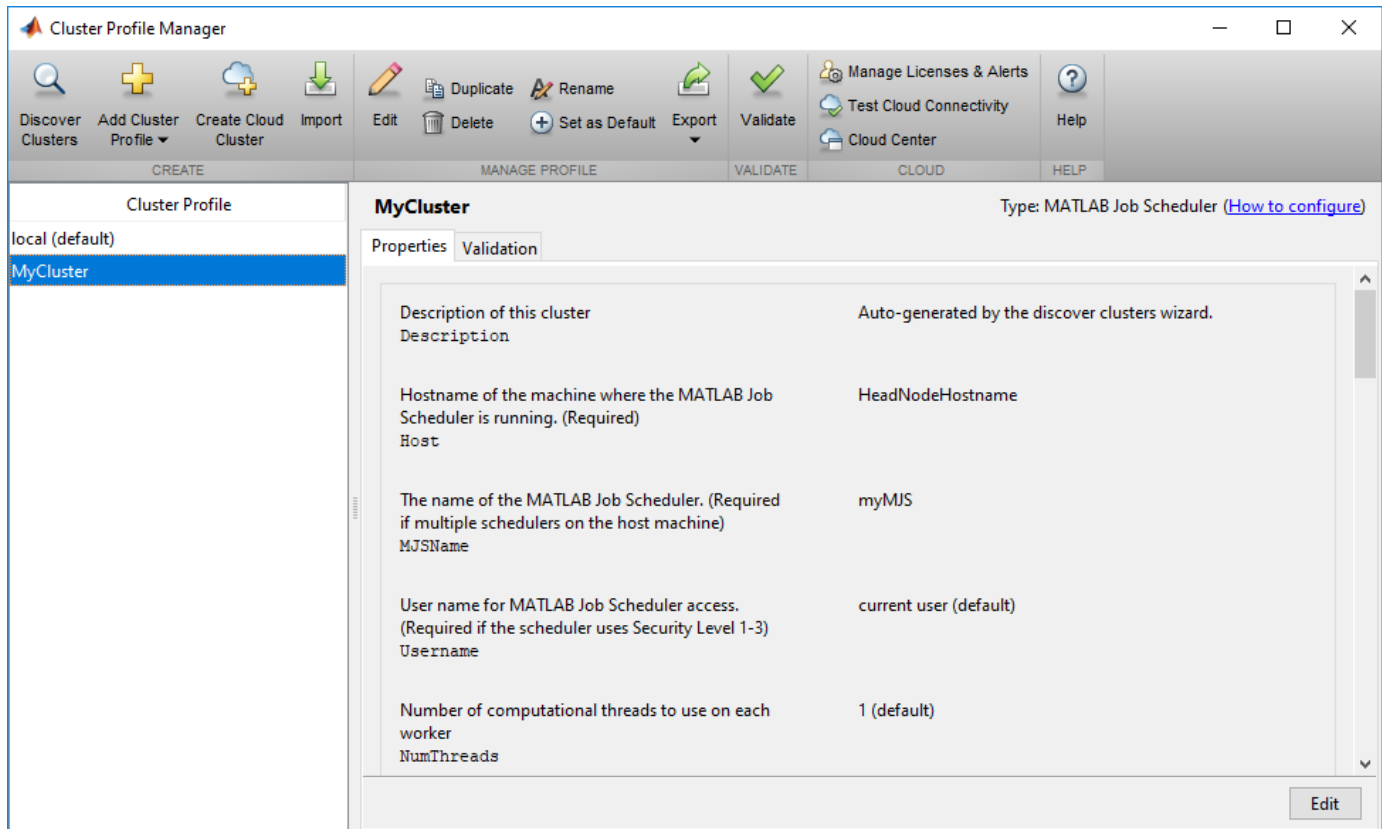
Run Your Code on a Cluster Parallel Pool

If you want to run parallel functions in the cluster by default, set your cluster profile as default in **Parallel > Select a Default Cluster**:
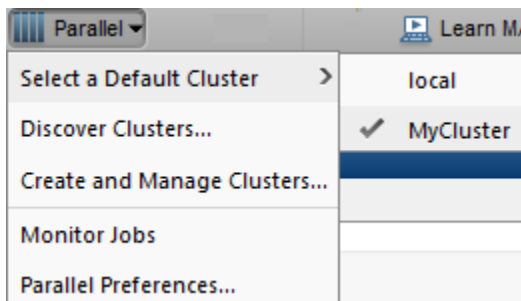
You can also use a programmatic approach to specify your cluster. To do so, start a parallel pool in the cluster by specifying the name of your cluster profile in the `parpool` command. In the following code, replace `MyCluster` with the name of your cluster profile. Also specify the number of workers with the second input argument.

```
parpool('MyCluster',64);
```

```
Starting parallel pool (parpool) using the 'MyCluster' profile ...
connected to 64 workers.
```

As before, measure the speedup with the number of workers by running the same code several times, and limiting the maximum number of workers. Because the cluster in this example allows for more

workers than the local setup, `numWorkers` can hold more values. If you run this code, the `parfor` loop now runs in the cluster.

```
numWorkers = [1 2 4 6 16 32 64];
tCluster = zeros(size(numWorkers));

for w = 1:numel(numWorkers)
    tic;
    parfor (idx = 1:numel(compositeNumbers), numWorkers(w))
        factors(idx,:) = factor(compositeNumbers(idx));
    end
    tCluster(w) = toc;
end
```

Calculate the speedup, and plot it against the number of workers to visualize how the computations scale up with the number of workers. Compare the results with those of the local setup. Observe that the speedup increases with the number of workers. However, the scaling is not perfect due to overhead associated with parallelization.

```
figure(f);
hold on
speedup = tCluster(1)./tCluster;
plot(numWorkers, speedup);
title('Speedup with the number of workers');
xlabel('Number of workers');
xticks(numWorkers(2:end));
ylabel('Speedup');
```
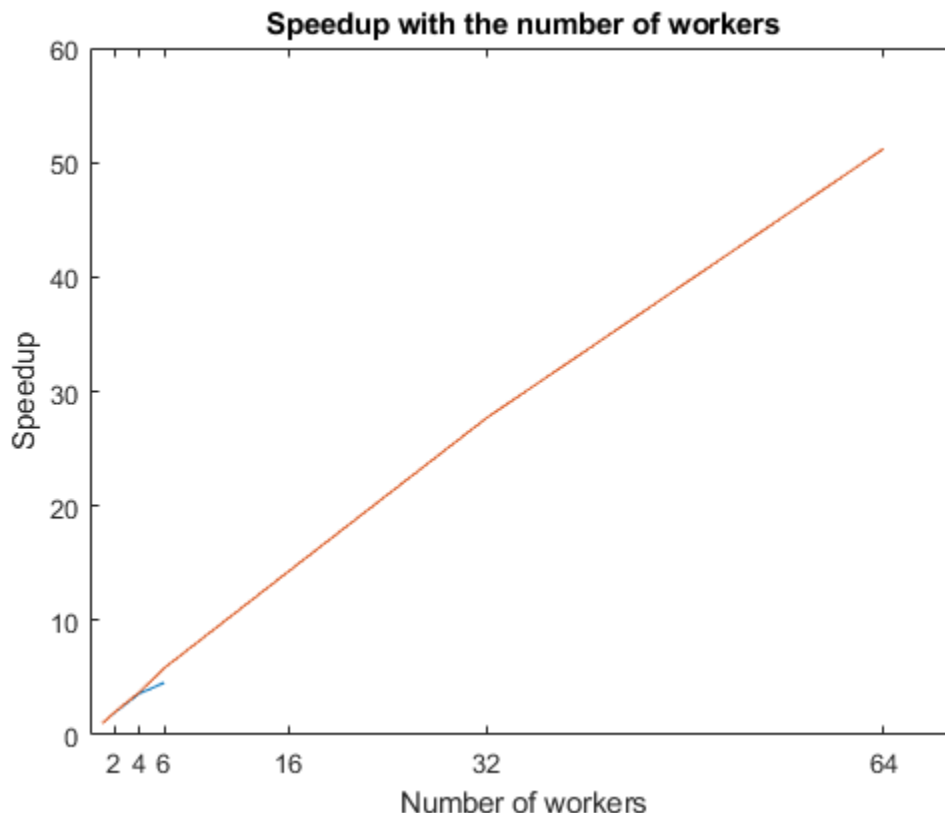
When you are done with your computations, delete the current parallel pool.

```
delete(gcp);
```

Offload and Scale Your Computations with `batch`

After you are done prototyping and running interactively, you can use batch jobs to offload the execution of long-running computations in the background with batch processing. The computation happens in the cluster, and you can close MATLAB and retrieve the results later.

Use the `batch` function to submit a batch job to your cluster. You can place the contents of your algorithm in a script, and use the `batch` function to submit it. For example, the script myParallelAlgorithm performs a simple benchmark based on the integer factorization problem shown in this example. The script measures the computation time of several problem complexities with different number of workers.

Note that if you send a script file using `batch`, MATLAB transfers all the workspace variables to the cluster, even if your script does not use them. If you have a large workspace, it impacts negatively the data transfer time. As a best practice, convert your script to a function file to avoid this communication overhead. You can do this by simply adding a function line at the beginning of your script. To learn how to convert myParallelAlgorithm to a function, see myParallelAlgorithmFcn.

The following code submits `myParallelAlgorithmFcn` as a batch job. `myParallelAlgorithmFcn` returns two output arguments, `numWorkers` and `time`, and you must specify 2 as the number of outputs input argument. Because the code needs a parallel pool for the `parfor` loop, use the `'Pool'` name-value pair in `batch` to specify the number of workers. The cluster uses an additional worker to run the function itself. By default, `batch` changes the current folder of the workers in the cluster to the current folder of the MATLAB client. It can be useful to control the current folder. For example, if your cluster uses a different filesystem, and therefore the paths are different, such as when you submit from a Windows client machine to a Linux cluster. Set the name-value pair `'CurrentFolder'` to a folder of your choice, or to `'.'` to avoid changing the folder of the workers.

```
totalNumberOfWorkers = 65;
cluster = parcluster('MyCluster');
job = batch(cluster,'myParallelAlgorithmFcn',2,'Pool',totalNumberOfWorkers-1,'CurrentFolder','.')
```

To monitor the state of your job after it is submitted, open the Job Monitor in **Parallel** > **Monitor Jobs**. When computations start in the cluster, the state of the job changes to `running`:
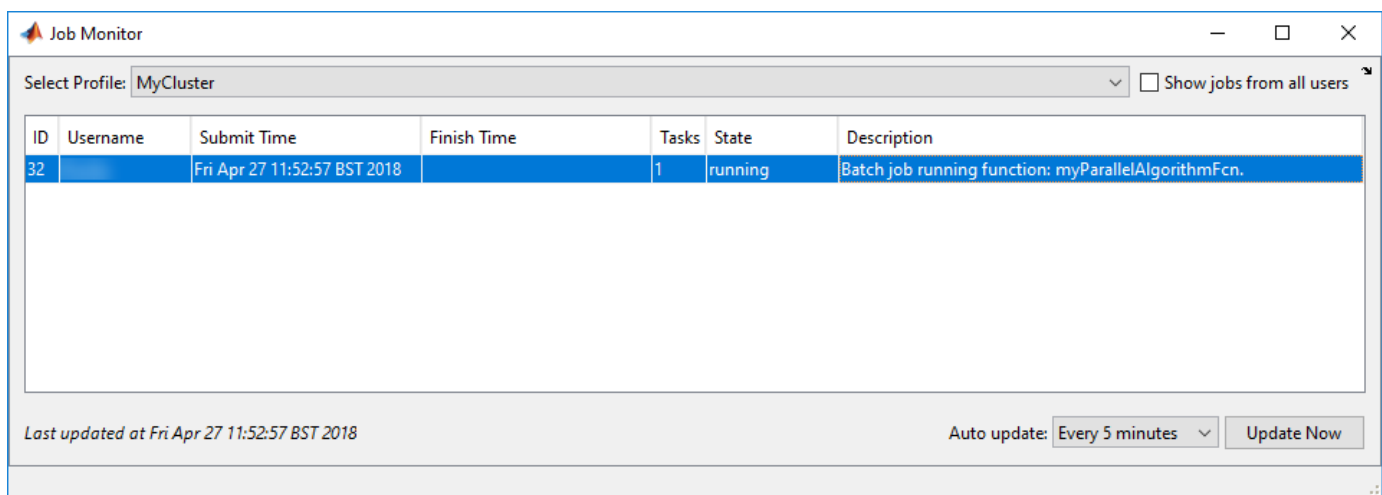
You can close MATLAB after the job has been submitted. When you open MATLAB again, the Job Monitor keeps track of the job for you, and you can interact with it if you right-click it. For example, to retrieve the job object, select **Show Details**, and to transfer the outputs of the batch job into the workspace, select **Fetch Outputs**.
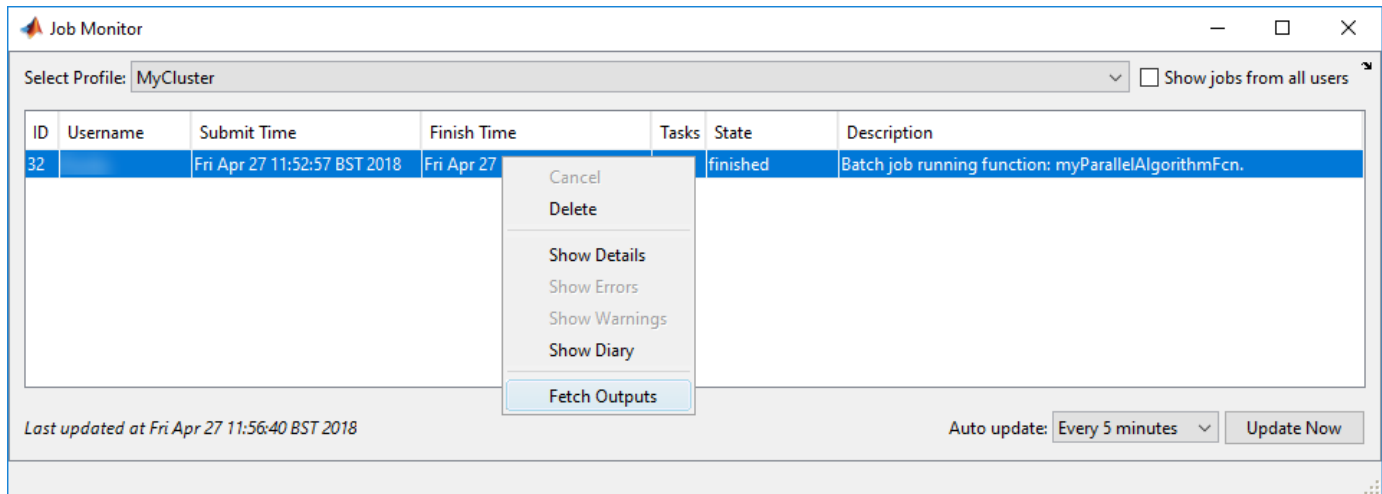
Alternatively, if you want to block MATLAB until the job completes, use the `wait` function on the job object.

```
wait(job);
```

To transfer the outputs of the function from the cluster, use the `fetchOutputs` function.

```
outputs = fetchOutputs(job);
numWorkers = outputs{1};
time = outputs{2};
```

After retrieving the results, you can use them for calculations on your local machine. Calculate the speedup, and plot it against the number of workers. Because the code runs factorizations for different problem complexities, you get a plot for each level. You can see that, for each problem complexity, the speedup increases with the number of workers, until the overhead for additional workers is greater than the performance gain from parallelization. As you increase the problem complexity, you achieve better speedup at large numbers of workers, because overhead associated with parallelization is less significant.

```
figure
speedup = time(1,:)./time;
plot(numWorkers,speedup);
legend('Problem complexity 1','Problem complexity 2','Problem complexity 3','Problem complexity 4
title('Speedup vs complexity');
xlabel('Number of workers');
xticks(numWorkers(2:end));
ylabel('Speedup');
```

Speedup vs complexity

Legend:
- Problem complexity 1
- Problem complexity 2
- Problem complexity 3
- Problem complexity 4

(x-axis: Number of workers; y-axis: Speedup)
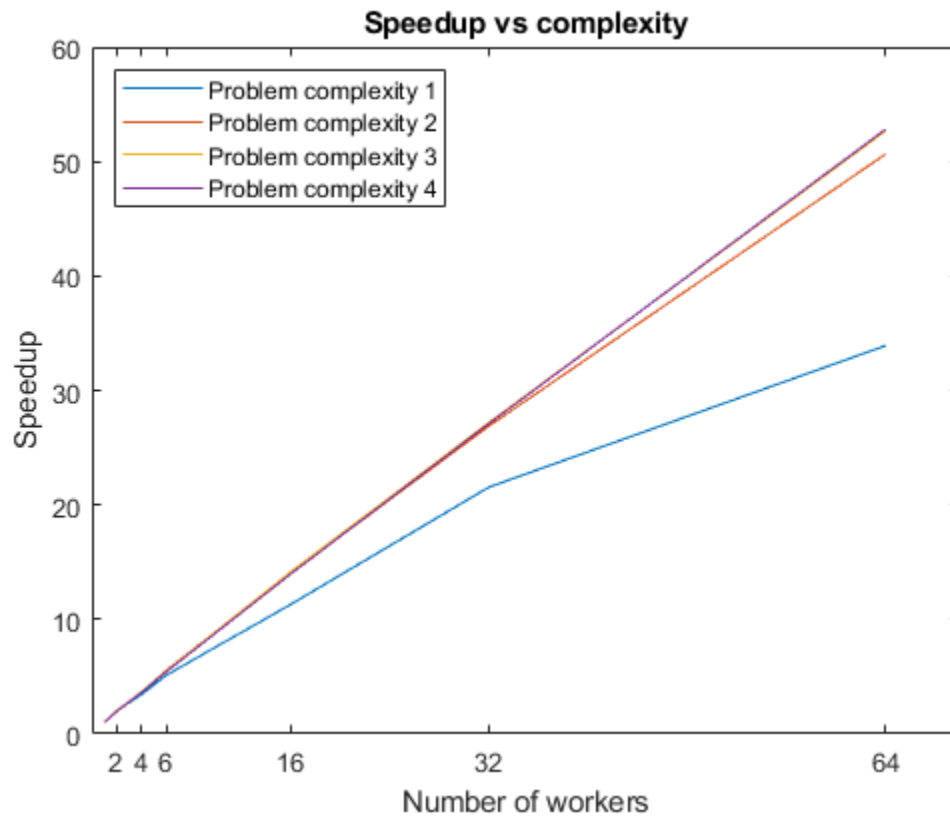
See Also
`batch` | `fetchOutputs (Job)` | `parfor` | `parpool`

Related Examples
- "Discover Clusters and Use Cluster Profiles" on page 5-11

More About
- "Parallel for-Loops (parfor)"
- "Get Started with MATLAB Parallel Server" (MATLAB Parallel Server)

Plot During Parameter Sweep with parfor

This example shows how to perform a parameter sweep in parallel and plot progress during parallel computations. You can use a `DataQueue` to monitor results during computations on a parallel pool. You can also use a `DataQueue` with parallel language features such as `parfor`, `parfeval` and `spmd`.

The example shows how to perform a parameter sweep on a classical system, the Van der Pol oscillator. This system can be expressed as a set of ODEs dependent on the two Van der Pol oscillator parameters, μ and ν:

$$\dot{x} = \nu y$$

$$\dot{y} = \mu\left(1 - x^2\right)y - x$$

You can perform a parallel parameter sweep over the parameters μ and ν using a `parfor` loop to find out the mean period of y when varying them. The following animation shows an execution of this example in a local cluster.
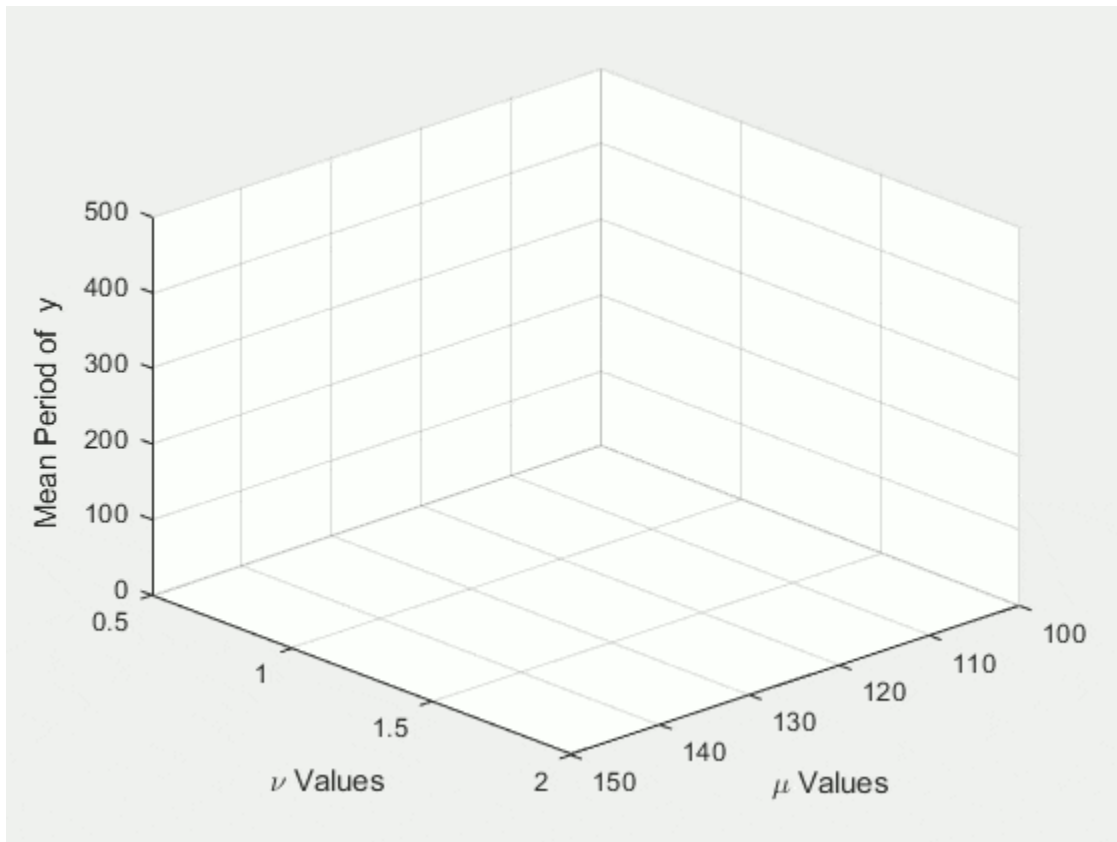
Set Up Parameter Sweep Values

Define the range of values for the parameters to be explored. Create a meshgrid to account for the different combinations of the parameters.

```
gridSize = 6;
mu = linspace(100, 150, gridSize);
```

```
nu = linspace(0.5, 2, gridSize);
[M,N] = meshgrid(mu,nu);
```

Prepare a Surface Plot to Visualize the Results

Declare a variable to store the results of the sweep. Use `nan` for preallocation to avoid plotting an initial surface. Create a surface plot to visualize the results of the sweep for each combination of the parameters. Prepare settings such as title, labels, and limits.

```
Z = nan(size(N));
c = surf(M, N, Z);
xlabel('\mu Values','Interpreter','Tex')
ylabel('\nu Values','Interpreter','Tex')
zlabel('Mean Period of  y')
view(137, 30)
axis([100 150 0.5 2 0 500]);
```
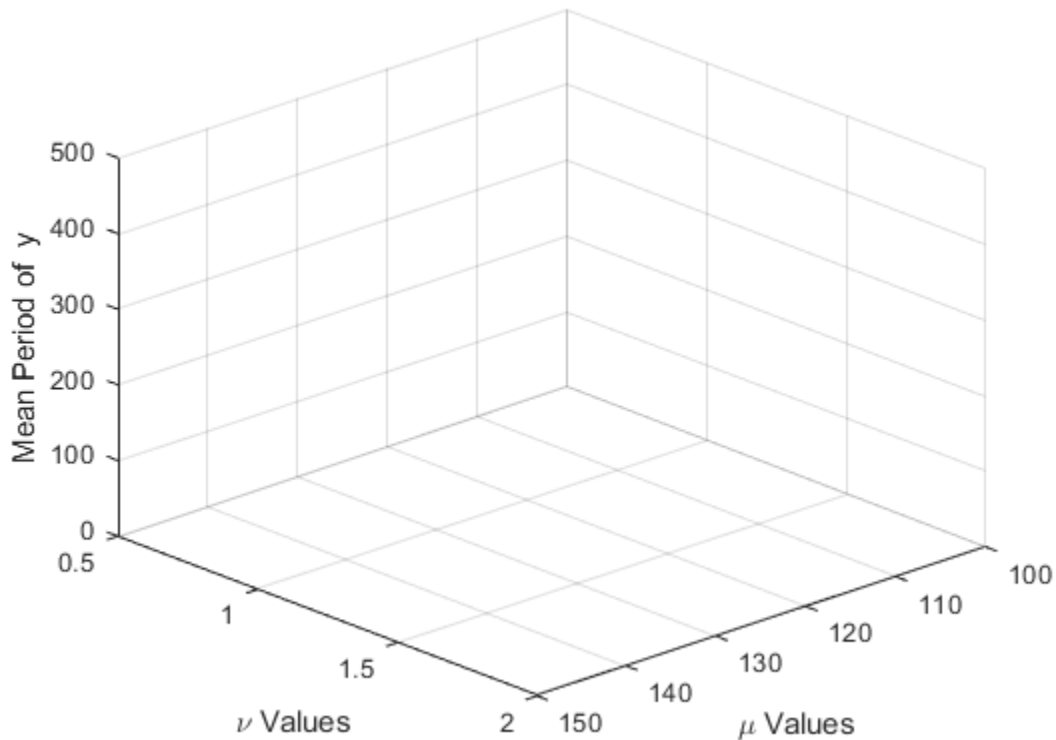

Set Up a DataQueue to Fetch Results During the Parameter Sweep

Create a `DataQueue` to send intermediate results from the workers to the client. Use the `afterEach` function to define a callback in the client that updates the surface each time a worker sends the current result.

```
D = parallel.pool.DataQueue;
D.afterEach(@(x) updateSurface(c, x));
```

Perform the Parameter Sweep and Plot Results

Use `parfor` to perform a parallel parameter sweep. Instruct the workers to solve the system for each combination of the parameters in the meshgrid, and compute the mean period. Immediately send the result of each iteration back to the client when the worker finishes computations.

```
parfor ii = 1:numel(N)
    [t, y] = solveVdp(M(ii), N(ii));
    l = islocalmax(y(:, 2));
    send(D, [ii mean(diff(t(l)))]);
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Scale Up to a Cluster

If you have access to a cluster, you can scale up your computation. To do this, delete the previous `parpool`, and open a new one using the profile for your larger cluster. The code below shows a cluster profile named `'MyClusterInTheCloud'`. To run this code yourself, you must replace `'MyClusterInTheCloud'` with the name of your cluster profile. Adjust the number of workers. The example shows 4 workers. Increase the size of the overall computation by increasing the size of the grid.

```
gridSize = 25;
delete(gcp('nocreate'));
parpool('MyClusterInTheCloud',4);
```

```
Starting parallel pool (parpool) using the 'MyClusterInTheCloud' profile ...
Connected to the parallel pool (number of workers: 4).
```

If you run the parameter sweep code again after setting the cluster profile, then the workers in the cluster compute and send the results to the MATLAB client when they become available. The following animation shows an execution of this example in a cluster.
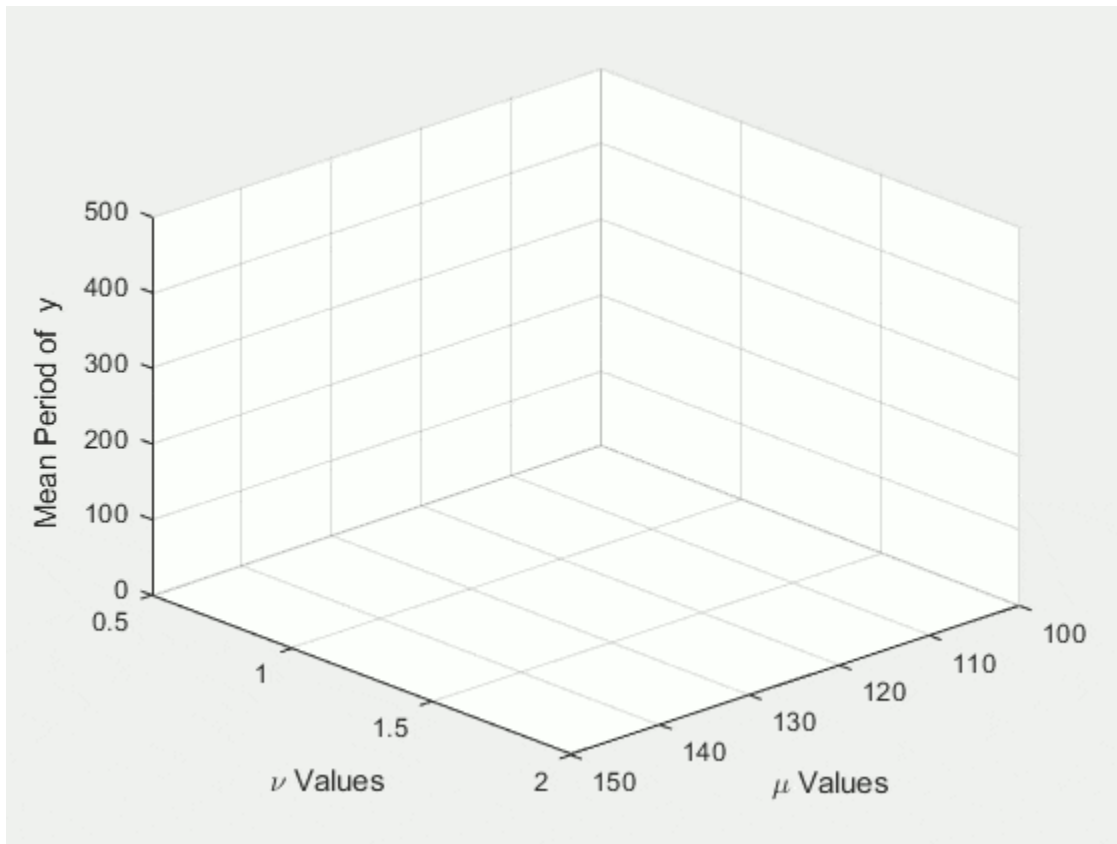
Helper Functions

Create a helper function to define the system of equations, and apply the solver on it.

```
function [t, y] = solveVdp(mu, nu)
f = @(~,y) [nu*y(2); mu*(1-y(1)^2)*y(2)-y(1)];
[t,y] = ode23s(f,[0 20*mu],[2; 0]);
end
```

Declare a function for the DataQueue to update the graph with the results that come from the workers.

```
function updateSurface(s, d)
s.ZData(d(1)) = d(2);
drawnow('limitrate');
end
```

See Also

"Parallel for-Loops (parfor)"

Update a User Interface Asynchronously Using afterEach and afterAll

You can perform asynchronous computations on workers using `parfeval` and leave the user interface responsive. Use `afterEach` to update the user interface when intermediate computations are ready. Use `afterAll` to update the user interface when all the computations are ready.

Create a simple user interface using a `waitbar`.

```
h = waitbar(0, 'Waiting...');
```


Use `parfeval` to carry out time-consuming computations in the workers, for example, eigenvalues of random matrices. The computations happen asynchronously and the user interface updates during computation. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx = 1:100
    f(idx) = parfeval(@(n) real(eig(randn(n))), 1, 5e2);
end
```

Compute the largest value in each of the computations when they become ready using `afterEach`. Update the proportion of finished futures in the waitbar when each of them completes using `afterEach`.

```
maxFuture = afterEach(f, @max, 1);
updateWaitbarFuture = afterEach(f, @(~) waitbar(sum(strcmp('finished', {f.State}))/numel(f), h),
```

Close the waitbar when all the computations are done. Use `afterAll` on `updateWaitbarFuture` to continue automatically with a close operation. `afterAll` obtains the figure handle from `updateWaitbarFuture` and executes its function on it.

```
closeWaitbarFuture = afterAll(updateWaitbarFuture, @(h) delete(h), 0);
```

Show a histogram after all the maximum values are computed. Use `afterAll` on `maxFuture` to continue the operation automatically. `afterAll` obtains the maximum values from `maxFuture` and calls `histogram` on them.

```
showsHistogramFuture = afterAll(maxFuture, @histogram, 0);
```

See Also

"Asynchronous Parallel Programming"

Simple Benchmarking of PARFOR Using Blackjack

This example benchmarks the `parfor` construct by repeatedly playing the card game of blackjack, also known as 21. We use `parfor` to play the card game multiple times in parallel, varying the number of MATLAB® workers, but always using the same number of players and hands.

Related examples:

Parallel Version

The basic parallel algorithm uses the `parfor` construct to execute independent passes through a loop. It is a part of the MATLAB® language, but behaves essentially like a regular `for`-loop if you do not have access to the Parallel Computing Toolbox™ product. Thus, our initial step is to convert a loop of the form

```
for i = 1:numPlayers
    S(:, i) = playBlackjack();
end
```

into the equivalent `parfor` loop:

```
parfor i = 1:numPlayers
    S(:, i) = playBlackjack();
end
```

We modify this slightly by specifying an optional argument to `parfor`, instructing it to limit to `n` the number of workers it uses for the computations. The actual code is as follows:

```
dbtype pctdemo_aux_parforbench
```

```
1    function S = pctdemo_aux_parforbench(numHands, numPlayers, n)
2    %PCTDEMO_AUX_PARFORBENCH Use parfor to play blackjack.
3    %   S = pctdemo_aux_parforbench(numHands, numPlayers, n) plays
4    %   numHands hands of blackjack numPlayers times, and uses no
5    %   more than n MATLAB(R) workers for the computations.
6
7    %   Copyright 2007-2009 The MathWorks, Inc.
8
9    S = zeros(numHands, numPlayers);
10   parfor (i = 1:numPlayers, n)
11       S(:, i) = pctdemo_task_blackjack(numHands, 1);
12   end
```

Check the Status of the Parallel Pool

We will use the parallel pool to allow the body of the `parfor` loop to run in parallel, so we start by checking whether the pool is open. We will then run the benchmark using anywhere between 2 and `poolSize` workers from this pool.

```
p = gcp;
if isempty(p)
    error('pctexample:backslashbench:poolClosed', ...
        ['This example requires a parallel pool. ' ...
```

```
            'Manually start a pool using the parpool command or set ' ...
            'your parallel preferences to automatically start a pool.']);
end
poolSize = p.NumWorkers;
```

Run the Benchmark: Weak Scaling

We time the execution of our benchmark calculations using 2 to `poolSize` workers. We use weak scaling, that is, we increase the problem size with the number of workers.

```
numHands = 2000;
numPlayers = 6;
fprintf('Simulating each player playing %d hands.\n', numHands);
t1 = zeros(1, poolSize);
for n = 2:poolSize
    tic;
        pctdemo_aux_parforbench(numHands, n*numPlayers, n);
    t1(n) = toc;
    fprintf('%d workers simulated %d players in %3.2f seconds.\n', ...
            n, n*numPlayers, t1(n));
end
```

```
Simulating each player playing 2000 hands.
2 workers simulated 12 players in 10.81 seconds.
3 workers simulated 18 players in 10.67 seconds.
4 workers simulated 24 players in 10.57 seconds.
5 workers simulated 30 players in 10.57 seconds.
6 workers simulated 36 players in 10.71 seconds.
7 workers simulated 42 players in 10.63 seconds.
8 workers simulated 48 players in 10.87 seconds.
9 workers simulated 54 players in 10.54 seconds.
10 workers simulated 60 players in 10.73 seconds.
11 workers simulated 66 players in 10.58 seconds.
12 workers simulated 72 players in 10.68 seconds.
13 workers simulated 78 players in 10.56 seconds.
14 workers simulated 84 players in 10.89 seconds.
15 workers simulated 90 players in 10.62 seconds.
16 workers simulated 96 players in 10.63 seconds.
17 workers simulated 102 players in 10.70 seconds.
18 workers simulated 108 players in 10.70 seconds.
19 workers simulated 114 players in 10.79 seconds.
20 workers simulated 120 players in 10.72 seconds.
21 workers simulated 126 players in 10.74 seconds.
22 workers simulated 132 players in 10.75 seconds.
23 workers simulated 138 players in 10.74 seconds.
24 workers simulated 144 players in 10.72 seconds.
25 workers simulated 150 players in 10.74 seconds.
26 workers simulated 156 players in 10.76 seconds.
27 workers simulated 162 players in 10.74 seconds.
28 workers simulated 168 players in 10.72 seconds.
29 workers simulated 174 players in 10.76 seconds.
30 workers simulated 180 players in 10.69 seconds.
31 workers simulated 186 players in 10.76 seconds.
32 workers simulated 192 players in 10.76 seconds.
33 workers simulated 198 players in 10.79 seconds.
34 workers simulated 204 players in 10.74 seconds.
35 workers simulated 210 players in 12.12 seconds.
36 workers simulated 216 players in 12.19 seconds.
```

9-103

```
37 workers simulated 222 players in 12.19 seconds.
38 workers simulated 228 players in 12.14 seconds.
39 workers simulated 234 players in 12.15 seconds.
40 workers simulated 240 players in 12.18 seconds.
41 workers simulated 246 players in 12.18 seconds.
42 workers simulated 252 players in 12.14 seconds.
43 workers simulated 258 players in 12.24 seconds.
44 workers simulated 264 players in 12.25 seconds.
45 workers simulated 270 players in 12.23 seconds.
46 workers simulated 276 players in 12.23 seconds.
47 workers simulated 282 players in 12.55 seconds.
48 workers simulated 288 players in 12.52 seconds.
49 workers simulated 294 players in 13.24 seconds.
50 workers simulated 300 players in 13.28 seconds.
51 workers simulated 306 players in 13.36 seconds.
52 workers simulated 312 players in 13.53 seconds.
53 workers simulated 318 players in 13.98 seconds.
54 workers simulated 324 players in 13.90 seconds.
55 workers simulated 330 players in 14.29 seconds.
56 workers simulated 336 players in 14.23 seconds.
57 workers simulated 342 players in 14.25 seconds.
58 workers simulated 348 players in 14.32 seconds.
59 workers simulated 354 players in 14.26 seconds.
60 workers simulated 360 players in 14.34 seconds.
61 workers simulated 366 players in 15.60 seconds.
62 workers simulated 372 players in 15.75 seconds.
63 workers simulated 378 players in 15.79 seconds.
64 workers simulated 384 players in 15.76 seconds.
```

We compare this against the execution using a regular `for`-loop in MATLAB®.

```
tic;
    S = zeros(numHands, numPlayers);
    for i = 1:numPlayers
        S(:, i) = pctdemo_task_blackjack(numHands, 1);
    end
t1(1) = toc;
fprintf('Ran in %3.2f seconds using a sequential for-loop.\n', t1(1));
```

```
Ran in 10.70 seconds using a sequential for-loop.
```

Plot the Speedup

We compare the speedup using `parfor` with different numbers of workers to the perfectly linear speedup curve. The speedup achieved by using `parfor` depends on the problem size as well as the underlying hardware and networking infrastructure.

```
speedup = (1:poolSize).*t1(1)./t1;
fig = pctdemo_setup_blackjack(1.0);
fig.Visible = 'on';
ax = axes('parent', fig);
x = plot(ax, 1:poolSize, 1:poolSize, '--', ...
    1:poolSize, speedup, 's', 'MarkerFaceColor', 'b');
t = ax.XTick;
t(t ~= round(t)) = []; % Remove all non-integer x-axis ticks.
ax.XTick = t;
legend(x, 'Linear Speedup', 'Measured Speedup', 'Location', 'NorthWest');
xlabel(ax, 'Number of MATLAB workers participating in computations');
ylabel(ax, 'Speedup');
```
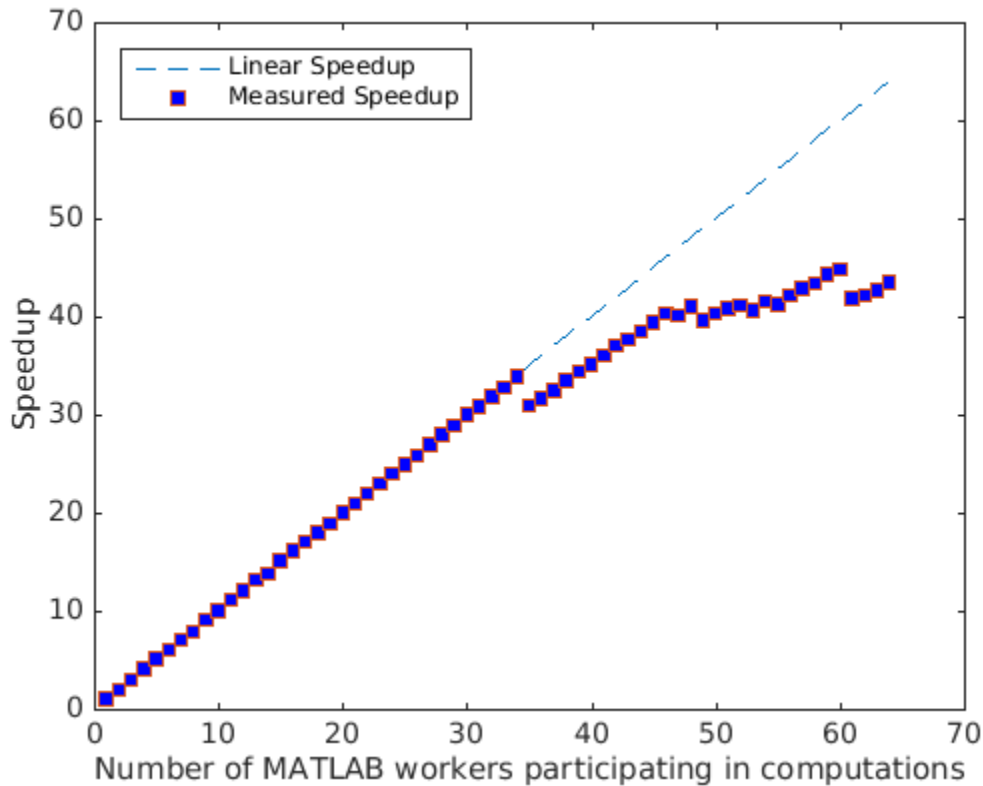
Measure the Speedup Distribution

To get reliable benchmark numbers, we need to run the benchmark multiple times. We therefore run the benchmark multiple times for `poolSize` workers to allow us to look at the spread of the speedup.

```matlab
numIter = 100;
t2 = zeros(1, numIter);
for i = 1:numIter
    tic;
        pctdemo_aux_parforbench(numHands, poolSize*numPlayers, poolSize);
    t2(i) = toc;
    if mod(i,20) == 0
        fprintf('Benchmark has run %d out of %d times.\n',i,numIter);
    end
end
```

```
Benchmark has run 20 out of 100 times.
Benchmark has run 40 out of 100 times.
Benchmark has run 60 out of 100 times.
Benchmark has run 80 out of 100 times.
Benchmark has run 100 out of 100 times.
```

Plot the Speedup Distribution

We take a close look at the speedup of our simple parallel program when using the maximum number of workers. The histogram of the speedup allows us to distinguish between outliers and the average speedup.

```
speedup = t1(1)./t2*poolSize;
clf(fig);
ax = axes('parent', fig);
hist(speedup, 5);
a = axis(ax);
a(4) = 5*ceil(a(4)/5); % Round y-axis to nearest multiple of 5.
axis(ax, a)
xlabel(ax, 'Speedup');
ylabel(ax, 'Frequency');
title(ax, sprintf('Speedup of parfor with %d workers', poolSize));
m = median(speedup);
fprintf(['Median speedup is %3.2f, which corresponds to '...
    'efficiency of %3.2f.\n'], m, m/poolSize);
```

Median speedup is 43.37, which corresponds to efficiency of 0.68.
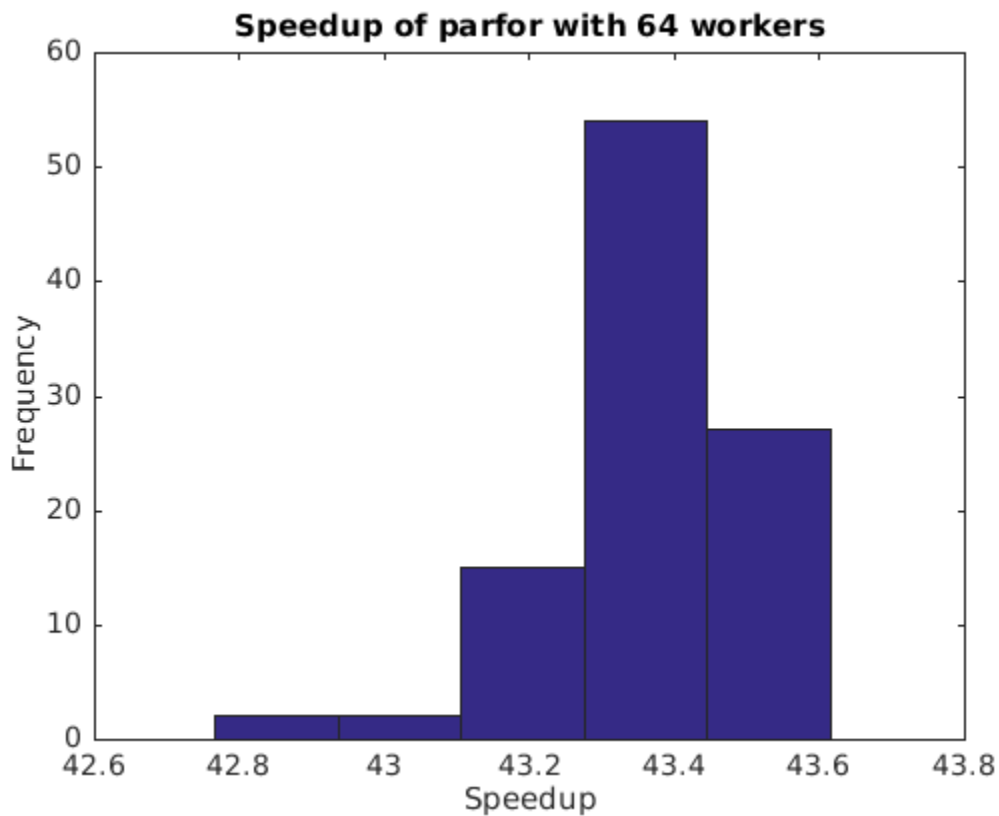
Use Distributed Arrays to Solve Systems of Linear Equations with Direct Methods

Distributed arrays are well-suited for large mathematical computations, such as large problems of linear algebra. This example shows how you can solve a system of linear equations of the form $Ax = b$ in parallel with a direct method using distributed arrays. In the same way as for arrays stored in the client memory, you can use `mldivide` to solve can systems of linear equations defined using distributed arrays, so you do not need to change your code.

Distributed arrays distribute data from your client workspace to a parallel pool in your local machine or in a cluster. Each worker stores a portion of the array in its memory, but can also communicate with the other workers to access all segments of the array. Distributed arrays can contain different types of data including full and sparse matrices.

Direct methods of solving linear equations typically factorize the coefficient matrix A to compute the solution. `mldivide` selects one of a set of direct solver methods depending on the structure of A and whether A is full or sparse.

This example demonstrates how to solve a simple system of linear equations of the form $Ax = b$ with an exact, known solution x. The system is defined by the matrix A and the column vector b. The solution x is also a column vector. In this example, the system is defined using full and sparse matrices. The required code is the same for systems defined using distributed arrays or arrays on the client memory.

For a related example that shows how to use iterative solvers and distributed arrays, see "Use Distributed Arrays to Solve Systems of Linear Equations with Iterative Methods" on page 9-112.

Solve a Full Matrix System

First define the coefficient matrix A as variable in the client memory, `A`, and then pass this matrix to the `distributed` function to create a distributed version of the same matrix, `ADist`. When you use the `distributed` function, MATLAB automatically starts a parallel pool using your default cluster settings.

```
n = 1e3;
A = randi(100,n,n);
ADist = distributed(A);
```

You can now define the right hand vector b. In this example, b is defined as the row sum of A, which leads to an exact solution to $Ax = b$ of the form $x_{\text{exact}} = [1, \ldots, 1]^T$.

```
b = sum(A,2);
bDist = sum(ADist,2);
```

Since `sum` acts on a distributed array, `bDist` is also distributed and its data is stored in the memory of the workers of your parallel pool. Finally, define the exact solutions for comparison with the solutions obtained using direct numerical methods.

```
xEx = ones(n,1);
xDistEx = ones(n,1,'distributed');
```

Now that you have defined your system of linear equations, you can use `mldivide` to solve the system directly. In MATLAB, you can call `mldivide` using the special operator \. You do not have to

change your code to solve the distributed system as `mldivide` has automatic support for distributed arrays.

Once you have calculated the solution, you can check the error between each element of the obtained result x and the expected values of x_{exact}.

```
x = A\b;
err = abs(xEx-x);

xDist = ADist\bDist;
errDist = abs(xDistEx-xDist);

figure
subplot(2,1,1)
semilogy(err,'o');
title('System of Linear Equations with Full Matrices');
ylabel('Absolute Error');
xlabel('Element in x');
ylim([10e-17,10e-13])
subplot(2,1,2)
semilogy(errDist,'o');
title('System of Linear Equations with Distributed Full Matrices');
ylabel('Absolute Error');
xlabel('Element in x');
ylim([10e-17,10e-13])
```
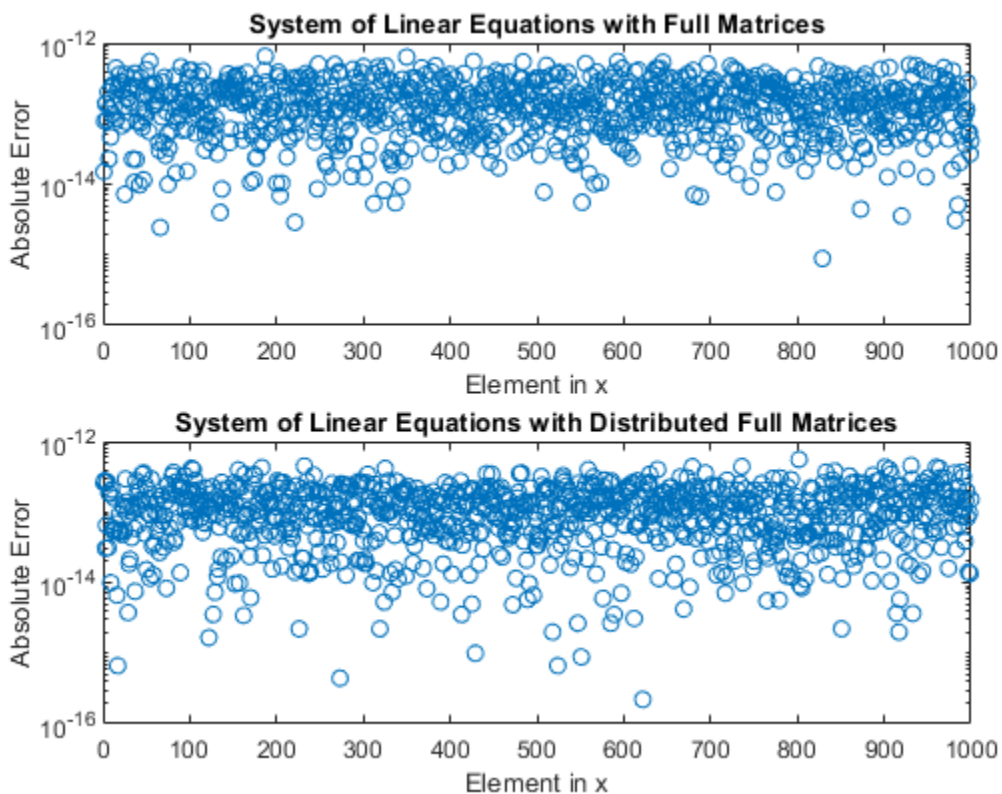
For both the distributed arrays and the arrays stored on the client, the absolute error between the calculated results for *x* and the exact result x_{exact} is small. The accuracy of the solution is approximately the same for both array types.

```
mean(err)
```

```
ans = 1.6031e-13
```

```
mean(errDist)
```

```
ans =

   1.2426e-13
```

Solve a Sparse Matrix System

Distributed arrays can also contain sparse data. To create the coefficient matrix *A*, use `sprand` and `speye` to directly generate a sparse matrix of random numbers plus the sparse identity matrix. Adding the identity matrix helps to prevent creating *A* as a singular or near-singular matrix, both of which are difficult to factorize.

```
n = 1e3;
density = 0.2;
A = sprand(n,n,density) + speye(n);
ADist = distributed(A);
```

Choosing the right hand vector *b* as the row sum of *A* yields an exact solution of the same form as the solution to the full matrix system.

```
b = sum(A,2);
bDist = sum(ADist,2);
xEx = ones(n,1);
xDistEx = ones(n,1,'distributed');
```

In the same way as with full matrices, you can now solve this system of linear equations directly using `mldivide` and check the error between the obtained result and its expected value.

```
x = A\b;
err = abs(xEx-x);

xDist = ADist\bDist;
errDist = abs(xDistEx-xDist);

figure
subplot(2,1,1)
semilogy(err,'o');
title('System of Linear Equations with In-Client Sparse Matrices');
ylabel('Absolute Error');
xlabel('Element in x');
ylim([10e-17,10e-13])
subplot(2,1,2)
semilogy(errDist,'o');
title('System of Linear Equations with Distributed Sparse Matrices');
ylabel('Absolute Error');
xlabel('Element in x');
ylim([10e-17,10e-13])
```
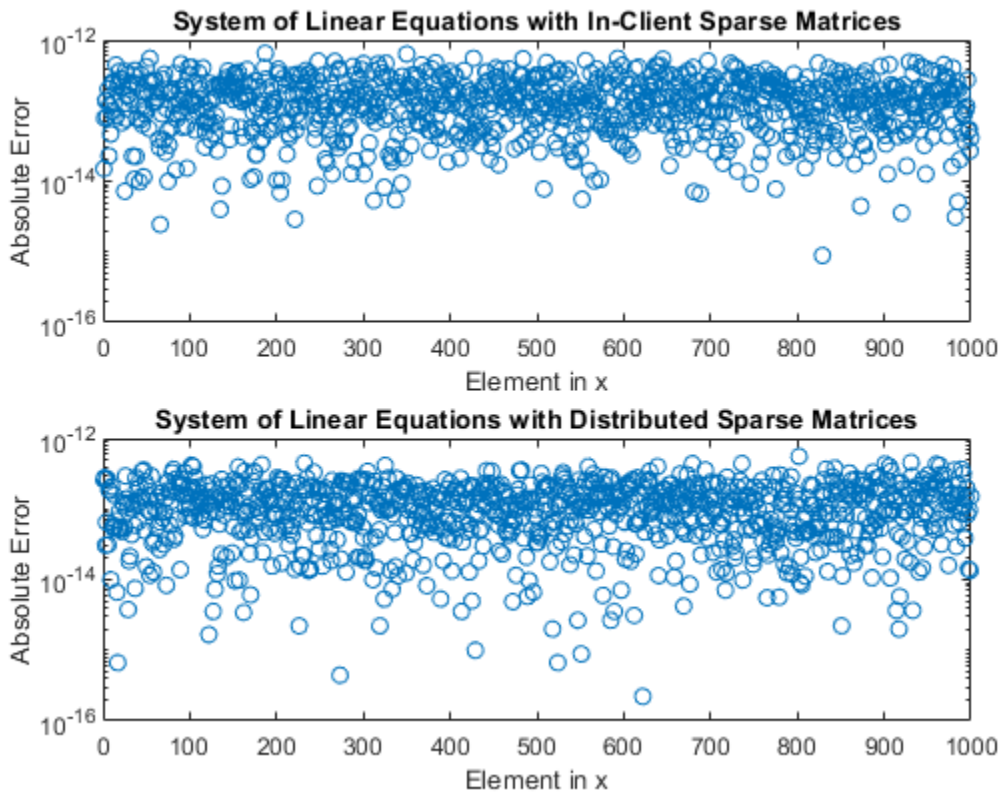
As with the full matrix system, solving the system of linear equations using both on-client arrays and distributed arrays produces solutions with comparable accuracy.

```
mean(err)
```

```
ans = 1.6031e-13
```

```
mean(errDist)
```

```
ans =

   1.2426e-13
```

After you are done with your computations, you can delete your parallel pool. The `gcp` function returns the current parallel pool object so you can delete the current pool.

```
delete(gcp('nocreate'));
```

Improving Efficiency of the Solution

For certain types of large and sparse coefficient matrix A, there are more efficient methods than direct factorization for solving your systems. In these cases, iterative methods might be more efficient at solving your system of linear equations. Iterative methods generate a series of approximate solutions that converge to a final result. For an example of how to use iterative methods to solve

linear equations with large, sparse input matrices, see "Use Distributed Arrays to Solve Systems of Linear Equations with Iterative Methods" on page 9-112.

See Also

`distributed` | `mldivide` | `sparse`

Related Examples

- "Use Distributed Arrays to Solve Systems of Linear Equations with Iterative Methods" on page 9-112

Use Distributed Arrays to Solve Systems of Linear Equations with Iterative Methods

For large-scale mathematical computations, iterative methods can be more efficient than direct methods. This example shows how you can solve systems of linear equations of the form $Ax = b$ in parallel using distributed arrays with iterative methods.

This example continues the topics covered in "Use Distributed Arrays to Solve Systems of Linear Equations with Direct Methods" on page 9-107. The direct solver methods implemented in `mldivide` can be used to solve distributed systems of linear equations in parallel but may not be efficient for certain large and sparse systems. Iterative methods generate a series of solutions from an initial guess, converging to a final result after several steps. These steps can be less computationally intensive than calculating the solution directly.

Distributed arrays distribute data from your client workspace to a parallel pool in your local machine or in a cluster. Each worker stores a portion of the array in its memory, but can also communicate with the other workers to access all segments of the array. Distributed arrays can contain different types of data including full and sparse matrices.

This example uses the `pcg` function to demonstrate how to solve large systems of linear equations using the conjugate gradient and the preconditioned conjugate gradient methods. Iterative methods can be used with both dense and sparse matrices but are most efficient for sparse matrix systems.

Define your System of Linear Equations using a Sparse Matrix

When you use the `distributed` function, MATLAB automatically starts a parallel pool using your default cluster settings. This example uses the Wathen matrix from the MATLAB `gallery` function. This matrix is a sparse, symmetric, and random matrix with overall dimension $N = 3n^2 + 4n + 1$.

```
n = 400;
A = distributed(gallery('wathen',n,n));

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

N = 3*n^2+4*n+1

N = 481601
```

You can now define the right hand vector b. In this example, b is defined as the row sum of A, which leads to an exact solution to $Ax = b$ of the form $x_{\text{exact}} = [1, \ldots, 1]^T$.

```
b = sum(A,2);
```

Since `sum` acts on a distributed array, `b` is also distributed and its data is stored in the memory of the workers of your parallel pool. Finally, you can define the exact solution for comparison with the solutions obtained using iterative methods.

```
xExact = ones(N,1,'distributed');
```

Solve your System of Linear Equations with the Conjugate Gradient Method

The MATLAB function `pcg` provides the conjugate gradient (CG) method, which iteratively generates a series of approximate solutions for x, improving the solution with each step.

```
[xCG_1,flagCG_1,relres_CG1,iterCG_1,resvecCG_1] = pcg(A,b);
```

When the system is solved, you can check the error between each element of the obtained result xCG_1 and the expected values of xExact. The error in the computed result is relatively high.

```
errCG_1 = abs(xExact-xCG_1);
```

```
figure(1)
hold off
semilogy(errCG_1,'o');
title('System of Linear Equations with Sparse Matrix');
ylabel('Absolute Error');
xlabel('Element in x');
```
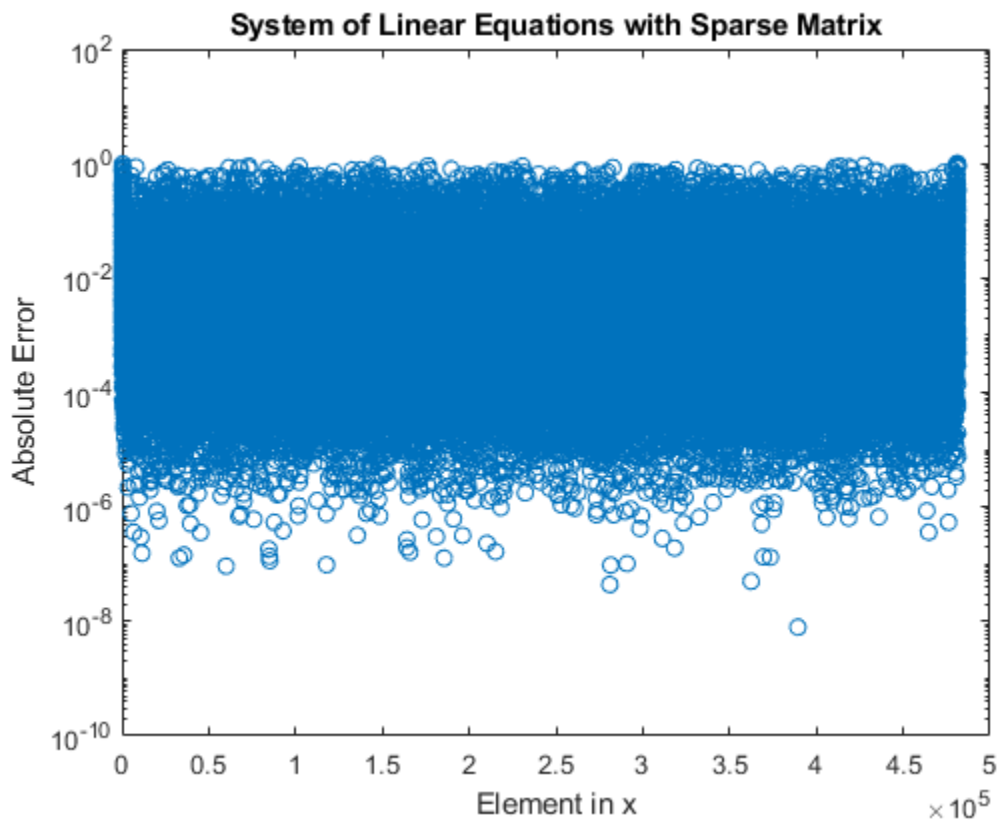

The iterative computation ends when the series of approximate solutions converges to a specific tolerance or after the maximum number of iteration steps. For both distributed and on-client arrays, pcg uses the same default settings:

- The default maximum tolerance is 10^{-6}.
- The default maximum number of iteration steps is 20 or the order of coefficient matrix A if less than 20.

As a second output argument, the pcg function also returns a convergence flag that gives you more information about the obtained result, including whether the computed solution converged to the desired tolerance. For example, a value of 0 indicates the solution has properly converged.

```
flagCG_1
```

```
flagCG_1 = 1
```

In this example, the solution does not converge within the default maximum number of iterations, which results in the high error.

To increase the likelihood of convergence, you can customize the settings for tolerance and maximum number of iteration steps.

```
tolerance = 1e-12;
maxit = N;

tCG = tic;
    [xCG_2,flagCG_2,relresCG_2,iterCG_2,resvecCG_2] = pcg(A,b,tolerance,maxit);
tCG = toc(tCG);

flagCG_2
```

```
flagCG_2 = 0
```

With the custom settings, the solution converges. This solution has an improved absolute error compared to the previous solution.

```
errCG_2 = abs(xExact-xCG_2);
figure(2)
hold off
semilogy(errCG_1,'o');
hold on
semilogy(errCG_2,'d');
title('Comparison of Absolute Error');
ylabel('Absolute Error');
xlabel('Element in x');
legend('Default tolerance and iterations','Improved tolerance and iterations');
hold off
```
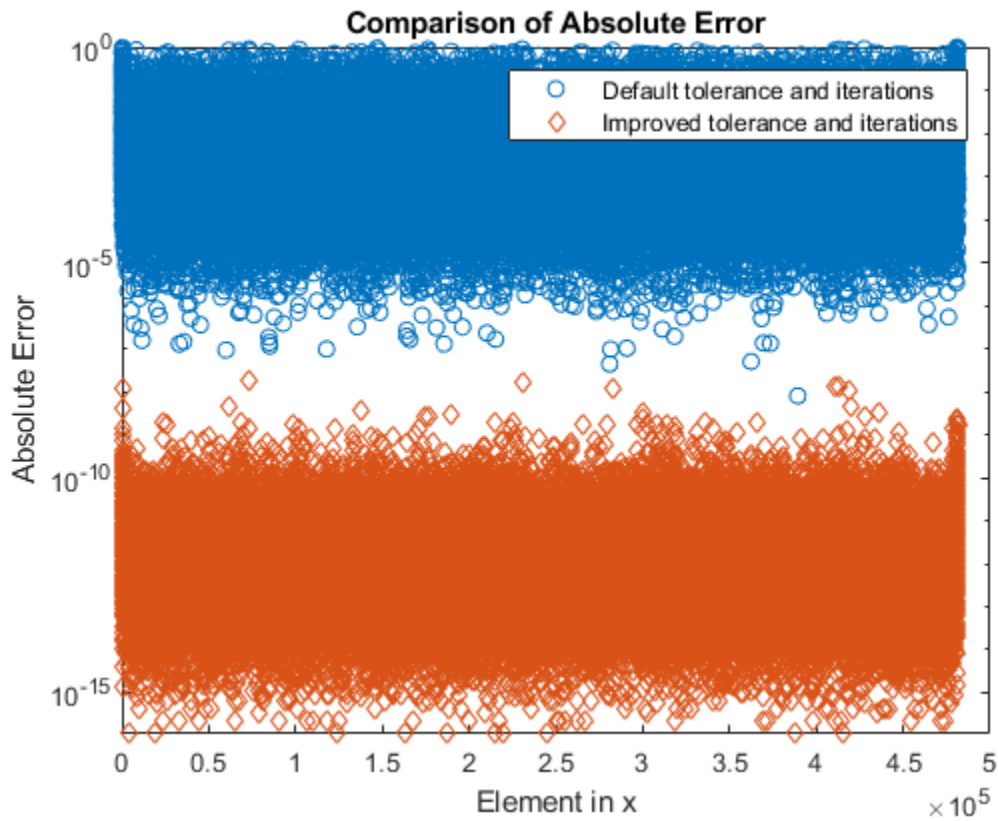
The `pcg` method also returns a vector of the residual norm at each iteration step, `norm(b-A*x)/ norm(b)`. The relative residual norm shows the ratio of accuracies between consecutive iteration steps. The evolution of the residuals during the iterative process can help you understand why the solution did not converge without custom settings.

```
figure(3)
f=semilogy(resvecCG_2./resvecCG_2(1));
hold on
semilogy(f.Parent.XLim,[1e-6 1e-6],'--')
semilogy([20 20], f.Parent.YLim,'--')
semilogy(f.Parent.XLim,[1e-12 1e-12],'--')
title('Evolution of Relative Residual');
ylabel('Relative Residual');
xlabel('Iteration Step');
legend('Residuals of CG','Default Tolerance','Default Number of Steps','Custom Tolerance')
hold off
```
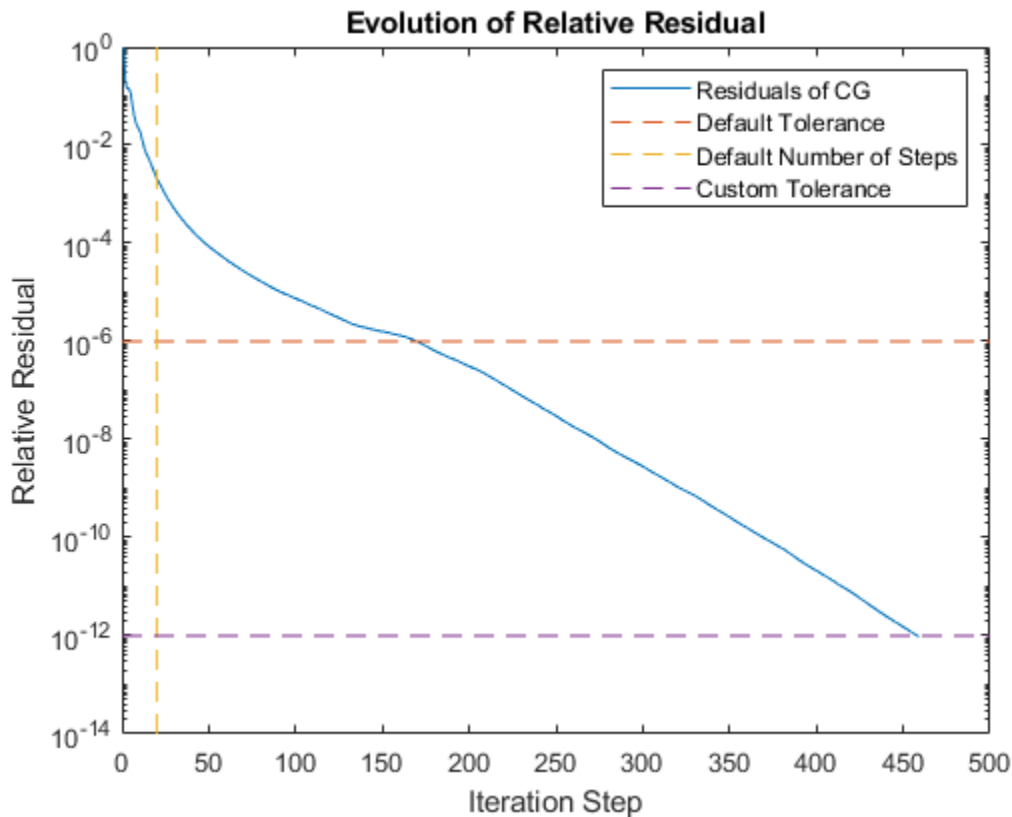
It is clear that the default number of steps is not enough to achieve a good solution for this system.

Solve your System of Linear Equations with the Preconditioned Conjugate Gradient Method

You can improve the efficiency of solving your system using the preconditioned conjugate gradient (PCG) method. First, precondition your system of linear equations using a preconditioner matrix M. Next, solve your preconditioned system using the CG method. The PCG method can take much fewer iterations than the CG method.

The MATLAB function `pcg` is also used for the PCG method. You can supply a suitable preconditioner matrix M as an additional input.

An ideal preconditioner matrix is a matrix whose inverse M^{-1} is a close approximation to the inverse of the coefficient matrix, A^{-1}, but is easier to compute. This example uses the diagonal of A to precondition the system of linear equations.

```
M = spdiags(spdiags(A,0),0,N,N);
tPCG = tic;
    [xPCG,flagPCG,relresPCG,iterPCG,resvecPCG]=pcg(A,b,tolerance,maxit,M);
tPCG = toc(tPCG);

figure(4)
hold off;
semilogy(resvecCG_2./resvecCG_2(1))
hold on;
semilogy(resvecPCG./resvecPCG(1))
```

```matlab
title('Evolution of Relative Residual');
ylabel('Relative Residual');
xlabel('Iteration Step');
legend('Residuals of CG','Residuals of PCG with M \approx diag(A)')
```
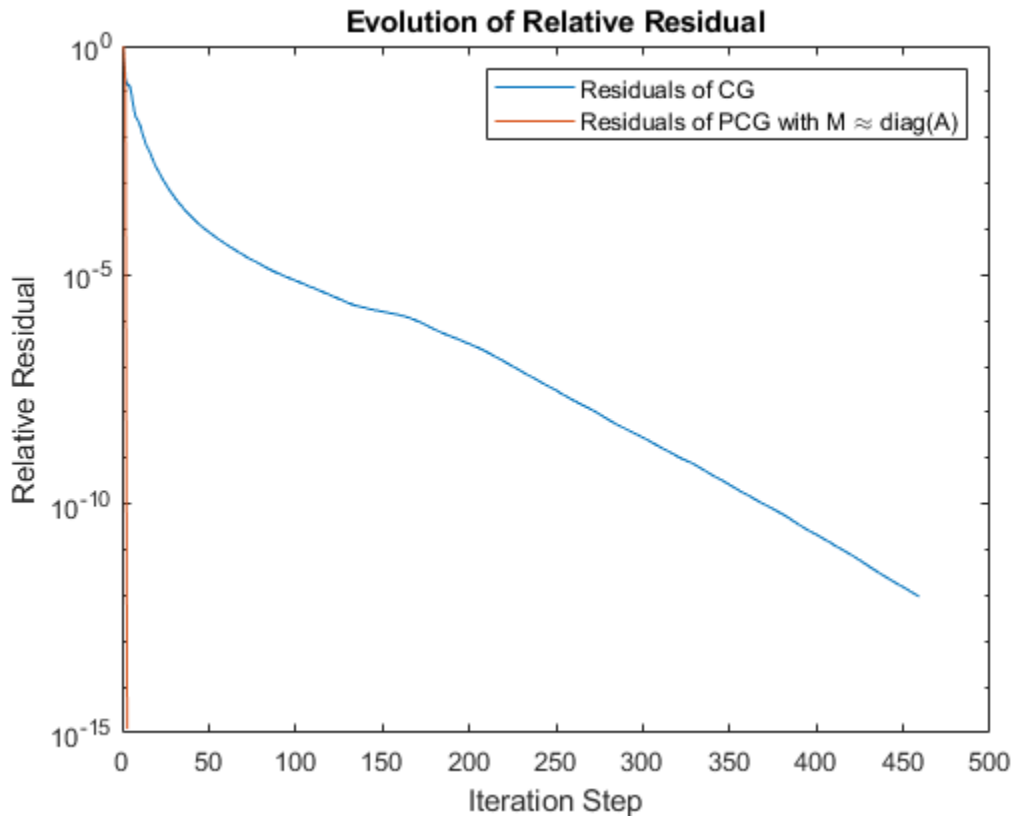

The previous figure shows that the PCG method needs drastically fewer steps to converge compared to the nonpreconditioned system. This result is also reflected in the execution times.

```matlab
fprintf([...
    '\nTime to solve system with CG:  %d s', ...
    '\nTime to solve system with PCG: %d s'],tCG,tPCG);
```

```
Time to solve system with CG:  1.244593e+01 s
Time to solve system with PCG: 7.657432e-01 s
```

As well as solving this example system in fewer iteration steps, the PCG method also returns a more accurate solution.

```matlab
errPCG = abs(xExact-xPCG);
figure(5)
hold off
semilogy(errCG_1,'o');
hold on
semilogy(errCG_2,'d');
semilogy(errPCG,'x');
title('Comparison of absolute error');
ylabel('Absolute error');
```

```
xlabel('Element in x');
legend('CG default','CG custom','PCG');
```
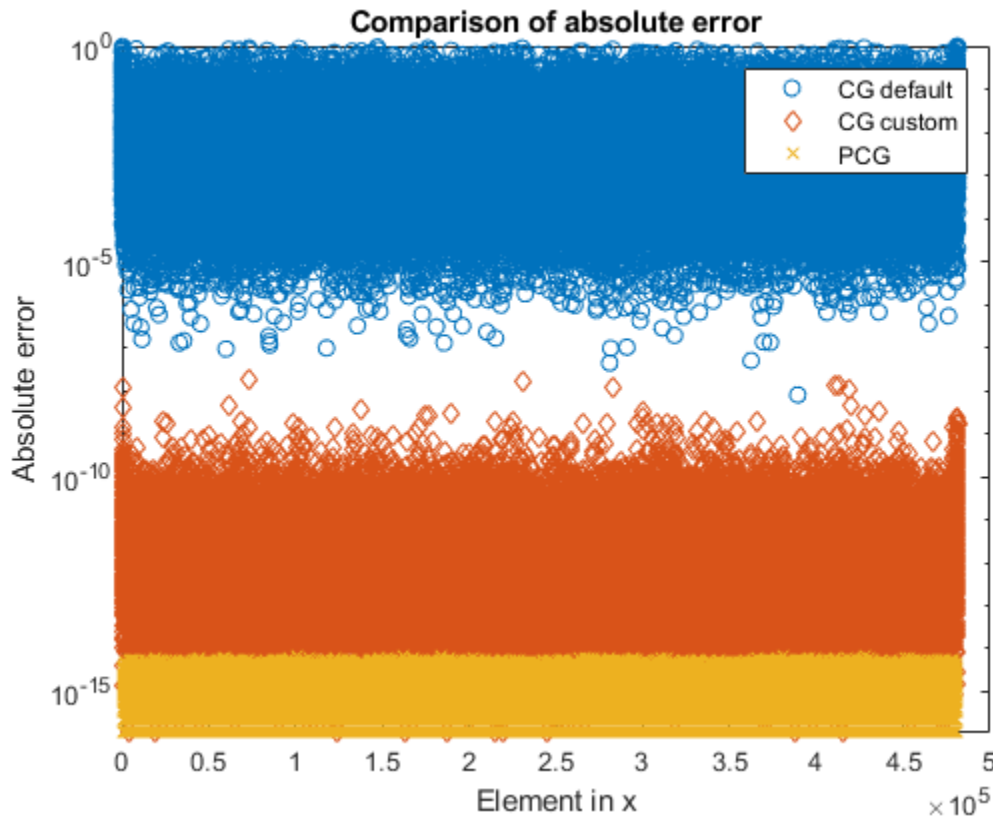

After you are done with your computations, you can delete your parallel pool. The `gcp` function returns the current parallel pool object so you can delete the current pool.

```
delete(gcp('nocreate'))
```

The Wathen matrix used in this example is a good demonstration of how a good preconditioner can dramatically improve the efficiency of the solution. The Wathen matrix has relatively small off-diagonal components, so choosing $M = \text{diag}(A)$ gives a suitable preconditioner. For an arbitrary matrix A, finding a preconditioner might not be so straightforward.

For an example of how to approximate a differential equation by a linear system and solve it using a distributed iterative solver with a multigrid preconditioner, see "Solve Differential Equation Using Multigrid Preconditioner on Distributed Discretization" on page 9-13.

See Also
`distributed` | `pcg` | `sparse`

Related Examples
- "Use Distributed Arrays to Solve Systems of Linear Equations with Direct Methods" on page 9-107

Using GOP to Achieve MPI_Allreduce Functionality

In this example, we look at the `gop` function and the functions that build on it: `gplus` and `gcat`. These seemingly simple functions turn out to be very powerful tools in parallel programming.

The `gop` function allows us to perform any associative binary operation on a variable that is defined on all the labs. This allows us not only to sum a variable across all the labs, but also to find its minimum and maximum across the labs, concatenate them, and perform many other useful operations.

Related Documentation:

- `spmd` in the Parallel Computing Toolbox™ User's Guide

The code shown in this example can be found in this function:

```
function paralleltutorial_gop
```

```
%#ok<*NOPRT>: disable code analyzer printing warning
```

Introduction

When doing parallel programming, we often run into the situation of having a variable defined on all the labs, and we want to perform an operation on the variable as it exists on all the labs. For example, if we enter an spmd statement and define

```
spmd
    x = labindex
end
```

```
Lab  1:

  x =

      1

Lab  2:

  x =

      2

Lab  3:

  x =

      3

Lab  4:

  x =

      4

Lab  5:

  x =
```

```
         5
Lab   6:
   x =
         6
Lab   7:
   x =
         7
Lab   8:
   x =
         8
Lab   9:
   x =
         9
Lab 10:
   x =
        10
Lab 11:
   x =
        11
Lab 12:
   x =
        12
```

on all the labs, we might want to calculate the sum of the values of x across the labs. This is exactly what the `gplus` operation does, it sums the x across the labs and duplicates the result on all labs:

```
spmd
    s = gplus(x);
end
```

The variables assigned to inside an spmd statement are represented on the client as Composite. We can bring the resulting values from the labs to the client by indexing into the Composite much like that of cell arrays:

```
s{1} % Display the value of s on lab 1.  All labs store the same value.
```

```
ans =

     78
```

Also, `gop`, `gplus`, and `gcat` allow us to specify a single lab to which the function output should be returned, and they return an empty vector on the other labs.

```
spmd
    s = gplus(x, 1);
end
s{1}
```

```
ans =

     78
```

This example shows how to perform a host of operations similar to addition across all the labs. In MPI, these are known as collective operations, such as MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX, etc.

Create the Input Data for Our Examples

The data we use for all our examples is very simple: a 1-by-2 variant array that is only slightly more complicated than the `x` we defined in the beginning:

```
spmd
    x = labindex + (1:2)
end
```

```
Lab   1:

  x =

       2     3

Lab   2:

  x =

       3     4

Lab   3:

  x =

       4     5

Lab   4:

  x =

       5     6

Lab   5:
```

```
    x =

          6      7

Lab   6:

    x =

          7      8

Lab   7:

    x =

          8      9

Lab   8:

    x =

          9     10

Lab   9:

    x =

         10     11

Lab 10:

    x =

         11     12

Lab 11:

    x =

         12     13

Lab 12:

    x =

         13     14
```

Using GPLUS and GCAT

Now that we have initialized our vector x to different values on the labs, we can ask questions such as what is the element-by-element sum of the values of x across the labs? What about the product, the minimum, and the maximum? As to be expected from our introduction,

```
spmd
    s = gplus(x);
end
s{1}
```

```
ans =

    90   102
```

returns the element-by-element addition of the values of x. However, `gplus` is only a special case of the `gop` operation, short for Global OPeration. The `gop` function allows us to perform any associative operation across the labs on the elements of a variant array. The most basic example of an associative operation is addition; it is associative because addition is independent of the grouping which is used:

```
(a + b) + c = a + (b + c)
```

In MATLAB®, addition can be denoted by the `@plus` function handle, so we can also write `gplus(x)` as

```
spmd
    s = gop(@plus, x);
end
s{1}
```

```
ans =

    90   102
```

We can concatenate the vector x across the labs by using the `gcat` function, and we can choose the dimension to concatenate along.

```
spmd
    y1 = gcat(x, 1); % Concatenate along rows.
    y2 = gcat(x, 2); % Concatenate along columns.
end
y1{1}
y2{1}
```

```
ans =

     2     3
     3     4
     4     5
     5     6
     6     7
     7     8
     8     9
     9    10
    10    11
    11    12
    12    13
    13    14
```

```
ans =

  Columns 1 through 13
```

```
    2    3    3    4    4    5    5    6    6    7    7    8    8

  Columns 14 through 24

    9    9   10   10   11   11   12   12   13   13   14
```

Other Elementary Uses of GOP

It is simple to calculate the element-by-element product of the values of x across the labs:

```
spmd
    p = gop(@times, x);
end
p{1}

ans =

   1.0e+10 *

   0.6227    4.3589
```

We can also find the element-by-element maximum of x across the labs:

```
spmd
    M = gop(@max, x);
    m = gop(@min, x);
end
M{1}
m{1}

ans =

    13    14

ans =

     2     3
```

Logical Operations

MATLAB has even more built-in associative operations. The logical AND, OR, and XOR operations are represented by the @and, @or, and @xor function handles. For example, look at the logical array

```
spmd
    y = (x > 4)
end

Lab  1:

  y =

     0     0
```

```
Lab  2:

  y =

        0      0

Lab  3:

  y =

        0      1

Lab  4:

  y =

        1      1

Lab  5:

  y =

        1      1

Lab  6:

  y =

        1      1

Lab  7:

  y =

        1      1

Lab  8:

  y =

        1      1

Lab  9:

  y =

        1      1

Lab 10:

  y =

        1      1

Lab 11:

  y =
```

```
        1      1
```

```
Lab 12:

  y =

        1      1
```

We can then easily perform these logical operations on the elements of y across the labs:

```
spmd
    yand = gop(@and, y);
    yor = gop(@or, y);
    yxor = gop(@xor, y);
end
yand{1}
yor{1}
yxor{1}
```

```
ans =

     0     0
```

```
ans =

     1     1
```

```
ans =

     1     0
```

Bitwise Operations

To conclude our tour of the associative operations that are built into MATLAB, we look at the bitwise AND, OR, and XOR operations. These are represented by the `@bitand`, `@bitor`, and `@bitxor` function handles.

```
spmd
    xbitand = gop(@bitand, x);
    xbitor = gop(@bitor, x);
    xbitxor = gop(@bitxor,  x);
end
xbitand{1}
xbitor{1}
xbitxor{1}
```

```
ans =

     0     0
```

```
ans =
```

```
     15      15


ans =

      0      12
```

Finding Locations of Min and Max

We need to do just a little bit of programming to find the labindex corresponding to where the element-by-element maximum of x across the labs occurs. We can do this in just a few lines of code:

type pctdemo_aux_gop_maxloc

```
function [val, loc] = pctdemo_aux_gop_maxloc(inval)
%PCTDEMO_AUX_GOP_MAXLOC Find maximum value of a variant and its labindex.
%   [val, loc] = pctdemo_aux_gop_maxloc(inval) returns to val the maximum value
%   of inval across all the labs.  The labindex where this maximum value
%   resides is returned to loc.

%   Copyright 2007 The MathWorks, Inc.

    out = gop(@iMaxLoc, {inval, labindex*ones(size(inval))});
    val = out{1};
    loc = out{2};
end

function out = iMaxLoc(in1, in2)
% Calculate the max values and their locations.  Return them as a cell array.
    in1Largest = (in1{1} >= in2{1});
    maxVal = in1{1};
    maxVal(~in1Largest) = in2{1}(~in1Largest);
    maxLoc = in1{2};
    maxLoc(~in1Largest) = in2{2}(~in1Largest);
    out = {maxVal, maxLoc};
end
```

and when the function has been implemented, it can be applied just as easily as any of the built-in operations:

```
spmd
    [maxval, maxloc] = pctdemo_aux_gop_maxloc(x);
end
[maxval{1}, maxloc{1}]


ans =

     13      14      12      12
```

Similarly, we only need a few lines of code to find the labindex where the element-by-element minimum of x across the labs occurs:

type pctdemo_aux_gop_minloc

```
function [val, loc] = pctdemo_aux_gop_minloc(inval)
%PCTDEMO_AUX_GOP_MINLOC Find minimum value of a variant and its labindex.
%   [val, loc] = pctdemo_aux_gop_minloc(inval) returns to val the minimum value
%   of inval across all the labs.  The labindex where this minimum value
%   resides is returned to loc.

%   Copyright 2007 The MathWorks, Inc.

    out = gop(@iMinLoc, {inval, labindex*ones(size(inval))});
    val = out{1};
    loc = out{2};
end

function out = iMinLoc(in1, in2)
% Calculate the min values and their locations.  Return them as a cell array.
    in1Smallest = (in1{1} < in2{1});
    minVal = in1{1};
    minVal(~in1Smallest) = in2{1}(~in1Smallest);
    minLoc = in1{2};
    minLoc(~in1Smallest) = in2{2}(~in1Smallest);
    out = {minVal, minLoc};
end
```

We can then easily find the minimum with `gop`:

```
spmd
    [minval, minloc] = pctdemo_aux_gop_minloc(x);
end
[minval{1}, minloc{1}]


ans =

    2    3    1    1
```

Resource Contention in Task Parallel Problems

This example looks at why it is so hard to give a concrete answer to the question "How will my (parallel) application perform on my multi-core machine or on my cluster?" The answer most commonly given is "It depends on your application as well as your hardware," and we will try to explain why this is all one can say without more information.

This example shows the contention for memory access that we can run into on a regular multi-core computer. This example illustrates the case of all the workers running on the same multi-core computer with only one CPU.

Related examples:

- "Benchmarking Independent Jobs on the Cluster" on page 9-137
- "Simple Benchmarking of PARFOR Using Blackjack" on page 9-102

To simplify the problem at hand, we will benchmark the computer's ability to execute task parallel problems that do not involve disk IO. This allows us to ignore several factors that might affect parallel applications, such as:

- Amount of inter-process communication
- Network bandwidth and latency for inter-process communication
- Process startup and shutdown times
- Time to dispatch requests to the processes
- Disk IO performance

This leaves us with only:

- Time spent executing task parallel code

Analogy for Resource Contention and Efficiency

To understand why it is worthwhile to perform such a simple benchmark, consider the following example: If one person can fill one bucket of water, carry it some distance, empty it, and take it back to refill it in one minute, how long will it take two people to go the same round trip with one bucket each? This simple analogy closely reflects the task parallel benchmarks in this example. At first glance, it seems absurd that there should be any decrease in efficiency when two people are simultaneously doing the same thing as compared to one person.

Competing for One Pipe: Resource Contention

If all things are perfect in our previous example, two people complete one loop with a bucket of water each in one minute. Each person quickly fills one bucket, carries the bucket over to the destination, empties it and walks back, and they make sure never to interfere or interrupt one another.

However, imagine that they have to fill the buckets from a single, small water hose. If they arrive at the hose at the same time, one would have to wait. This is one example of a **contention for a shared resource**. Maybe the two people don't need to simultaneously use the hose, and the hose therefore serves their needs; but if you have 10 people transporting a bucket each, some might always have to wait.

In our case, the water hose corresponds to the computer hardware that we use, in particular the access to the computer memory. If multiple programs are running simultaneously on one CPU core

each, and they all need access to data that is stored in the computer's memory, some of the programs may have to wait because of limited memory bandwidth.

Same Hose, Different Distance, Different Results

To take our analogy a bit further, imagine that we have contention at the hose when two people are carrying one bucket each, but then we change the task and ask them to carry the water quite a bit further away from the hose. When performing this modified task, the two people spend a larger proportion of their time doing work, i.e., walking with the buckets, and a smaller proportion of their time contending over the shared resource, the hose. They are therefore less likely to need the hose at the same time, so this modified task has a higher **parallel efficiency** than the original one.

In the case of the benchmarks in this example, this corresponds on the one hand to running programs that require lots of access to the computer's memory, but they perform very little work with the data once fetched. If, on the other hand, the programs perform lots of computations with the data, it becomes irrelevant how long it took to fetch the data, the computation time will overshadow the time spent waiting for access to the memory.

The predictability of the memory access of an algorithm also effects how contended the memory access will be. If the memory is accessed in a regular, predictable manner, we will not experience the same amount of contention as when memory is accessed in an irregular manner. This can be seen further below, where, for example, singular value decomposition calculations result in more contention than matrix multiplication.

The code shown in this example can be found in this function:

```
function paralleldemo_resource_bench
```

Check the Status of the parallel pool

We will use the parallel pool to call our task parallel test functions, so we start by checking whether the pool is open. Note that if the parallel pool is using workers running on multiple computers, you may not experience any of the resource contention that we attempt to illustrate.

```
p = gcp;
if isempty(p)
    error('pctexample:backslashbench:poolClosed', ...
        ['This example requires a parallel pool. ' ...
         'Manually start a pool using the parpool command or set ' ...
         'your parallel preferences to automatically start a pool.']);
end
poolSize = p.NumWorkers;
```

Set up Benchmarking Problem

For our calculations, we create an input matrix large enough that it needs to be brought from the computer's memory onto the CPU each time it is processed. That is, we make it so large that we deliberately cause resource contention.

```
sz = 2048;
m = rand(sz*sz, 1);
```

Repeated Summation

These computations are very simple: Repeatedly sum a single array. Since the computations are very lightweight, we expect to see resource contention when running multiple copies of this function simultaneously with a large input array.

```matlab
function sumOp(m)
    s = 0;
    for itr = 1:100 % Repeat multiple times to get accurate timing
        s = s + sum(m);
    end
end
```

The Benchmarking Function

The core of the timing function consists of a simple `spmd` statement. Notice that we retain the minimum execution time observed for a given level of concurrency, n. As stated at the beginning, we are benchmarking task parallel problems, measuring only the actual runtime. This means that we are not benchmarking the performance of MATLAB, the Parallel Computing Toolbox™, or the `spmd` language construct. Rather, we are benchmarking the ability of our OS and hardware to simultaneously run multiple copies of a program.

```matlab
function time = timingFcn(fcn, numConcurrent)

    time = zeros(1, length(numConcurrent));

    for ind = 1:length(numConcurrent)
        % Invoke the function handle fcn concurrently on n different labs.
        % Store the minimum time it takes all to complete.
        n = numConcurrent(ind);
        spmd(n)
            tconcurrent = inf;
            for itr = 1:5
                labBarrier;
                tic; % Measure only task parallel runtime.
                    fcn();
                tAllDone = gop(@max, toc);  % Time for all to complete.
                tconcurrent = min(tconcurrent, tAllDone);
            end
        end
        time(ind) = tconcurrent{1};
        clear tconcurrent itr tAllDone;
        if ind == 1
            fprintf('Execution times: %f', time(ind));
        else
            fprintf(', %f', time(ind));
        end
    end
    fprintf('\n');
end
```

Benchmarking the Summation

We measure how long it takes to simultaneously evaluate n copies of a summation function for different values of n. Since the summation calculations are so simple, we expect to encounter a resource contention when running multiples of these computations simultaneously on a multi-core CPU. Consequently, we expect it to take longer to evaluate the summation function when we are performing multiple such evaluations concurrently than it takes to execute a single such evaluation on an otherwise idle CPU.

```matlab
tsum = timingFcn(@() sumOp(m), 1:poolSize);
```

```
Execution times: 0.367254, 0.381174, 0.395128, 0.421978
```

Benchmarking FFT

We now look at a more computationally intensive problem, that of calculating the FFT of our vector. Since FFT is more computationally intensive than summation, we expect that we will not see the same performance degradations when concurrently evaluating multiple calls to the FFT function as we saw with calls to the summation function.

```
function fftOp(m)
    for itr = 1:10 % Repeat a few times for accurate timing
        fft(m);
    end
end

tfft = timingFcn(@() fftOp(m), 1:poolSize);
```

Execution times: 1.078532, 1.196302, 1.358666, 1.570749

Matrix Multiplication

Finally, we look at an even more computationally intensive problem than either FFT or summation. The memory access in matrix multiplication is also very regular, so this problem therefore has the potential to be executed quite efficiently in parallel on a multi-core machine.

```
function multOp(m)
    m*m;   %#ok<VUNUS> % No need to repeat for accurate timing.
end
m = reshape(m, sz, sz);
tmtimes = timingFcn(@() multOp(m), 1:poolSize);
clear m;
```

Execution times: 0.667003, 0.768181, 0.846016, 0.989242

Investigate the Resource Contention

We create a simple bar chart showing the speedup achieved by running multiple function invocations concurrently. This kind of graph shows the speedup with what is known as **weak scaling**. Weak scaling is where the number of processes/processors varies, and the problem size on each process/processor is fixed. This has the effect of increasing the total problem size as we increase the number of processes/processors. On the other hand, **strong scaling** is where the problem size is fixed and the number of processes/processors varies. The effect of this is that as we increase the number of processes/processors, the work done by each process/processor decreases.

```
allTimes = [tsum(:), tfft(:), tmtimes(:)];
% Normalize the execution times.
efficiency = bsxfun(@rdivide, allTimes(1, :), allTimes);
speedup = bsxfun(@times, efficiency, (1:length(tsum))');
fig = figure;
ax = axes('parent', fig);
bar(ax, speedup);
legend(ax, 'vector sum', 'vector fft', 'matrix mult', ...
        'Location', 'NorthWest')
xlabel(ax, 'Number of concurrent processes');
ylabel(ax, 'Speedup')
title(ax, ['Effect of number of concurrent processes on ', ...
            'resource contention and speedup']);
```

Effect of number of concurrent processes on resource contention and speedu
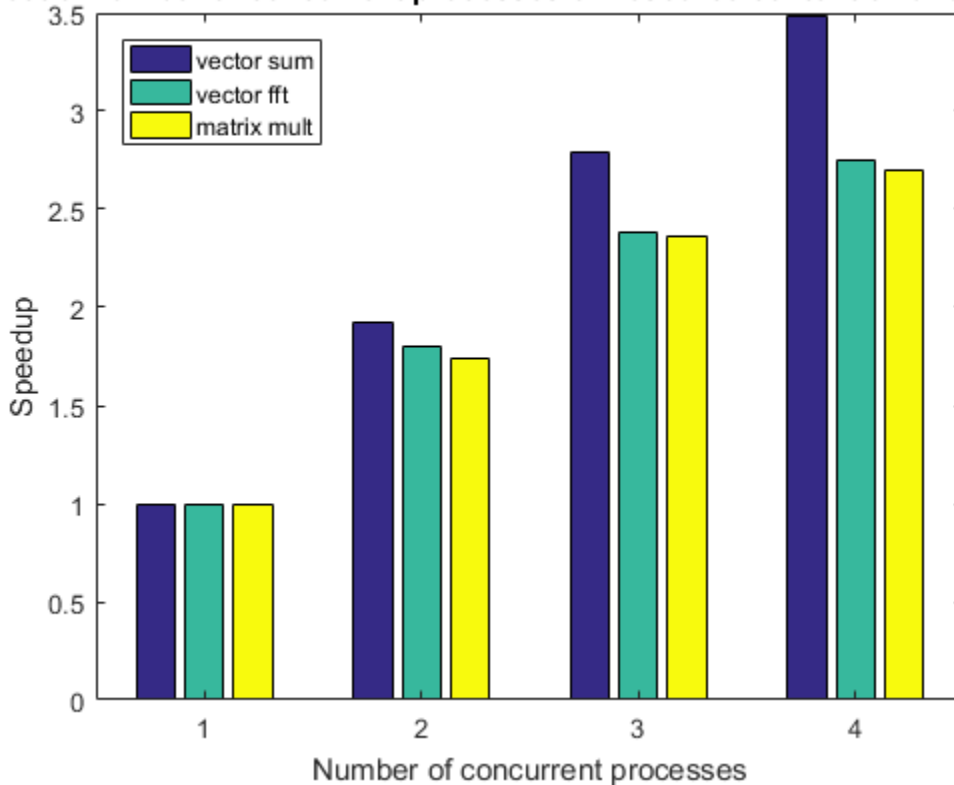
Effect on Real Applications

Looking at the graph above, we can see that these simple problems scale very differently on the same computer. When we also look at the fact that other problems and different computers may show very different behavior, it should become clear why it is impossible to give a general answer to the question "How will my (parallel) application perform on my multi-core machine or on my cluster?" The answer to that question truly depends on the application and the hardware in question.

Measure Effect of Data Size on Resource Contention

The resource contention does not depend only on the function that we are executing, but also on the size of the data that we process. To illustrate this, we measure the execution times of various functions with various sizes of input data. As before, we are benchmarking the ability of our hardware to perform these computations concurrently, and we are not benchmarking MATLAB or its algorithms. We use more functions than before so that we can investigate the effects of different memory access patterns as well as the effects of different data sizes.

```
szs = [128, 256, 512, 1024, 2048];
description = {'vector sum', 'vector fft', 'matrix mult', 'matrix LU', ...
    'matrix SVD', 'matrix eig'};
```

We loop through the different data sizes and the functions we want to benchmark, and measure the speedup observed. We compare the sequential execution time to the time it takes to execute concurrently as many invocations as we have labs in the pool.

```
speedup = zeros(length(szs), length(description));
for i = 1:length(szs)
```

```matlab
        sz = szs(i);
        fprintf('Using matrices of size %d-by-%d.\n', sz, sz);
        j = 1;
        for f = [{@sumOp; sz^2; 1}, {@fftOp; sz^2; 1}, {@multOp; sz; sz}, ...
                {@lu; sz; sz}, {@svd;  sz; sz}, {@eig;  sz; sz}]
            op = f{1};
            nrows = f{2};
            ncols = f{3};
            m = rand(nrows, ncols);
            % Compare sequential execution to execution on all labs.
            tcurr = timingFcn(@() op(m), [1, poolSize]);
            speedup(i, j) = tcurr(1)/tcurr(2)*poolSize;
            j = j + 1;
        end
    end
end

Using matrices of size 128-by-128.
Execution times: 0.001472, 0.001721
Execution times: 0.002756, 0.004069
Execution times: 0.000221, 0.000367
Execution times: 0.000200, 0.000369
Execution times: 0.001314, 0.002186
Execution times: 0.006565, 0.009958
Using matrices of size 256-by-256.
Execution times: 0.005472, 0.005629
Execution times: 0.010400, 0.013956
Execution times: 0.002175, 0.002994
Execution times: 0.000801, 0.001370
Execution times: 0.008052, 0.009112
Execution times: 0.042912, 0.057383
Using matrices of size 512-by-512.
Execution times: 0.021890, 0.022754
Execution times: 0.055563, 0.083730
Execution times: 0.011029, 0.017932
Execution times: 0.005655, 0.009090
Execution times: 0.052226, 0.067276
Execution times: 0.262720, 0.353336
Using matrices of size 1024-by-1024.
Execution times: 0.090294, 0.110154
Execution times: 0.317712, 0.445605
Execution times: 0.097819, 0.131056
Execution times: 0.037662, 0.057474
Execution times: 0.392037, 0.937005
Execution times: 1.063107, 1.579232
Using matrices of size 2048-by-2048.
Execution times: 0.365510, 0.422942
Execution times: 1.067788, 1.560758
Execution times: 0.667548, 0.980306
Execution times: 0.271354, 0.391217
Execution times: 4.111523, 7.820437
Execution times: 6.101292, 8.984251
```

Effect of number of concurrent processes on resource contention and speedup
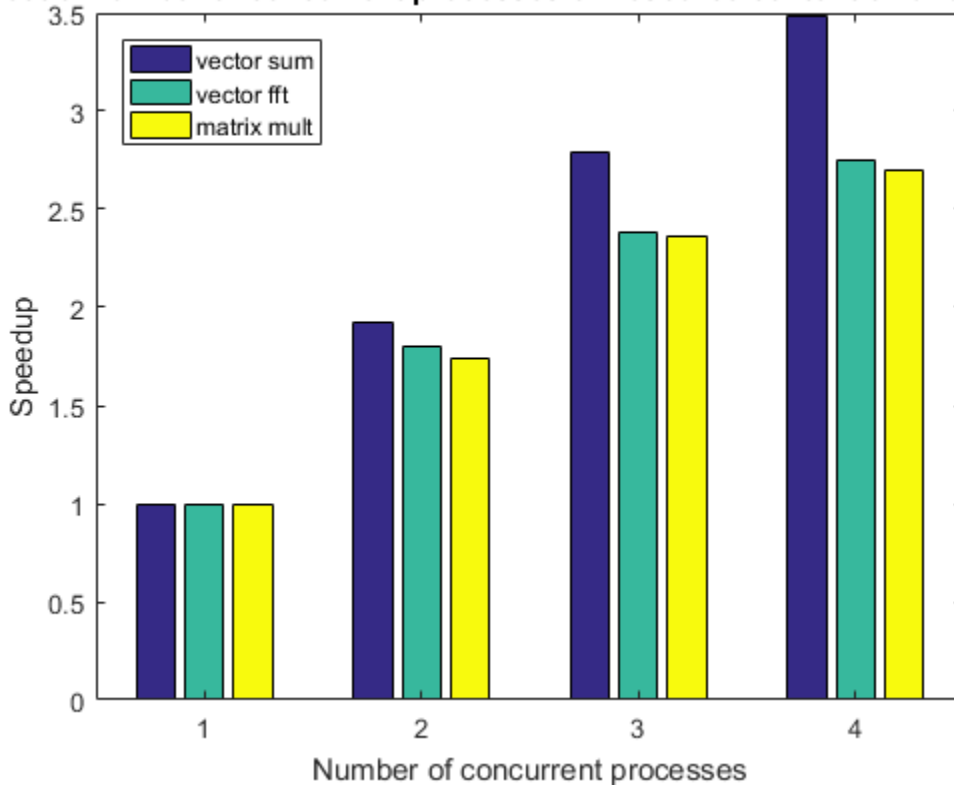
View Resource Contention As a Function of Data Size

When we look at the results, we have to keep in mind how a function interacts with the cache on a CPU. For small data sizes, we are always working out of the CPU cache for all these functions. In that case, we expect to see perfect speedup. When the input data is too large to fit into the CPU cache, we start seeing the performance degradation caused by contention for memory access. This happens in several ways, but for this example, the following are the most important:

- The function performs a relatively small amount of computation with the data. The sum of a vector is a good example of that.
- The function accesses data in small chunks or has irregular data access patterns. The following graph shows that with eigenvalue calculations (eig) and singular value decomposition (svd).

```
fig = figure;
ax = axes('parent', fig);
plot(ax, speedup);
lines = ax.Children;
set(lines, {'Marker'}, {'+', 'o', '*', 'x', 's', 'd'}');
hold(ax, 'on');
plot(ax, repmat(poolSize, 1, length(szs)), '--');
hold(ax, 'off');
legend(ax, [description, {'ideal speedup'}], 'Location', 'SouthWest');
ax.XTick = 1:length(szs);
ax.XTickLabel = arrayfun(@(x) {sprintf('%d^2', x)}, szs);
yl = ylim;
ylim([0, max([poolSize + 0.1, yl(2)])])
xlabel(ax, 'Number of elements per process');
```

9-135

```
ylabel(ax, 'Speedup')
title(ax, 'Effect of data size on resource contention and speedup');
```

Effect of data size on resource contention and speedup
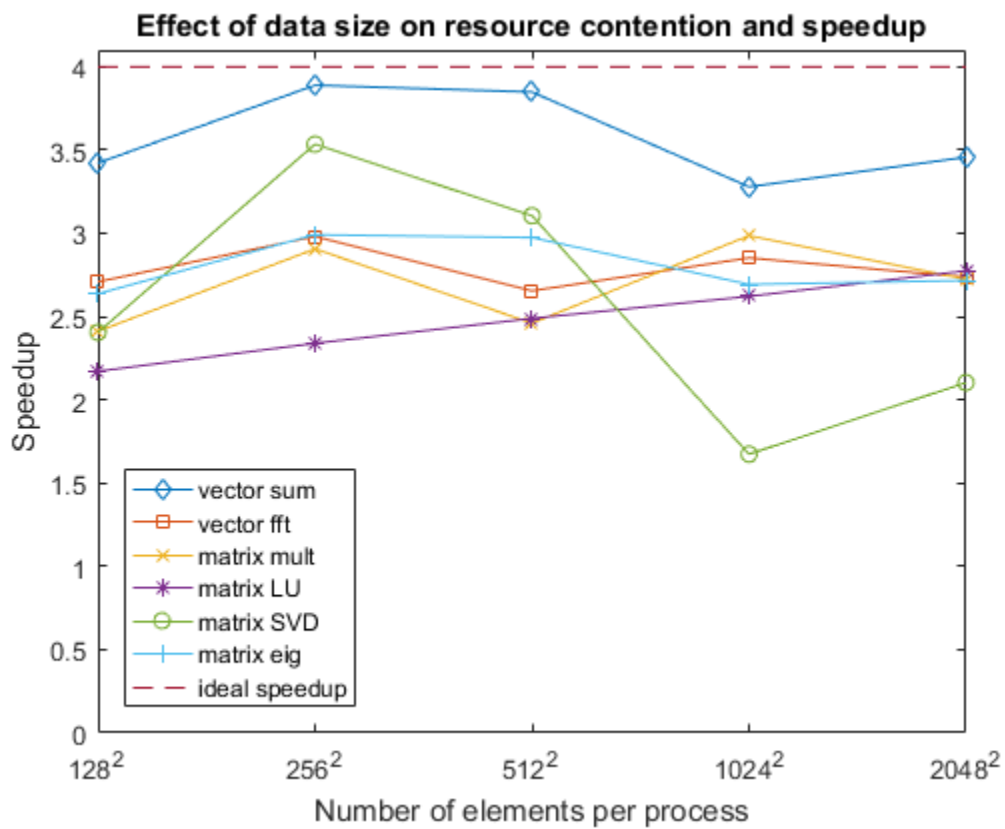

```
end
```

Benchmarking Independent Jobs on the Cluster

In this example, we show how to benchmark an application using independent jobs on the cluster, and we analyze the results in some detail. In particular, we:

- Show how to benchmark a mixture of sequential code and task parallel code.
- Explain strong and weak scaling.
- Discuss some of the potential bottlenecks, both on the client and on the cluster.

Note: If you run this example on a large cluster, it might take an hour to run.

Related examples:

- "Resource Contention in Task Parallel Problems" on page 9-129
- "Simple Benchmarking of PARFOR Using Blackjack" on page 9-102

The code shown in this example can be found in this function:

```
function paralleldemo_distribjob_bench
```

Check the Cluster Profile

Before we interact with the cluster, we verify that the MATLAB® client is configured according to our needs. Calling `parcluster` will give us a cluster using the default profile or will throw an error if the default is not usable.

```
myCluster = parcluster;
```

Timing

We time all operations separately to allow us to inspect them in detail. We will need all those detailed timings to understand where the time is spent, and to isolate the potential bottlenecks. For the purposes of the example, the actual function we benchmark is not very important; in this case we simulate hands of the card game blackjack or 21.

We write all of the operations to be as efficient as possible. For example, we use vectorized task creation. We use `tic` and `toc` for measuring the elapsed time of all the operations instead of using the job and task properties `CreateTime`, `StartTime`, `FinishTime`, etc., because `tic` and `toc` give us sub-second granularity. Note that we have also instrumented the task function so that it returns the time spent executing our benchmark computations.

```
function [times, description] = timeJob(myCluster, numTasks, numHands)
    % The code that creates the job and its tasks executes sequentially in
    % the MATLAB client starts here.
    % We first measure how long it takes to create a job.
    timingStart = tic;
    start = tic;
    job = createJob(myCluster);
    times.jobCreateTime = toc(start);
    description.jobCreateTime = 'Job creation time';

    % Create all the tasks in one call to createTask, and measure how long
    % that takes.
    start = tic;
    taskArgs = repmat({{numHands, 1}}, numTasks, 1);
```

```matlab
    createTask(job, @pctdemo_task_blackjack, 2, taskArgs);
    times.taskCreateTime = toc(start);
    description.taskCreateTime = 'Task creation time';

    % Measure how long it takes to submit the job to the cluster.
    start = tic;
    submit(job);
    times.submitTime = toc(start);
    description.submitTime = 'Job submission time';

    % Once the job has been submitted, we hope all its tasks execute in
    % parallel.  We measure how long it takes for all the tasks to start
    % and to run to completion.
    start = tic;
    wait(job);
    times.jobWaitTime = toc(start);
    description.jobWaitTime = 'Job wait time';

    % Tasks have now completed, so we are again executing sequential code
    % in the MATLAB client.  We measure how long it takes to retrieve all
    % the job results.
    start = tic;
    results = fetchOutputs(job);
    times.resultsTime = toc(start);
    description.resultsTime = 'Result retrieval time';

    % Verify that the job ran without any errors.
    if ~isempty([job.Tasks.Error])
        taskErrorMsgs = pctdemo_helper_getUniqueErrors(job);
        delete(job);
        error('pctexample:distribjobbench:JobErrored',  ...
          ['The following error(s) occurred during task ' ...
          'execution:\n\n%s'], taskErrorMsgs);
    end

    % Get the execution time of the tasks.  Our task function returns this
    % as its second output argument.
    times.exeTime = max([results{:,2}]);
    description.exeTime = 'Task execution time';

    % Measure how long it takes to delete the job and all its tasks.
    start = tic;
    delete(job);
    times.deleteTime = toc(start);
    description.deleteTime = 'Job deletion time';

    % Measure the total time elapsed from creating the job up to this
    % point.
    times.totalTime = toc(timingStart);
    description.totalTime = 'Total time';

    times.numTasks = numTasks;
    description.numTasks = 'Number of tasks';
end
```

We look at some of the details of what we are measuring:

- **Job creation time**: The time it takes to create a job. For a MATLAB Job Scheduler cluster, this involves a remote call, and the MATLAB Job Scheduler allocates space in its data base. For other cluster types, job creation involves writing a few files to disk.

- **Task creation time**: The time it takes to create and save the task information. The MATLAB Job Scheduler saves this in its data base, whereas other cluster types save it in files on the file system.

- **Job submission time**: The time it takes to submit the job. For a MATLAB Job Scheduler cluster, we tell it to start executing the job it has in its data base. We ask other cluster types to execute all the tasks we have created.

- **Job wait time**: The time we wait after the job submission until job completion. This includes all the activities that take place between job submission and when the job has completed, such as: cluster may need to start all the workers and to send the workers the task information; the workers read the task information, and execute the task function. In the case of a MATLAB Job Scheduler cluster, the workers then send the task results to the MATLAB Job Scheduler, which writes them to its data base, whereas for the other cluster types, the workers write the task results to disk.

- **Task execution time**: The time spent simulating blackjack. We instrument the task function to accurately measure this time. This time is also included in the job wait time.

- **Results retrieval time**: The time it takes to bring the job results into the MATLAB client. For the MATLAB Job Scheduler, we obtain them from its data base. For other cluster types, we read them from the file system.

- **Job deletion time**: The time it takes to delete all the job and task information. The MATLAB Job Scheduler deletes it from its data base. For the other cluster types, we delete the files from the file system.

- **Total time**: The time it takes to perform all of the above.

Choosing Problem Size

We know that most clusters are designed for batch execution of medium or long running jobs, so we deliberately try to have our benchmark calculations fall within that range. Yet, we do not want this example to take hours to run, so we choose the problem size so that each task takes approximately 1 minute on our hardware, and we then repeat the timing measurements a few times for increased accuracy. As a rule of thumb, if your calculations in a task take much less than a minute, you should consider whether `parfor` meets your low-latency needs better than jobs and tasks.

```
numHands = 1.2e6;
numReps = 5;
```

We explore speedup by running on a different number of workers, starting with 1, 2, 4, 8, 16, etc., and ending with as many workers as we can possibly use. In this example, we assume that we have dedicated access to the cluster for the benchmarking, and that the cluster's `NumWorkers` property has been set correctly. Assuming that to be the case, each task will execute right away on a dedicated worker, so we can equate the number of tasks we submit with the number of workers that execute them.

```
numWorkers = myCluster.NumWorkers ;
if isinf(numWorkers) || (numWorkers == 0)
    error('pctexample:distribjobbench:InvalidNumWorkers', ...
          ['Cannot deduce the number of workers from the cluster.  ' ...
           'Set the NumWorkers on your default profile to be ' ...
           'a value other than 0 or inf.']);
end
```

```
numTasks = [pow2(0:ceil(log2(numWorkers) - 1)), numWorkers];
```

Weak Scaling Measurements

We vary the number of tasks in a job, and have each task perform a fixed amount of work. This is called **weak scaling**, and is what we really care the most about, because we usually scale up to the cluster to solve larger problems. It should be compared with the strong scaling benchmarks shown later in this example. Speedup based on weak scaling is also known as **scaled speedup**.

```
fprintf(['Starting weak scaling timing.  ' ...
         'Submitting a total of %d jobs.\n'], numReps*length(numTasks));
for j = 1:length(numTasks)
    n = numTasks(j);
    for itr = 1:numReps
        [rep(itr), description] = timeJob(myCluster, n, numHands);  %#ok<AGROW>
    end
    % Retain the iteration with the lowest total time.
    totalTime = [rep.totalTime];
    fastest = find(totalTime == min(totalTime), 1);
    weak(j) = rep(fastest);  %#ok<AGROW>
    fprintf('Job wait time with %d task(s): %f seconds\n', ...
            n, weak(j).jobWaitTime);
end
```

```
Starting weak scaling timing.  Submitting a total of 45 jobs.
Job wait time with 1 task(s): 59.631733 seconds
Job wait time with 2 task(s): 60.717059 seconds
Job wait time with 4 task(s): 61.343568 seconds
Job wait time with 8 task(s): 60.759119 seconds
Job wait time with 16 task(s): 63.016560 seconds
Job wait time with 32 task(s): 64.615484 seconds
Job wait time with 64 task(s): 66.581806 seconds
Job wait time with 128 task(s): 91.043285 seconds
Job wait time with 256 task(s): 150.411704 seconds
```

Sequential Execution

We measure the sequential execution time of the computations. Note that this time should be compared to the execution time on the cluster only if they have the same hardware and software configuration.

```
seqTime = inf;
for itr = 1:numReps
    start = tic;
    pctdemo_task_blackjack(numHands, 1);
    seqTime = min(seqTime, toc(start));
end
fprintf('Sequential execution time: %f seconds\n', seqTime);
```

```
Sequential execution time: 84.771630 seconds
```

Speedup Based on Weak Scaling and Total Execution Time

We first look at the overall speedup achieved by running on different numbers of workers. The speedup is based on the total time used for the computations, so it includes both the sequential and the parallel portions of our code.

This speedup curve represents the capabilities of multiple items with unknown weights associated with each of them: The cluster hardware, the cluster software, the client hardware, the client software, and the connection between the client and the cluster. Therefore, the speedup curve does not represent any one of these, but all taken together.

If the speedup curve meets your desired performance targets, you know that all the aforementioned factors work well together in this particular benchmark. However, if the speedup curve fails to meet your targets, you do not know which of the many factors listed above is the most to blame. It could even be that the approach taken in the parallelization of the application is to blame rather than either the other software or hardware.

All too often, novices believe that this single graph gives the complete picture of the performance of their cluster hardware or software. This is indeed not the case, and one always needs to be aware that this graph does not allow us to draw any conclusions about potential performance bottlenecks.

```
titleStr = sprintf(['Speedup based on total execution time\n' ...
                    'Note: This graph does not identify performance ' ...
                    'bottlenecks']);
pctdemo_plot_distribjob('speedup', [weak.numTasks], [weak.totalTime], ...
    weak(1).totalTime, titleStr);
```
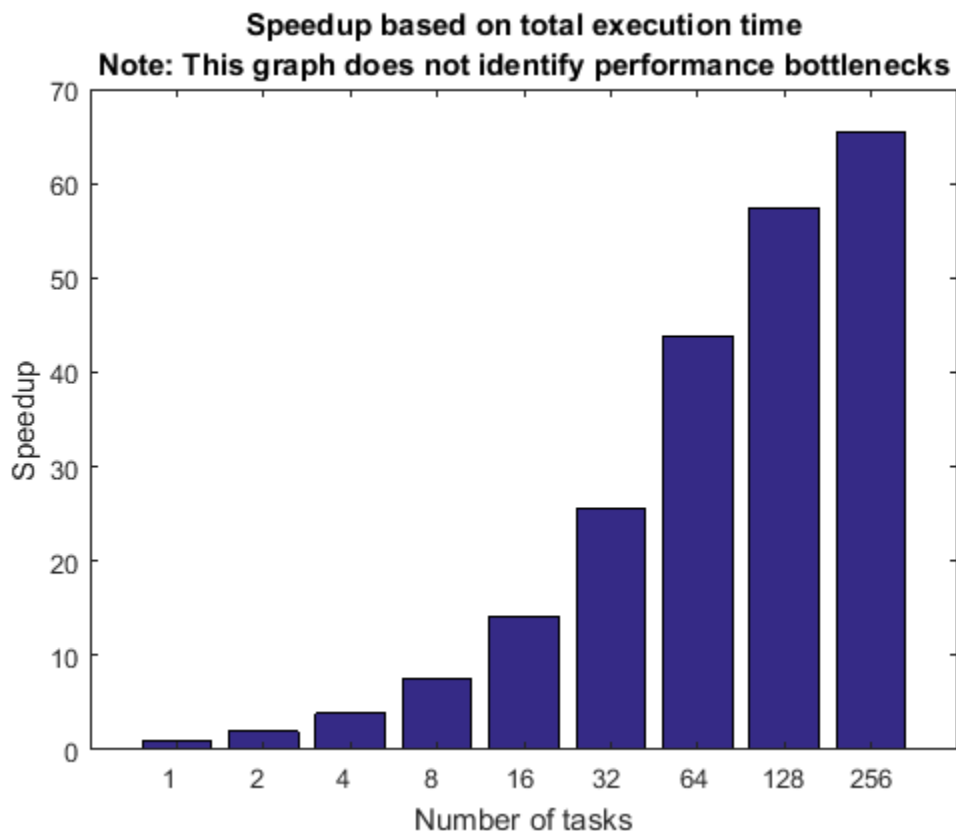

Detailed Graphs, Part 1

We dig a little bit deeper and look at the times spent in the various steps of our code. We benchmarked weak scaling, that is, the more tasks we create, the more work we perform. Therefore, the size of the task output data increases as we increase the number of tasks. With that in mind, we expect the following to take longer the more tasks we create:

- Task creation
- Retrieval of job output arguments
- Job destruction time

We have no reason to believe that the following increases with the number of tasks:

- Job creation time

After all, the job is created before we define any of its tasks, so there is no reason why it should vary with the number of tasks. We might expect to see only some random fluctuations in the job creation time.

```
pctdemo_plot_distribjob('fields', weak, description, ...
    {'jobCreateTime', 'taskCreateTime',  'resultsTime', 'deleteTime'}, ...
    'Time in seconds');
```
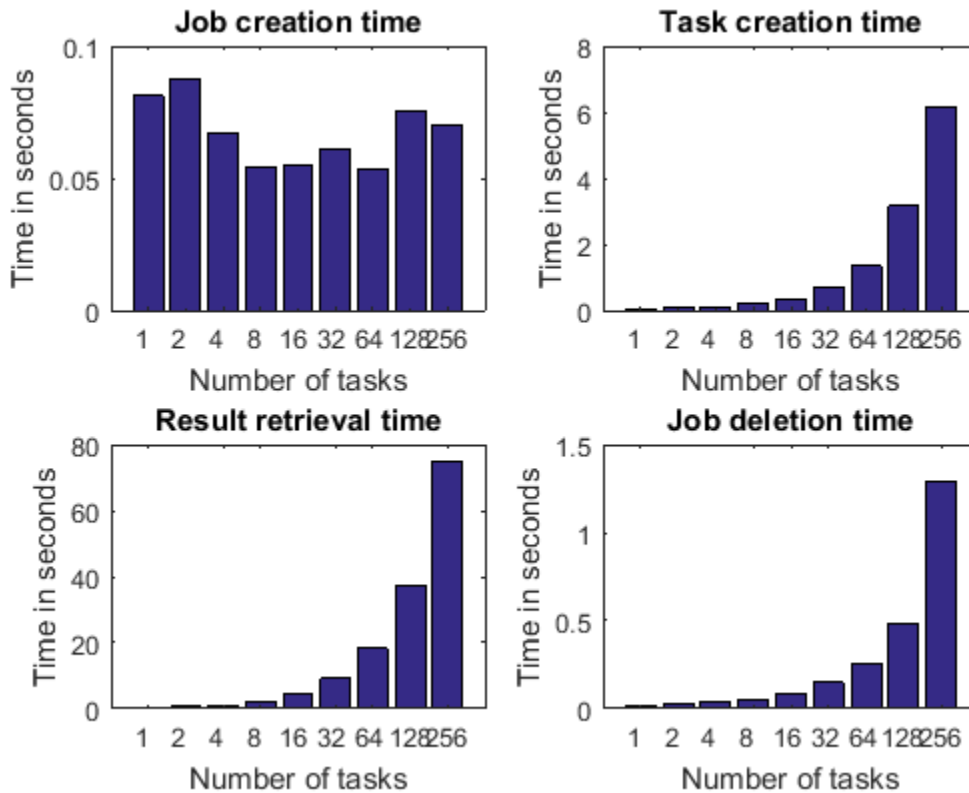

Normalized Times

We already concluded that task creation time is expected to increase as we increase the number of tasks, as does the time to retrieve job output arguments and to delete the job. However, this increase is due to the fact that we are performing more work as we increase the number of workers/tasks. It is therefore meaningful to measure the efficiency of these three activities by looking at the time it takes to perform these operations, and normalize it by the number of tasks. This way, we can look to see if any of the following times stay constant, increase, or decrease as we vary the number of tasks:

- The time it takes to create a single task
- The time it takes to retrieve output arguments from a single task
- The time it takes to delete a task in a job

The normalized times in this graph represent the capabilities of the MATLAB client and the portion of the cluster hardware or software that it might interact with. It is generally considered good if these curves stay flat, and excellent if they are decreasing.

```
pctdemo_plot_distribjob('normalizedFields', weak, description, ...
    {'taskCreateTime', 'resultsTime', 'deleteTime'});
```
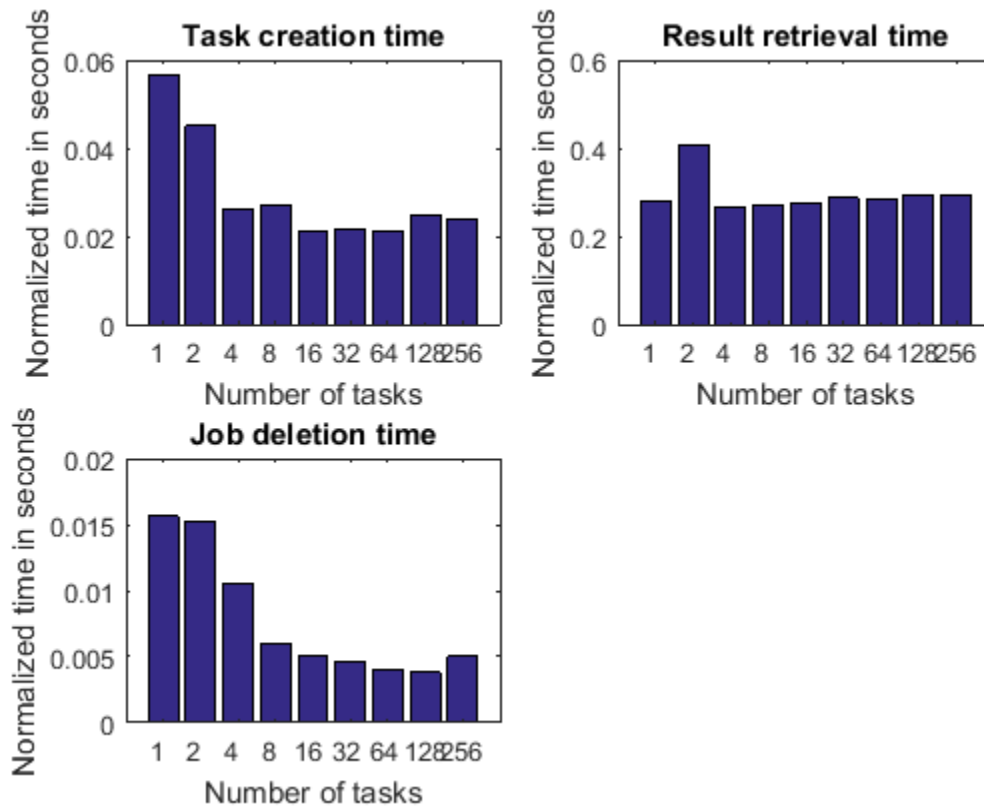

These graphs sometimes show that the time spent retrieving the results per task goes down as the number of tasks increases. That is undeniably good: We become more efficient the more work we perform. This might happen if there is a fixed amount of overhead for the operation and if it takes a fixed amount of time per task in the job.

We cannot expect a speedup curve based on total execution time to look particularly good if it includes a significant amount of time spent on sequential activities such as the above, where the time spent increases with the number of tasks. In that case, the sequential activities will dominate once there are sufficiently many tasks.

Detailed Graphs, Part 2

It is possible that the time spent in each of following steps varies with the number of tasks, but we hope it does not:

- Job submission time.
- Task execution time. This captures the time spent simulating blackjack. Nothing more, nothing less.

In both cases, we look at the elapsed time, also referred to as wall clock time. We look at neither the total CPU time on the cluster nor the normalized time.

```
pctdemo_plot_distribjob('fields', weak, description, ...
    {'submitTime', 'exeTime'});
```
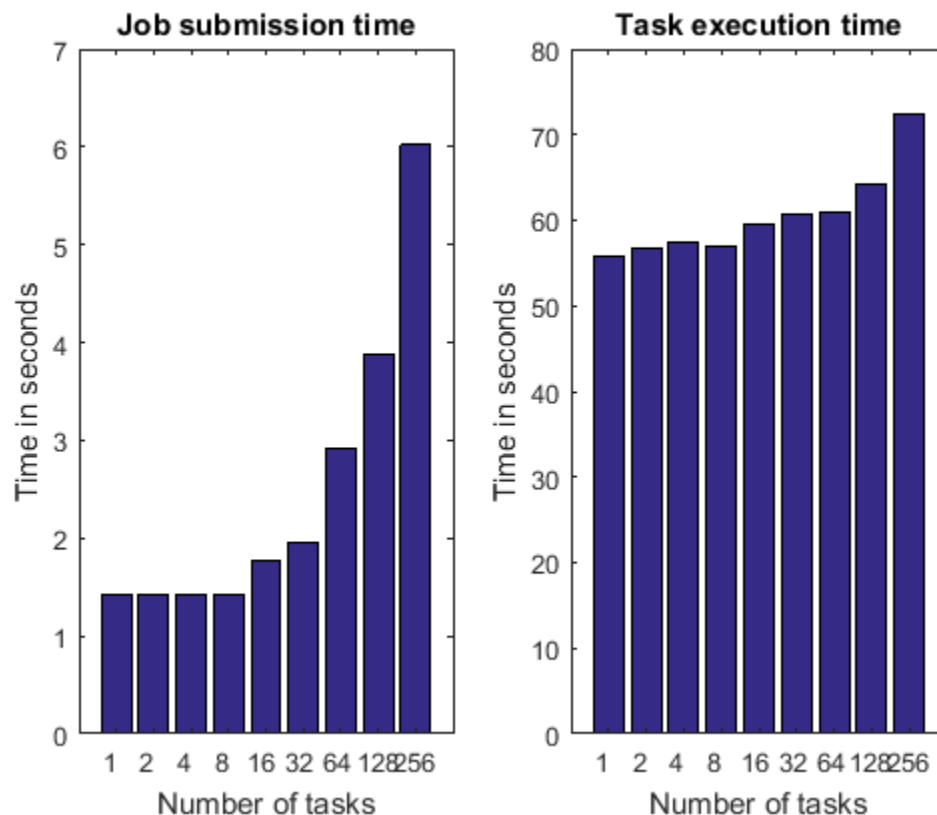

There are situations where each of the times shown above could increase with the number of tasks. For example:

- With some third-party cluster types, the job submission involves one system call for each task in the job, or the job submission involves copying files across the network. In those cases, the job submission time may increase linearly with the number of tasks.
- The graph of the task execution time is the most likely to expose hardware limitations and resource contention. For example, the task execution time could increase if we are executing multiple workers on the same computer, due to contention for limited memory bandwidth. Another example of resource contention is if the task function were to read or write large data files using a single, shared file system. The task function in this example, however, does not access the file system at all. These types of hardware limitations are covered in great detail in the example "Resource Contention in Task Parallel Problems" on page 9-129.

Speedup Based on Weak Scaling and Job Wait Time

Now that we have dissected the times spent in the various stages of our code, we want to create a speedup curve that more accurately reflects the capabilities of our cluster hardware and software. We do this by calculating a speedup curve based on the job wait time.

When calculating this speedup curve based on the job wait time, we first compare it to the time it takes to execute a job with a single task on the cluster.

```
titleStr = 'Speedup based on job wait time compared to one task';
pctdemo_plot_distribjob('speedup', [weak.numTasks], [weak.jobWaitTime], ...
    weak(1).jobWaitTime, titleStr);
```
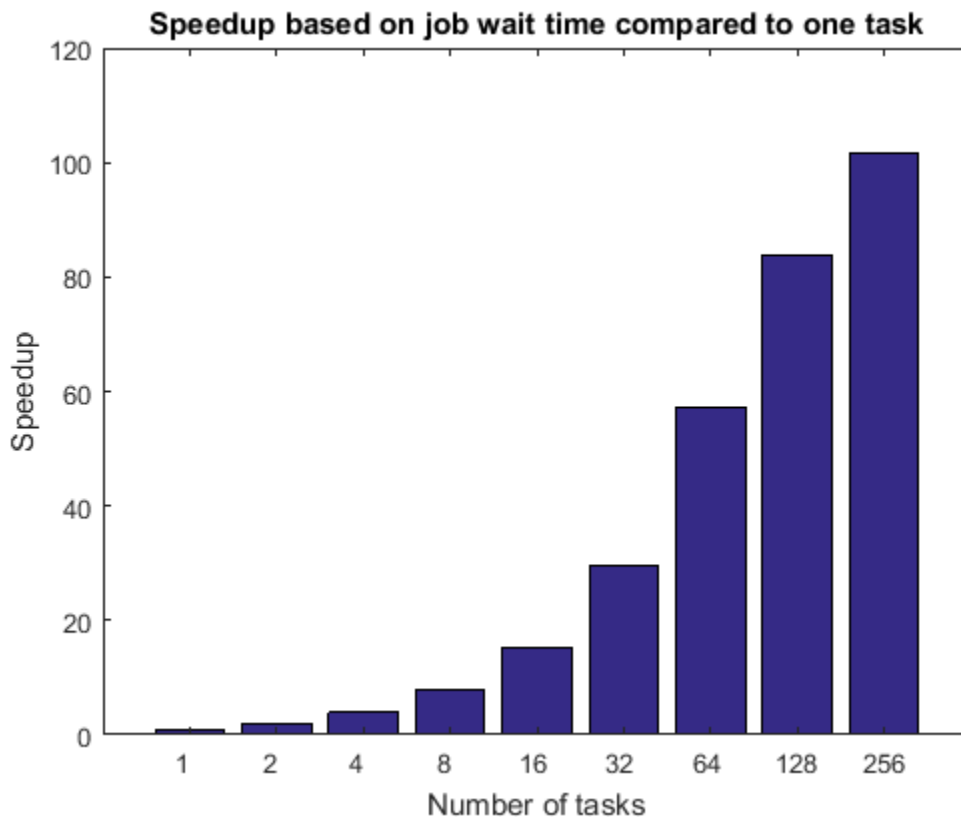

Job wait time might include the time to start all the MATLAB workers. It is therefore possible that this time is bounded by the IO capabilities of a shared file system. The job wait time also includes the average task execution time, so any deficiencies seen there also apply here. If we do not have dedicated access to the cluster, we could expect the speedup curve based on job wait time to suffer significantly.

Next, we compare the job wait time to the sequential execution time, assuming that the hardware of the client computer is comparable to the compute nodes. If the client is not comparable to the cluster nodes, this comparison is absolutely meaningless. If your cluster has a substantial time lag when assigning tasks to workers, e.g., by assigning tasks to workers only once per minute, this graph will be heavily affected because the sequential execution time does not suffer this lag. Note that this graph will have the same shape as the previous graph, they will only differ by a constant, multiplicative factor.

```
titleStr = 'Speedup based on job wait time compared to sequential time';
pctdemo_plot_distribjob('speedup', [weak.numTasks], [weak.jobWaitTime], ...
    seqTime, titleStr);
```
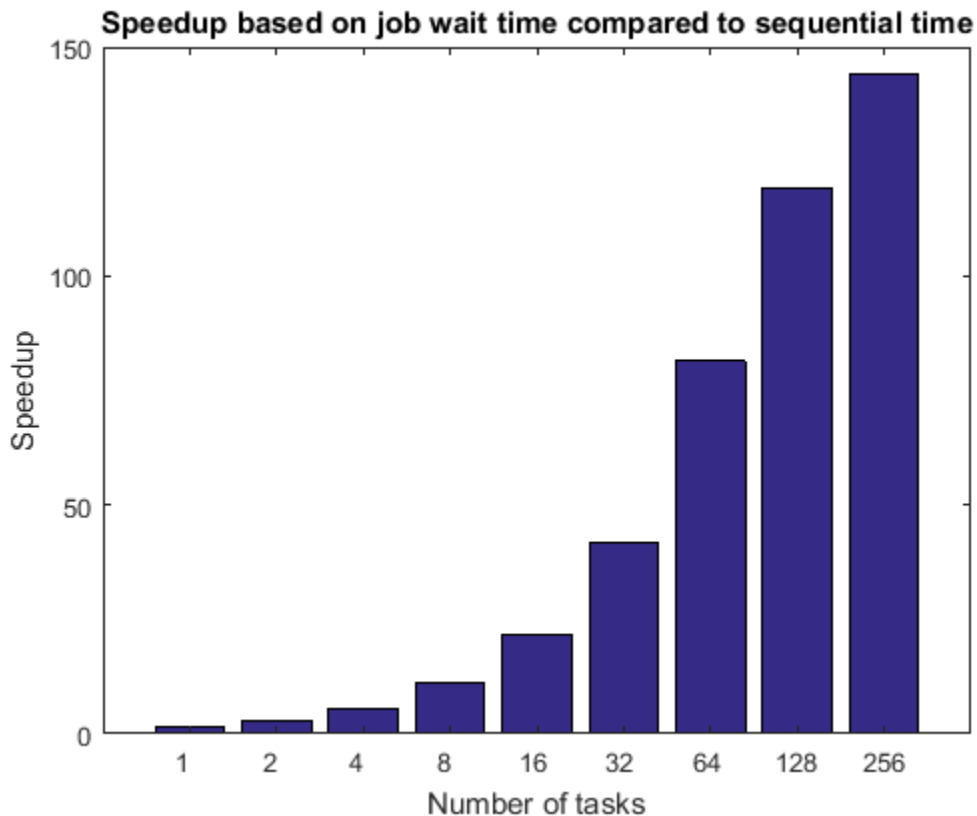

Comparing Job Wait Time with Task Execution Time

As we have mentioned before, the job wait time consists of the task execution time plus scheduling, wait time in the cluster's queue, MATLAB startup time, etc. On an idle cluster, the difference between the job wait time and task execution time should remain constant, at least for small number of tasks. As the number of tasks grows into the tens, hundreds, or thousands, we are bound to eventually run into some limitations. For example, once we have sufficiently many tasks/workers, the cluster cannot tell all the workers simultaneously to start executing their task, or if the MATLAB workers all use the same file system, they might end up saturating the file server.

```
titleStr = 'Difference between job wait time and task execution time';
pctdemo_plot_distribjob('barTime', [weak.numTasks], ...
    [weak.jobWaitTime] - [weak.exeTime], titleStr);
```
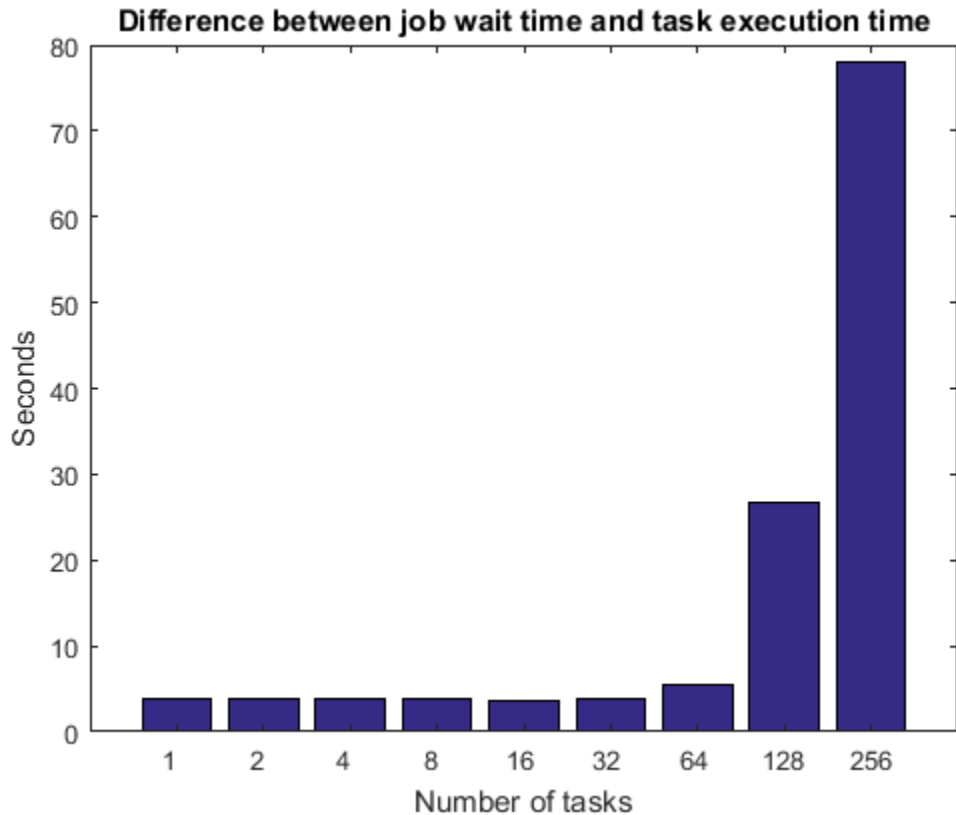
Strong Scaling Measurements

We now measure the execution time of a fixed-size problem, while varying the number of workers we use to solve the problem. This is called **strong scaling**, and it is well known that if an application has any sequential parts, there is an upper limit to the speedup that can be achieved with strong scaling. This is formalized in **Amdahl's law**, which has been widely discussed and debated over the years.

You can easily run into the limits of speedup with strong scaling when submitting jobs to the cluster. If the task execution has a fixed overhead (which it ordinarily does), even if it is as little as one second, the execution time of our application will never go below one second. In our case, we start with an application that executes in approximately 60 seconds on one MATLAB worker. If we divide the computations among 60 workers, it might take as little as one second for each worker to compute its portion of the overall problem. However, the hypothetical task execution overhead of one second has become a major contributor to the overall execution time.

Unless your application runs for a long time, jobs and tasks are usually not the way to achieve good results with strong scaling. If the overhead of task execution is close to the execution time of your application, you should investigate whether `parfor` meets your requirements. Even in the case of `parfor`, there is a fixed amount of overhead, albeit much smaller than with regular jobs and tasks, and that overhead limits to the speedup that can be achieved with strong scaling. Your problem size relative to your cluster size may or may not be so large that you experience those limitations.

As a general rule of thumb, it is only possible to achieve strong scaling of small problems on large numbers of processors with specialized hardware and a great deal of programming effort.

```
fprintf(['Starting strong scaling timing.  ' ...
          'Submitting a total of %d jobs.\n'], numReps*length(numTasks))
for j = 1:length(numTasks)
    n = numTasks(j);
    strongNumHands = ceil(numHands/n);
    for itr = 1:numReps
        rep(itr) = timeJob(myCluster, n, strongNumHands);
    end
    ind = find([rep.totalTime] == min([rep.totalTime]), 1);
    strong(n) = rep(ind); %#ok<AGROW>
    fprintf('Job wait time with %d task(s): %f seconds\n', ...
            n, strong(n).jobWaitTime);
end
```

```
Starting strong scaling timing.  Submitting a total of 45 jobs.
Job wait time with 1 task(s): 60.531446 seconds
Job wait time with 2 task(s): 31.745135 seconds
Job wait time with 4 task(s): 18.367432 seconds
Job wait time with 8 task(s): 11.172390 seconds
Job wait time with 16 task(s): 8.155608 seconds
Job wait time with 32 task(s): 6.298422 seconds
Job wait time with 64 task(s): 5.253394 seconds
Job wait time with 128 task(s): 5.302715 seconds
Job wait time with 256 task(s): 49.428909 seconds
```

Speedup Based on Strong Scaling and Total Execution Time

As we have already discussed, speedup curves that depict the sum of the time spent executing sequential code in the MATLAB client and time executing parallel code on the cluster can be very misleading. The following graph shows this information in the worst-case scenario of strong scaling. We deliberately chose the original problem to be so small relative to our cluster size that the speedup curve would look bad. Neither the cluster hardware nor software was designed with this kind of a use in mind.

```
titleStr = sprintf(['Speedup based on total execution time\n' ...
                    'Note: This graph does not identify performance ' ...
                    'bottlenecks']);
pctdemo_plot_distribjob('speedup', [strong.numTasks], ...
    [strong.totalTime].*[strong.numTasks], strong(1).totalTime, titleStr);
```
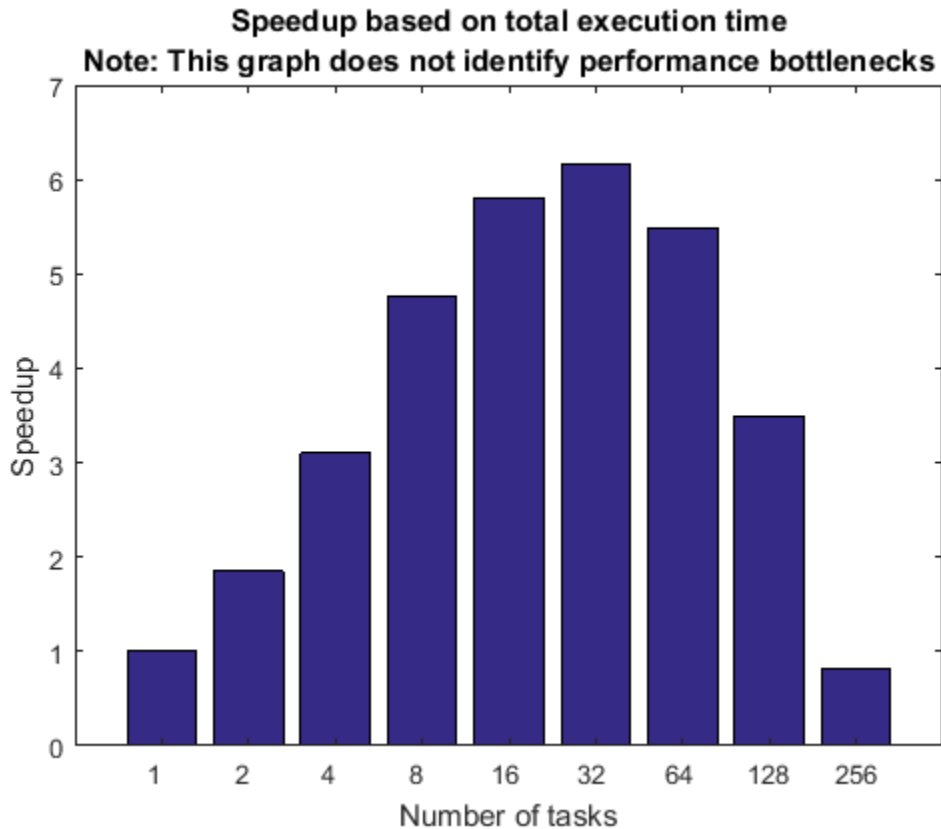
Alternative for Short Tasks: PARFOR

The strong scaling results did not look good because we deliberately used jobs and tasks to execute calculations of short duration. We now look at how `parfor` applies to that same problem. Note that we do not include the time it takes to open the pool in our time measurements.

```
pool = parpool(numWorkers);
parforTime = inf;
strongNumHands = ceil(numHands/numWorkers);
for itr = 1:numReps
    start = tic;
    r = cell(1, numWorkers);
    parfor i = 1:numWorkers
        r{i} = pctdemo_task_blackjack(strongNumHands, 1); %#ok<PFOUS>
    end
    parforTime = min(parforTime, toc(start));
end
delete(pool);
```

```
Starting parallel pool (parpool) using the 'bigMJS' profile ... connected to 256 workers.
Analyzing and transferring files to the workers ...done.
```

Speedup Based on Strong Scaling with PARFOR

The original, sequential calculations took approximately one minute, so each worker needs to perform only a few seconds of computations on a large cluster. We therefore expect strong scaling performance to be much better with `parfor` than with jobs and tasks.

```
fprintf('Execution time with parfor using %d workers: %f seconds\n', ...
    numWorkers, parforTime);
fprintf(['Speedup based on strong scaling with parfor using ', ...
    '%d workers: %f\n'], numWorkers, seqTime/parforTime);
```

```
Execution time with parfor using 256 workers: 1.126914 seconds
Speedup based on strong scaling with parfor using 256 workers: 75.224557
```

Summary

We have seen the difference between weak and strong scaling, and discussed why we prefer to look at weak scaling: It measures our ability to solve larger problems on the cluster (more simulations, more iterations, more data, etc.). The large number of graphs and the amount of detail in this example should also be a testament to the fact that benchmarks cannot be boiled down to a single number or a single graph. We need to look at the whole picture to understand whether the application performance can be attributed to the application, the cluster hardware or software, or a combination of both.

We have also seen that for short calculations, `parfor` can be a great alternative to jobs and tasks. For more benchmarking results using `parfor`, see the example "Simple Benchmarking of PARFOR Using Blackjack" on page 9-102.

```
end
```

Benchmarking A\b

This example shows how to benchmark solving a linear system on a cluster. The MATLAB® code to solve for x in A*x = b is very simple. Most frequently, one uses matrix left division, also known as mldivide or the backslash operator (\), to calculate x (that is, x = A\b). Benchmarking the performance of matrix left division on a cluster, however, is not as straightforward.

One of the most challenging aspects of benchmarking is to avoid falling into the trap of looking for a single number that represents the overall performance of the system. We will look at the performance curves that might help you identify the performance bottlenecks on your cluster, and maybe even help you see how to benchmark your code and be able to draw meaningful conclusions from the results.

Related examples:

- "Simple Benchmarking of PARFOR Using Blackjack" on page 9-102
- "Benchmarking Independent Jobs on the Cluster" on page 9-137
- "Resource Contention in Task Parallel Problems" on page 9-129

The code shown in this example can be found in this function:

```
function results = paralleldemo_backslash_bench(memoryPerWorker)
```

It is very important to choose the appropriate matrix size for the cluster. We can do this by specifying the amount of system memory in GB available to each worker as an input to this example function. The default value is very conservative; you should specify a value that is appropriate for your system.

```
if nargin == 0
    memoryPerWorker = 8.00; % In GB
%    warning('pctexample:backslashbench:BackslashBenchUsingDefaultMemory', ...
%            ['Amount of system memory available to each worker is ', ...
%             'not specified.  Using the conservative default value ', ...
%             'of %.2f gigabytes per worker.'], memoryPerWorker);
end
```

Avoiding Overhead

To get an accurate measure of our capability to solve linear systems, we need to remove any possible source of overhead. This includes getting the current parallel pool and temporarily disabling the deadlock detection capabilities.

```
p = gcp;
if isempty(p)
    error('pctexample:backslashbench:poolClosed', ...
        ['This example requires a parallel pool. ' ...
         'Manually start a pool using the parpool command or set ' ...
         'your parallel preferences to automatically start a pool.']);
end
poolSize = p.NumWorkers;
pctRunOnAll 'mpiSettings(''DeadlockDetection'', ''off'');'
```

```
Starting parallel pool (parpool) using the 'bigMJS' profile ... connected to 12 workers.
```

The Benchmarking Function

We want to benchmark matrix left division (\), and not the cost of entering an spmd block, the time it takes to create a matrix, or other parameters. We therefore separate the data generation from the

solving of the linear system, and measure only the time it takes to do the latter. We generate the input data using the 2-D block-cyclic codistributor, as that is the most effective distribution scheme for solving a linear system. Our benchmarking then consists of measuring the time it takes all the workers to complete solving the linear system A*x = b. Again, we try to remove any possible source of overhead.

```matlab
function [A, b] = getData(n)
    fprintf('Creating a matrix of size %d-by-%d.\n', n, n);
    spmd
        % Use the codistributor that usually gives the best performance
        % for solving linear systems.
        codistr = codistributor2dbc(codistributor2dbc.defaultLabGrid, ...
                                    codistributor2dbc.defaultBlockSize, ...
                                    'col');
        A = codistributed.rand(n, n, codistr);
        b = codistributed.rand(n, 1, codistr);
    end
end

function time = timeSolve(A, b)
    spmd
        tic;
        x = A\b; %#ok<NASGU> We don't need the value of x.
        time = gop(@max, toc); % Time for all to complete.
    end
    time = time{1};
end
```

Choosing Problem Size

Just like with a great number of other parallel algorithms, the performance of solving a linear system in parallel depends greatly on the matrix size. Our *a priori* expectations are therefore that the computations be:

- Somewhat inefficient for small matrices
- Quite efficient for large matrices
- Inefficient if the matrices are too large to fit into system memory and the operating systems start swapping memory to disk

It is therefore important to time the computations for a number of different matrix sizes to gain an understanding of what "small," "large," and "too large" mean in this context. Based on previous experiments, we expect:

- "Too small" matrices to be of size 1000-by-1000
- "Large" matrices to occupy slightly less than 45% of the memory available to each worker
- "Too large" matrices occupy 50% or more of system memory available to each worker

These are heuristics, and the precise values may change between releases. It is therefore important that we use matrix sizes that span this entire range and verify the expected performance.

Notice that by changing the problem size according to the number of workers, we employ weak scaling. Other benchmarking examples, such as "Simple Benchmarking of PARFOR Using Blackjack" on page 9-102 and "Benchmarking Independent Jobs on the Cluster" on page 9-137, also employ weak scaling. As those examples benchmark task parallel computations, their weak scaling consists of making the number of iterations proportional to the number of workers. This example, however, is

benchmarking data parallel computations, so we relate the upper size limit of the matrices to the number of workers.

```
% Declare the matrix sizes ranging from 1000-by-1000 up to 45% of system
% memory available to each worker.
maxMemUsagePerWorker = 0.45*memoryPerWorker*1024^3; % In bytes.
maxMatSize = round(sqrt(maxMemUsagePerWorker*poolSize/8));
matSize = round(linspace(1000, maxMatSize, 5));
```

Comparing Performance: Gigaflops

We use the number of floating point operations per second as our measure of performance because that allows us to compare the performance of the algorithm for different matrix sizes and different number of workers. If we are successful in testing the performance of matrix left division for a sufficiently wide range of matrix sizes, we expect the performance graph to look similar to the following:
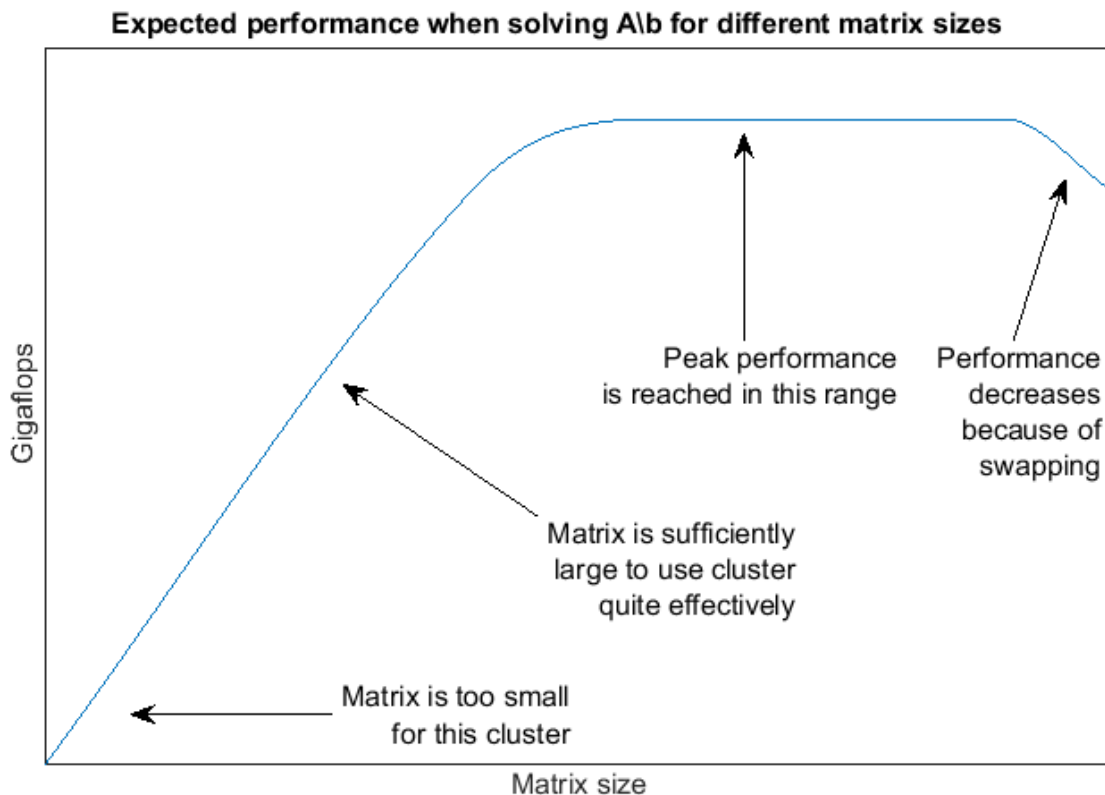
By generating graphs such as these, we can answer questions such as:

*   Are the smallest matrices so small that we get poor performance?
*   Do we see a performance decrease when the matrix is so large that it occupies 45% of total system memory?
*   What is the best performance we can possibly achieve for a given number of workers?
*   For which matrix sizes do 16 workers perform better than 8 workers?

- Is the system memory limiting the peak performance?

Given a matrix size, the benchmarking function creates the matrix A and the right-hand side b once, and then solves A\b multiple times to get an accurate measure of the time it takes. We use the floating operations count of the HPC Challenge, so that for an n-by-n matrix, we count the floating point operations as 2/3*n^3 + 3/2*n^2.

```matlab
function gflops = benchFcn(n)
    numReps = 3;
    [A, b] = getData(n);
    time = inf;
    % We solve the linear system a few times and calculate the Gigaflops
    % based on the best time.
    for itr = 1:numReps
        tcurr = timeSolve(A, b);
        if itr == 1
            fprintf('Execution times: %f', tcurr);
        else
            fprintf(', %f', tcurr);
        end
        time = min(tcurr, time);
    end
    fprintf('\n');
    flop = 2/3*n^3 + 3/2*n^2;
    gflops = flop/time/1e9;
end
```

Executing the Benchmarks

Having done all the setup, it is straightforward to execute the benchmarks. However, the computations may take a long time to complete, so we print some intermediate status information as we complete the benchmarking for each matrix size.

```matlab
fprintf(['Starting benchmarks with %d different matrix sizes ranging\n' ...
        'from %d-by-%d to %d-by-%d.\n'], ...
        length(matSize), matSize(1), matSize(1), matSize(end), ...
        matSize(end));
gflops = zeros(size(matSize));
for i = 1:length(matSize)
    gflops(i) = benchFcn(matSize(i));
    fprintf('Gigaflops: %f\n\n', gflops(i));
end
results.matSize = matSize;
results.gflops = gflops;

Starting benchmarks with 5 different matrix sizes ranging
from 1000-by-1000 to 76146-by-76146.
Creating a matrix of size 1000-by-1000.
Analyzing and transferring files to the workers ...done.
Execution times: 1.038931, 0.592114, 0.575135
Gigaflops: 1.161756

Creating a matrix of size 19787-by-19787.
Execution times: 119.402579, 118.087116, 119.323904
Gigaflops: 43.741681

Creating a matrix of size 38573-by-38573.
Execution times: 552.256063, 549.088060, 555.753578
```

```
Gigaflops: 69.685485

Creating a matrix of size 57360-by-57360.
Execution times: 3580.232186, 3726.588242, 3113.261810
Gigaflops: 40.414533

Creating a matrix of size 76146-by-76146.
Execution times: 9261.720799, 9099.777287, 7968.750495
Gigaflops: 36.937936
```

Plotting the Performance

We can now plot the results, and compare to the expected graph shown above.

```
fig = figure;
ax = axes('parent', fig);
plot(ax, matSize/1000, gflops);
lines = ax.Children;
lines.Marker = '+';
ylabel(ax, 'Gigaflops')
xlabel(ax, 'Matrix size in thousands')
titleStr = sprintf(['Solving A\\b for different matrix sizes on ' ...
                    '%d workers'], poolSize);
title(ax, titleStr, 'Interpreter', 'none');
```
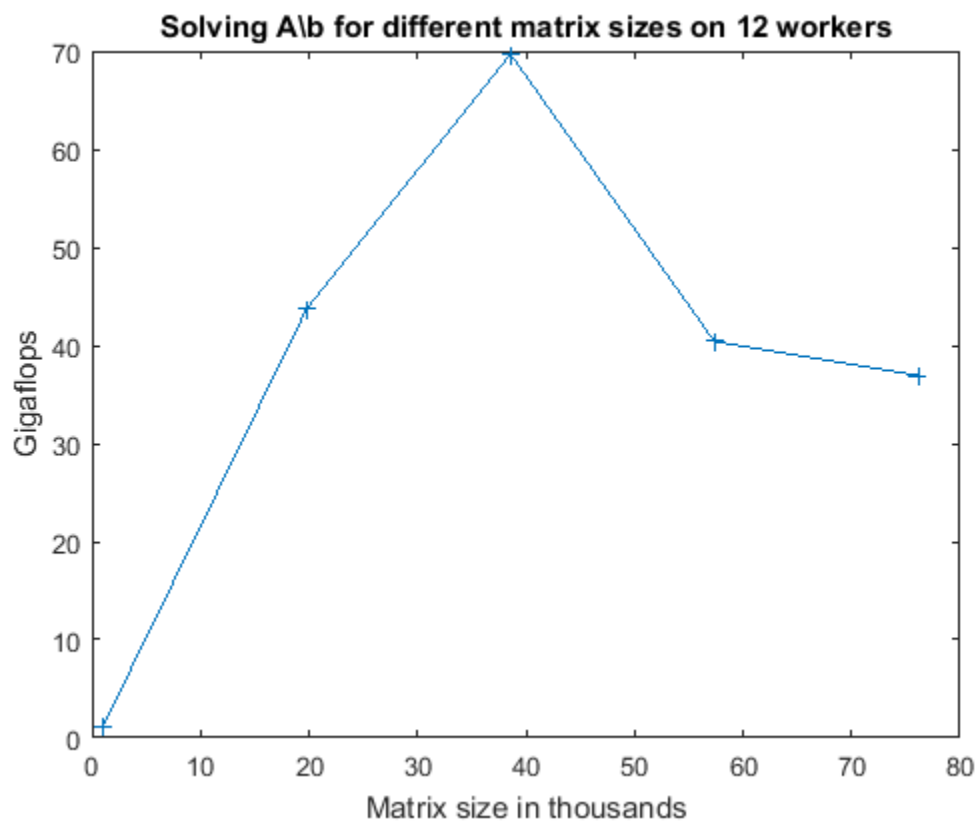

If the benchmark results are not as good as you might expect, here are some things to consider:

- The underlying implementation is using ScaLAPACK, which has a proven reputation of high performance. It is therefore very unlikely that the algorithm or the library is causing inefficiencies, but rather the way in which it is used, as described in the items below.
- If the matrices are too small or too large for your cluster, the resulting performance will be poor.
- If the network communications are slow, performance will be severely impacted.
- If the CPUs and the network communications are both very fast, but the amount of memory is limited, it is possible you are not able to benchmark with sufficiently large matrices to fully utilize the available CPUs and network bandwidth.
- For ultimate performance, it is important to use a version of MPI that is tailored for your networking setup, and have the workers running in such a manner that as much of the communication happens through shared memory as possible. It is, however, beyond the scope of this example to explain how to identify and solve those types of problems.

Comparing Different Numbers of Workers

We now look at how to compare different numbers of workers by viewing data obtained by running this example using different numbers of workers. This data is obtained on a different cluster from the one above.

Other examples such as "Benchmarking Independent Jobs on the Cluster" on page 9-137 have explained that when benchmarking parallel algorithms for different numbers of workers, one usually employs weak scaling. That is, as we increase the number of workers, we increase the problem size proportionally. In the case of matrix left division, we have to show additional care because the performance of the division depends greatly on the size of the matrix. The following code creates a graph of the performance in Gigaflops for all of the matrix sizes that we tested with and all the different numbers of workers, as that gives us the most detailed picture of the performance characteristics of matrix left division *on this particular cluster*.

```
s = load('pctdemo_data_backslash.mat', 'workers4', 'workers8', ...
        'workers16', 'workers32', 'workers64');
fig = figure;
ax = axes('parent', fig);
plot(ax, s.workers4.matSize./1000, s.workers4.gflops, ...
    s.workers8.matSize./1000, s.workers8.gflops, ...
    s.workers16.matSize./1000, s.workers16.gflops, ...
    s.workers32.matSize./1000, s.workers32.gflops, ...
    s.workers64.matSize./1000, s.workers64.gflops);
lines = ax.Children;
set(lines, {'Marker'}, {'+'; 'o'; 'v'; '.'; '*'});
ylabel(ax, 'Gigaflops')
xlabel(ax, 'Matrix size in thousands')
title(ax, ...
    'Comparison data for solving A\\b on different numbers of workers');
legend('4 workers', '8 workers', '16 workers', '32 workers',  ...
    '64 workers', 'location', 'NorthWest');
```
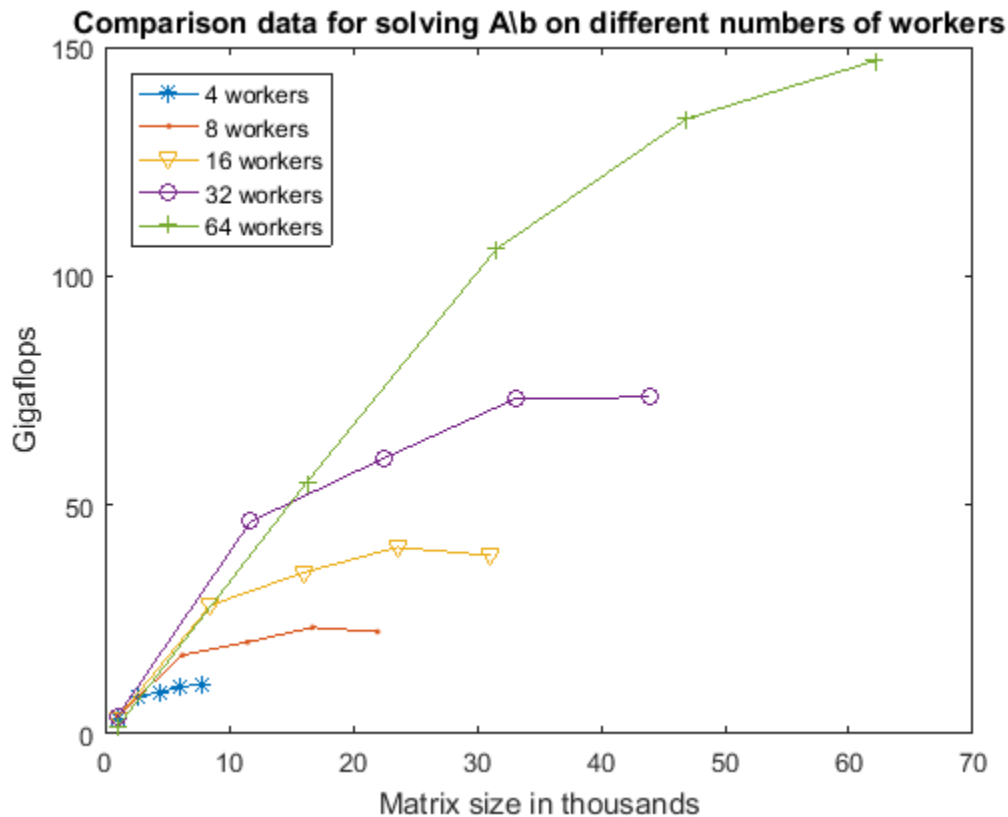
The first thing we notice when looking at the graph above is that 64 workers allow us to solve much larger linear systems of equations than is possible with only 4 workers. Additionally, we can see that even if one could work with a matrix of size 60,000-by-60,000 on 4 workers, we would get a performance of approximately only 10 Gigaflops. Thus, even if the 4 workers had sufficient memory to solve such a large problem, 64 workers would nevertheless greatly outperform them.

Looking at the slope of the curve for 4 workers, we can see that there is only a modest performance increase between the three largest matrix sizes. Comparing this with the earlier graph of the expected performance of A\b for different matrix sizes, we conclude that we are quite close to achieving peak performance for 4 workers with matrix size of 7772-by-7772.

Looking at the curve for 8 and 16 workers, we can see that the performance drops for the largest matrix size, indicating that we are near or already have exhausted available system memory. However, we see that the performance increase between the second and third largest matrix sizes is very modest, indicating stability of some sort. We therefore conjecture that when working with 8 or 16 workers, we would most likely not see a significant increase in the Gigaflops if we increased the system memory and tested with larger matrix sizes.

Looking at the curves for 32 and 64 workers, we see that there is a significant performance increase between the second and third largest matrix sizes. For 64 workers, there is also a significant performance increase between the two largest matrix sizes. We therefore conjecture that we run out of system memory for 32 and 64 workers before we have reached peak performance. If that is correct, then adding more memory to the computers would both allow us to solve larger problems and perform better at those larger matrix sizes.

Speedup

The traditional way of measuring speedup obtained with linear algebra algorithms such as backslash is to compare the peak performance. We therefore calculate the maximum number of Gigaflops achieved for each number of workers.

```
peakPerf = [max(s.workers4.gflops), max(s.workers8.gflops), ...
            max(s.workers16.gflops), max(s.workers32.gflops), ...
            max(s.workers64.gflops)];
disp('Peak performance in Gigaflops for 4-64 workers:')
disp(peakPerf)

disp('Speedup when going from 4 workers to 8, 16, 32 and 64 workers:')
disp(peakPerf(2:end)/peakPerf(1))

Peak performance in Gigaflops for 4-64 workers:
   10.9319   23.2508   40.7157   73.5109  147.0693

Speedup when going from 4 workers to 8, 16, 32 and 64 workers:
    2.1269    3.7245    6.7244   13.4532
```

We therefore conclude that we get a speedup of approximately 13.5 when increasing the number of workers 16 fold, going from 4 workers to 64. As we noted above, the performance graph indicates that we might be able to increase the performance on 64 workers (and thereby improve the speedup even further), by increasing the system memory on the cluster computers.

The Cluster Used

This data was generated using 16 dual-processor, octa-core computers, each with 64 GB of memory, connected with GigaBit Ethernet. When using 4 workers, they were all on a single computer. We used 2 computers for 8 workers, 4 computers for 16 workers, etc.

Re-enabling the Deadlock Detection

Now that we have concluded our benchmarking, we can safely re-enable the deadlock detection in the current parallel pool.

```
pctRunOnAll 'mpiSettings(''DeadlockDetection'', ''on'');'

end


ans =

  struct with fields:

    matSize: [1000 19787 38573 57360 76146]
     gflops: [1.1618 43.7417 69.6855 40.4145 36.9379]
```

Benchmarking A\b on the GPU

This example looks at how we can benchmark the solving of a linear system on the GPU. The MATLAB® code to solve for x in A*x = b is very simple. Most frequently, we use matrix left division, also known as mldivide or the backslash operator (\), to calculate x (that is, x = A\b).

Related examples:

- "Benchmarking A\b" on page 9-151 using distributed arrays.

The code shown in this example can be found in this function:

```
function results = paralleldemo_gpu_backslash(maxMemory)
```

It is important to choose the appropriate matrix size for the computations. We can do this by specifying the amount of system memory in GB available to the CPU and the GPU. The default value is based only on the amount of memory available on the GPU, and you can specify a value that is appropriate for your system.

```
if nargin == 0
    g = gpuDevice;
    maxMemory = 0.4*g.AvailableMemory/1024^3;
end
```

The Benchmarking Function

We want to benchmark matrix left division (\), and not the cost of transferring data between the CPU and GPU, the time it takes to create a matrix, or other parameters. We therefore separate the data generation from the solving of the linear system, and measure only the time it takes to do the latter.

```
function [A, b] = getData(n, clz)
    fprintf('Creating a matrix of size %d-by-%d.\n', n, n);
    A = rand(n, n, clz) + 100*eye(n, n, clz);
    b = rand(n, 1, clz);
end

function time = timeSolve(A, b, waitFcn)
    tic;
    x = A\b; %#ok<NASGU> We don't need the value of x.
    waitFcn(); % Wait for operation to complete.
    time = toc;
end
```

Choosing Problem Size

As with a great number of other parallel algorithms, the performance of solving a linear system in parallel depends greatly on the matrix size. As seen in other examples, such as "Benchmarking A\b" on page 9-151, we compare the performance of the algorithm for different matrix sizes.

```
% Declare the matrix sizes to be a multiple of 1024.
maxSizeSingle = floor(sqrt(maxMemory*1024^3/4));
maxSizeDouble = floor(sqrt(maxMemory*1024^3/8));
step = 1024;
if maxSizeDouble/step >= 10
    step = step*floor(maxSizeDouble/(5*step));
end
```

```
sizeSingle = 1024:step:maxSizeSingle;
sizeDouble = 1024:step:maxSizeDouble;
```

Comparing Performance: Gigaflops

We use the number of floating point operations per second as our measure of performance because that allows us to compare the performance of the algorithm for different matrix sizes.

Given a matrix size, the benchmarking function creates the matrix A and the right-hand side b once, and then solves A\b a few times to get an accurate measure of the time it takes. We use the floating point operations count of the HPC Challenge, so that for an n-by-n matrix, we count the floating point operations as 2/3*n^3 + 3/2*n^2.

The function is passed in a handle to a 'wait' function. On the CPU, this function does nothing. On the GPU, this function waits for all pending operations to complete. Waiting in this way ensures accurate timing.

```
function gflops = benchFcn(A, b, waitFcn)
    numReps = 3;
    time = inf;
    % We solve the linear system a few times and calculate the Gigaflops
    % based on the best time.
    for itr = 1:numReps
        tcurr = timeSolve(A, b, waitFcn);
        time = min(tcurr, time);
    end

    % Measure the overhead introduced by calling the wait function.
    tover = inf;
    for itr = 1:numReps
        tic;
        waitFcn();
        tcurr = toc;
        tover = min(tcurr, tover);
    end
    % Remove the overhead from the measured time. Don't allow the time to
    % become negative.
    time = max(time - tover, 0);
    n = size(A, 1);
    flop = 2/3*n^3 + 3/2*n^2;
    gflops = flop/time/1e9;
end

% The CPU doesn't need to wait: this function handle is a placeholder.
function waitForCpu()
end

% On the GPU, to ensure accurate timing, we need to wait for the device
% to finish all pending operations.
function waitForGpu(theDevice)
    wait(theDevice);
end
```

Executing the Benchmarks

Having done all the setup, it is straightforward to execute the benchmarks. However, the computations can take a long time to complete, so we print some intermediate status information as

we complete the benchmarking for each matrix size. We also encapsulate the loop over all the matrix sizes in a function, to benchmark both single- and double-precision computations.

```
function [gflopsCPU, gflopsGPU] = executeBenchmarks(clz, sizes)
    fprintf(['Starting benchmarks with %d different %s-precision ' ...
        'matrices of sizes\nranging from %d-by-%d to %d-by-%d.\n'], ...
            length(sizes), clz, sizes(1), sizes(1), sizes(end), ...
            sizes(end));
    gflopsGPU = zeros(size(sizes));
    gflopsCPU = zeros(size(sizes));
    gd = gpuDevice;
    for i = 1:length(sizes)
        n = sizes(i);
        [A, b] = getData(n, clz);
        gflopsCPU(i) = benchFcn(A, b, @waitForCpu);
        fprintf('Gigaflops on CPU: %f\n', gflopsCPU(i));
        A = gpuArray(A);
        b = gpuArray(b);
        gflopsGPU(i) = benchFcn(A, b, @() waitForGpu(gd));
        fprintf('Gigaflops on GPU: %f\n', gflopsGPU(i));
    end
end
```

We then execute the benchmarks in single and double precision.

```
[cpu, gpu] = executeBenchmarks('single', sizeSingle);
results.sizeSingle = sizeSingle;
results.gflopsSingleCPU = cpu;
results.gflopsSingleGPU = gpu;
[cpu, gpu] = executeBenchmarks('double', sizeDouble);
results.sizeDouble = sizeDouble;
results.gflopsDoubleCPU = cpu;
results.gflopsDoubleGPU = gpu;
```

```
Starting benchmarks with 7 different single-precision matrices of sizes
ranging from 1024-by-1024 to 19456-by-19456.
Creating a matrix of size 1024-by-1024.
Gigaflops on CPU: 43.805496
Gigaflops on GPU: 78.474002
Creating a matrix of size 4096-by-4096.
Gigaflops on CPU: 96.459635
Gigaflops on GPU: 573.278854
Creating a matrix of size 7168-by-7168.
Gigaflops on CPU: 184.997657
Gigaflops on GPU: 862.755636
Creating a matrix of size 10240-by-10240.
Gigaflops on CPU: 204.404384
Gigaflops on GPU: 978.362901
Creating a matrix of size 13312-by-13312.
Gigaflops on CPU: 218.773070
Gigaflops on GPU: 1107.983667
Creating a matrix of size 16384-by-16384.
Gigaflops on CPU: 233.529176
Gigaflops on GPU: 1186.423754
Creating a matrix of size 19456-by-19456.
Gigaflops on CPU: 241.482550
Gigaflops on GPU: 1199.151846
Starting benchmarks with 5 different double-precision matrices of sizes
```

```
ranging from 1024-by-1024 to 13312-by-13312.
Creating a matrix of size 1024-by-1024.
Gigaflops on CPU: 34.902918
Gigaflops on GPU: 72.191488
Creating a matrix of size 4096-by-4096.
Gigaflops on CPU: 74.458136
Gigaflops on GPU: 365.339897
Creating a matrix of size 7168-by-7168.
Gigaflops on CPU: 93.313782
Gigaflops on GPU: 522.514165
Creating a matrix of size 10240-by-10240.
Gigaflops on CPU: 104.219804
Gigaflops on GPU: 628.301313
Creating a matrix of size 13312-by-13312.
Gigaflops on CPU: 108.826886
Gigaflops on GPU: 681.881032
```

Plotting the Performance

We can now plot the results, and compare the performance on the CPU and the GPU, both for single and double precision.

First, we look at the performance of the backslash operator in single precision.

```
fig = figure;
ax = axes('parent', fig);
plot(ax, results.sizeSingle, results.gflopsSingleGPU, '-x', ...
     results.sizeSingle, results.gflopsSingleCPU, '-o')
grid on;
legend('GPU', 'CPU', 'Location', 'NorthWest');
title(ax, 'Single-precision performance')
ylabel(ax, 'Gigaflops');
xlabel(ax, 'Matrix size');
drawnow;
```
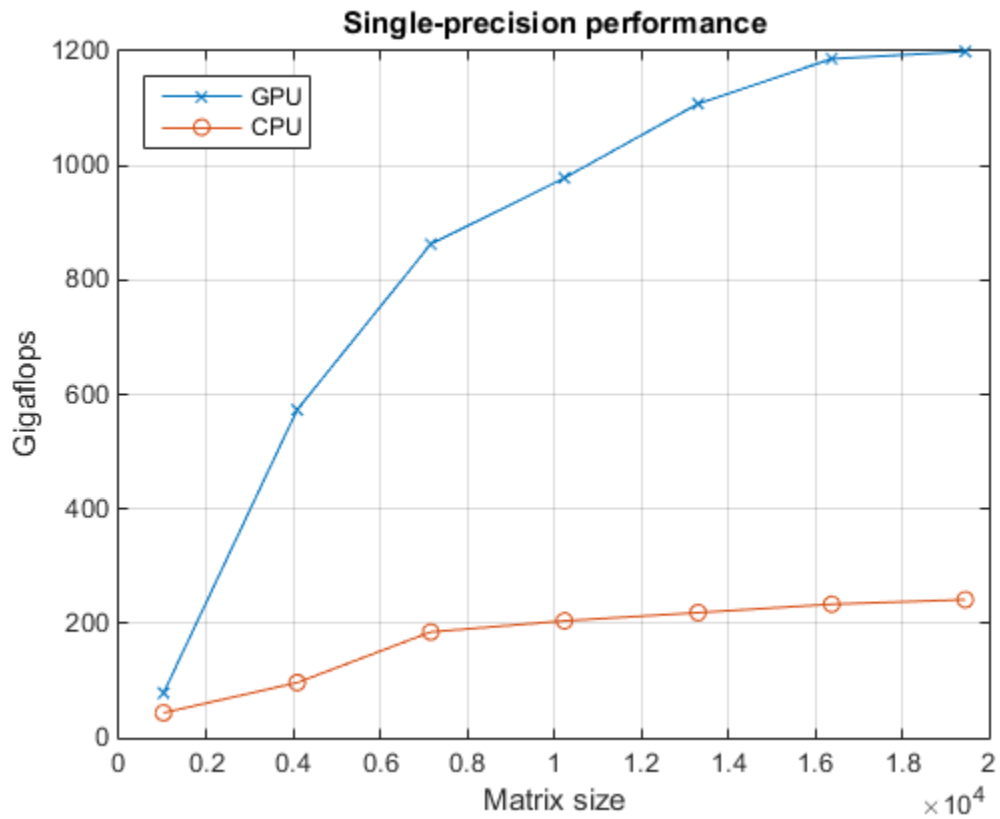
Now, we look at the performance of the backslash operator in double precision.

```
fig = figure;
ax = axes('parent', fig);
plot(ax, results.sizeDouble, results.gflopsDoubleGPU, '-x', ...
     results.sizeDouble, results.gflopsDoubleCPU, '-o')
legend('GPU', 'CPU', 'Location', 'NorthWest');
grid on;
title(ax, 'Double-precision performance')
ylabel(ax, 'Gigaflops');
xlabel(ax, 'Matrix size');
drawnow;
```

Double-precision performance
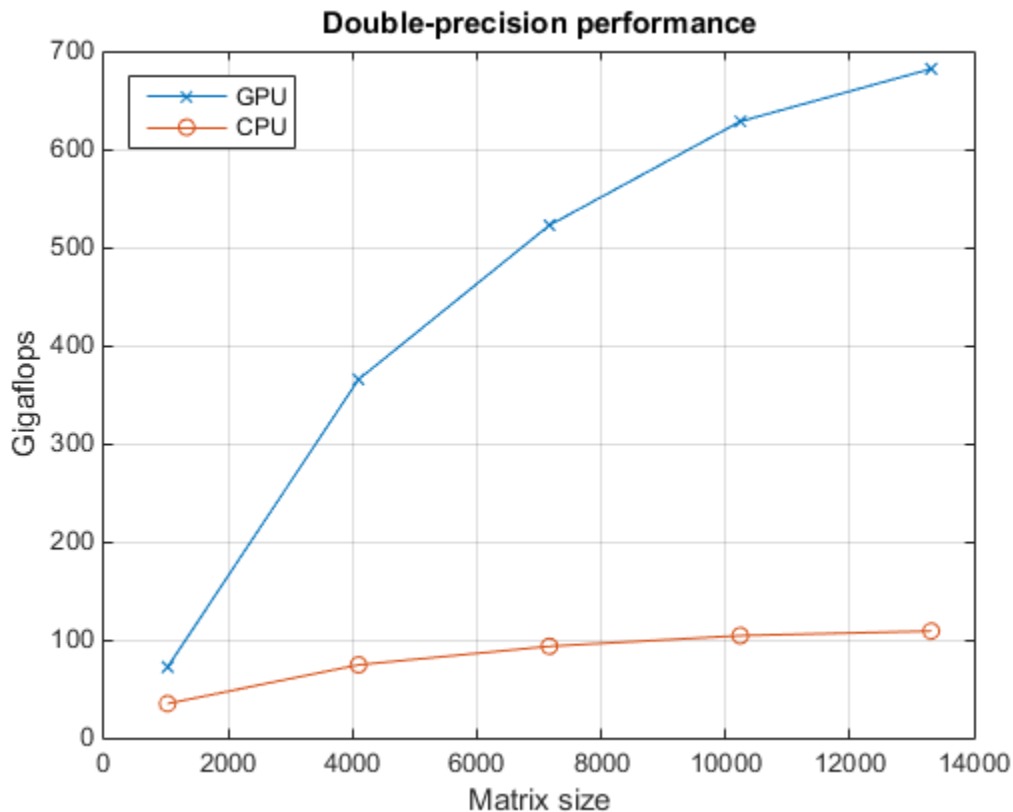
Finally, we look at the speedup of the backslash operator when comparing the GPU to the CPU.

```
speedupDouble = results.gflopsDoubleGPU./results.gflopsDoubleCPU;
speedupSingle = results.gflopsSingleGPU./results.gflopsSingleCPU;
fig = figure;
ax = axes('parent', fig);
plot(ax, results.sizeSingle, speedupSingle, '-v', ...
    results.sizeDouble, speedupDouble, '-*')
grid on;
legend('Single-precision', 'Double-precision', 'Location', 'SouthEast');
title(ax, 'Speedup of computations on GPU compared to CPU');
ylabel(ax, 'Speedup');
xlabel(ax, 'Matrix size');
drawnow;
```

Speedup of computations on GPU compared to CPU
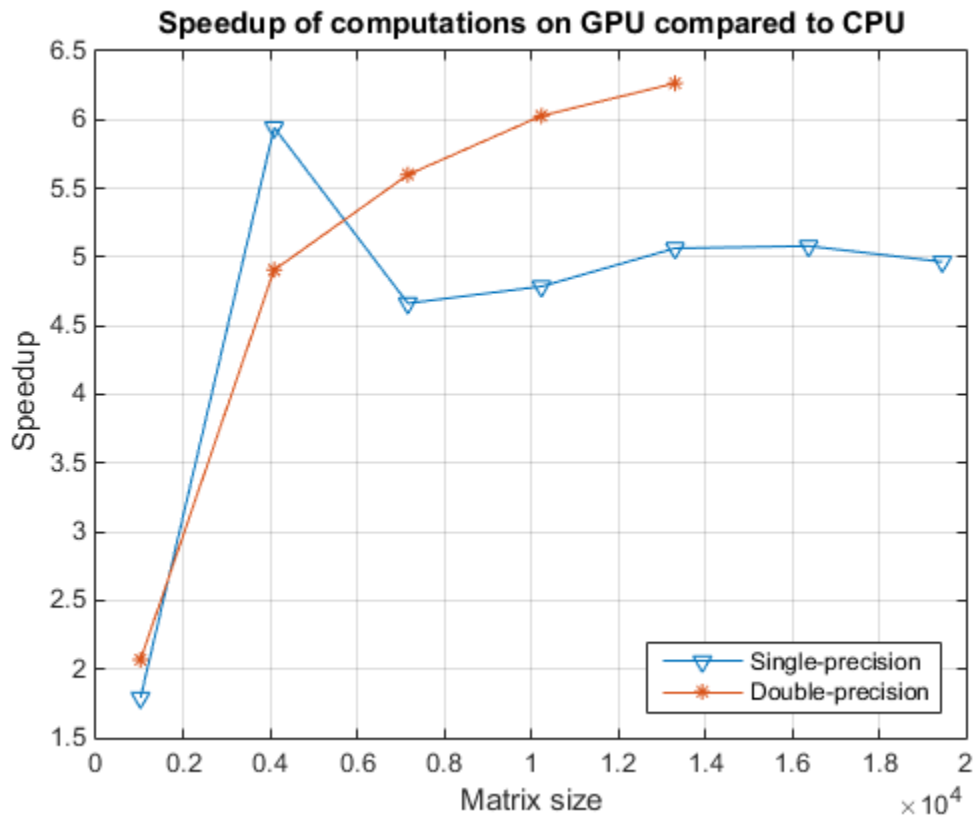
```
end
```

```
ans =

        sizeSingle: [1024 4096 7168 10240 13312 16384 19456]
    gflopsSingleCPU: [1x7 double]
    gflopsSingleGPU: [1x7 double]
        sizeDouble: [1024 4096 7168 10240 13312]
    gflopsDoubleCPU: [34.9029 74.4581 93.3138 104.2198 108.8269]
    gflopsDoubleGPU: [72.1915 365.3399 522.5142 628.3013 681.8810]
```

Using FFT2 on the GPU to Simulate Diffraction Patterns

This example uses Parallel Computing Toolbox™ to perform a two-dimensional Fast Fourier Transform (FFT) on a GPU. The two-dimensional Fourier transform is used in optics to calculate far-field diffraction patterns. These diffraction patterns are observed when a monochromatic light source passes through a small aperture, such as in Young's double-slit experiment.

Defining the Coordinate System

Before we simulate the light that has passed through an aperture, we must define our coordinate system. To get the correct numerical behavior when we call `fft2`, we must carefully arrange x and y so that the zero value is in the correct place.

N2 is half the size in each dimension.

```
N2 = 1024;
[gx, gy] = meshgrid( gpuArray.colon( -1, 1/N2, (N2-1)/N2 ) );
```

Simulating the Diffraction Pattern for a Rectangular Aperture

We simulate the effect of passing a parallel beam of monochromatic light through a small rectangular aperture. The two-dimensional Fourier transform describes the light field at a large distance from the aperture. We start by forming `aperture` as a logical mask based on the coordinate system, then the light source is simply a double-precision version of the aperture. The far-field light signal is found using `fft2`.

```
aperture      = ( abs(gx) < 4/N2 ) .* ( abs(gy) < 2/N2 );
lightsource   = double( aperture );
farfieldsignal = fft2( lightsource );
```

Displaying the Light Intensity for a Rectangular Aperture

We calculate the far-field light intensity from the magnitude squared of the light field. Finally, we use `fftshift` to aid visualization.

```
farfieldintensity = real( farfieldsignal .* conj( farfieldsignal ) );

imagesc( fftshift( farfieldintensity ) );
axis( 'equal' ); axis( 'off' );
title( 'Rectangular aperture far-field diffraction pattern' );
```

Rectangular aperture far-field diffraction pattern
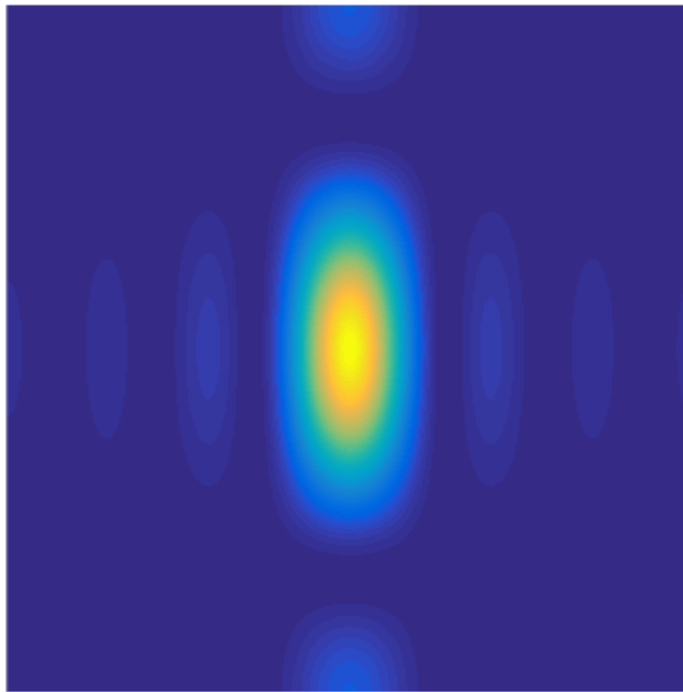
Simulating Young's Double-Slit Experiment

One of the most famous experiments in optics is Young's double-slit experiment which shows light interference when an aperture comprises two parallel slits. A series of bright points is visible where constructive interference takes place. In this case, we form the aperture representing two slits. We restrict the aperture in the y direction to ensure that the resulting pattern is not entirely concentrated along the horizontal axis.

```
slits         = (abs( gx ) <= 10/N2) .* (abs( gx ) >= 8/N2);
aperture      = slits .* (abs(gy) < 20/N2);
lightsource   = double( aperture );
farfieldsignal = fft2( lightsource );
```

Displaying the Light Intensity for Young's Double-Slit

We calculate and display the intensity as before.

```
farfieldintensity = real( farfieldsignal .* conj( farfieldsignal ) );
imagesc( fftshift( farfieldintensity ) );
axis( 'equal' ); axis( 'off' );
title( 'Double slit far-field diffraction pattern' );
```

Double slit far-field diffraction pattern
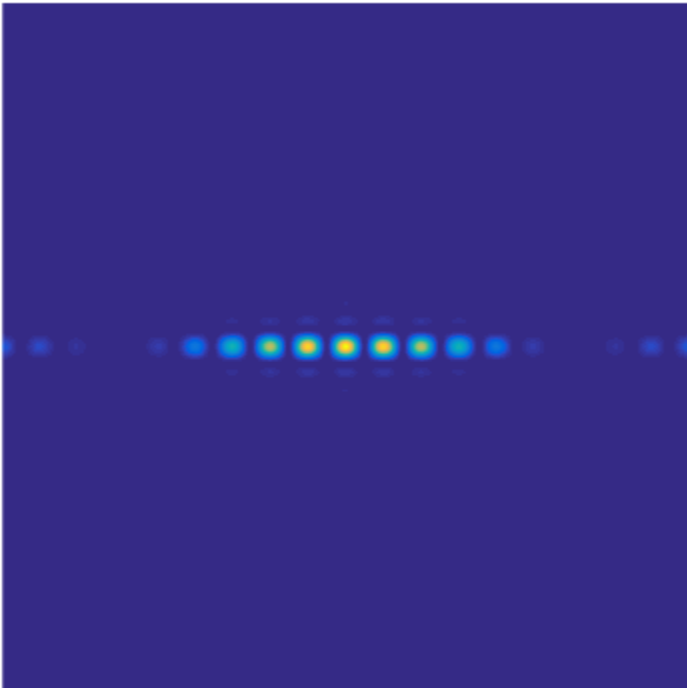
Improve Performance of Element-wise MATLAB® Functions on the GPU using ARRAYFUN

This example shows how `arrayfun` can be used to run a MATLAB® function natively on the GPU. When the MATLAB function contains many element-wise operations, `arrayfun` can provide improved performance when compared to simply executing the MATLAB function directly on the GPU with gpuArray input data. The MATLAB function can be in its own file or can be a nested or anonymous function. It must contain only scalar operations and arithmetic.

We put the example into a function to allow nested functions:

```
function paralleldemo_gpu_arrayfun
```

Using Horner's Rule to Calculate Exponentials

Horner's rule allows the efficient evaluation of power series expansions. We will use it to calculate the first 10 terms of the power series expansion for the exponential function `exp`. We can implement this as a MATLAB function.

```
function y = horner(x)
%HORNER - series expansion for exp(x) using Horner's rule
y = 1 + x.*(1 + x.*((1 + x.*((1 + ...
        x.*((1 + x.*((1 + x.*((1 + x.*((1 + ...
        x.*((1 + x./9)./8))./7))./6))./5))./4))./3))./2));
end
```

Preparing horner for the GPU

To run this function on the GPU with minimal code changes, we could pass a `gpuArray` object as input to the `horner` function. Since `horner` contains only individual element-wise operations, we might not realize very good performance on the GPU when performing each operation one at a time. However, we can improve the performance by executing all of the element-wise operations in the `horner` function at one time using `arrayfun`.

To run this function on the GPU using `arrayfun`, we use a handle to the `horner` function. `horner` automatically adapts to different size and type inputs. We can compare the results computed on the GPU using both `gpuArray` objects and `arrayfun` with standard MATLAB CPU execution simply by evaluating the function directly.

```
hornerFcn = @horner;
```

Create the Input Data

We create some inputs of different types and sizes, and use `gpuArray` to send them to the GPU.

```
data1  = rand( 2000, 'single' );
data2  = rand( 1000, 'double' );
gdata1 = gpuArray( data1 );
gdata2 = gpuArray( data2 );
```

Evaluate horner on the GPU

To evaluate the `horner` function on the GPU, we have two choices. With minimal code changes we can evaluate the original function on the GPU by providing a `gpuArray` object as input. However, to improve the performance on the GPU call `arrayfun`, using the same calling convention as the original MATLAB function.

We can compare the accuracy of the results by evaluating the original function directly in MATLAB on the CPU. We expect some slight numerical differences because the floating-point arithmetic on the GPU does not precisely match the arithmetic performed on the CPU.

```
gresult1 = arrayfun( hornerFcn, gdata1 );
gresult2 = arrayfun( hornerFcn, gdata2 );

comparesingle = max( max( abs( gresult1 - horner( data1 ) ) ) );
comparedouble = max( max( abs( gresult2 - horner( data2 ) ) ) );

fprintf( 'Maximum discrepancy for single precision: %g\n', comparesingle );
fprintf( 'Maximum discrepancy for double precision: %g\n', comparedouble );

Maximum discrepancy for single precision: 2.38419e-07
Maximum discrepancy for double precision: 0
```

Comparing Performance between GPU and CPU

We can compare the performance of the GPU versions to the native MATLAB CPU version. Current generation GPUs have much better performance in single precision, so we compare that.

```
% CPU execution
tic
hornerFcn( data1 );
tcpu = toc;

% GPU execution using only gpuArray objects
tgpuObject = gputimeit(@() hornerFcn(gdata1));

% GPU execution using gpuArray objects with arrayfun
tgpuArrayfun = gputimeit(@() arrayfun(hornerFcn, gdata1));


fprintf( 'Speed-up achieved using gpuArray objects only: %g\n',...
    tcpu / tgpuObject );
fprintf( 'Speed-up achieved using gpuArray objects with arrayfun: %g\n',...
    tcpu / tgpuArrayfun );

Speed-up achieved using gpuArray objects only: 24.6764
Speed-up achieved using gpuArray objects with arrayfun: 98.3555

end
```

Measuring GPU Performance

This example shows how to measure some of the key performance characteristics of a GPU.

GPUs can be used to speed up certain types of computations. However, GPU performance varies widely between different GPU devices. In order to quantify the performance of a GPU, three tests are used:

- How quickly can data be sent to the GPU or read back from it?
- How fast can the GPU kernel read and write data?
- How fast can the GPU perform computations?

After measuring these, the performance of the GPU can be compared to the host CPU. This provides a guide as to how much data or computation is required for the GPU to provide an advantage over the CPU.

Setup

```
gpu = gpuDevice();
fprintf('Using a %s GPU.\n', gpu.Name)
sizeOfDouble = 8; % Each double-precision number needs 8 bytes of storage
sizes = power(2, 14:28);
```

```
Using a Tesla K40c GPU.
```

Testing host/GPU bandwidth

The first test estimates how quickly data can be sent to and read from the GPU. Because the GPU is plugged into the PCI bus, this largely depends on how fast the PCI bus is and how many other things are using it. However, there are also some overheads that are included in the measurements, particularly the function call overhead and the array allocation time. Since these are present in any "real world" use of the GPU, it is reasonable to include these.

In the following tests, memory is allocated and data is sent to the GPU using the `gpuArray` function. Memory is allocated and data is transferred back to host memory using `gather`.

Note that PCI express v3, as used in this test, has a theoretical bandwidth of 0.99GB/s per lane. For the 16-lane slots (PCIe3 x16) used by NVIDIA's compute cards this gives a theoretical 15.75GB/s.

```
sendTimes = inf(size(sizes));
gatherTimes = inf(size(sizes));
for ii=1:numel(sizes)
    numElements = sizes(ii)/sizeOfDouble;
    hostData = randi([0 9], numElements, 1);
    gpuData = randi([0 9], numElements, 1, 'gpuArray');
    % Time sending to GPU
    sendFcn = @() gpuArray(hostData);
    sendTimes(ii) = gputimeit(sendFcn);
    % Time gathering back from GPU
    gatherFcn = @() gather(gpuData);
    gatherTimes(ii) = gputimeit(gatherFcn);
end
sendBandwidth = (sizes./sendTimes)/1e9;
[maxSendBandwidth,maxSendIdx] = max(sendBandwidth);
fprintf('Achieved peak send speed of %g GB/s\n',maxSendBandwidth)
gatherBandwidth = (sizes./gatherTimes)/1e9;
```

```
[maxGatherBandwidth,maxGatherIdx] = max(gatherBandwidth);
fprintf('Achieved peak gather speed of %g GB/s\n',max(gatherBandwidth))
```

```
Achieved peak send speed of 6.18519 GB/s
Achieved peak gather speed of 3.31891 GB/s
```

On the plot below, the peak for each case is circled. With small data set sizes, overheads dominate. With larger amounts of data the PCI bus is the limiting factor.

```
hold off
semilogx(sizes, sendBandwidth, 'b.-', sizes, gatherBandwidth, 'r.-')
hold on
semilogx(sizes(maxSendIdx), maxSendBandwidth, 'bo-', 'MarkerSize', 10);
semilogx(sizes(maxGatherIdx), maxGatherBandwidth, 'ro-', 'MarkerSize', 10);
grid on
title('Data Transfer Bandwidth')
xlabel('Array size (bytes)')
ylabel('Transfer speed (GB/s)')
legend('Send to GPU', 'Gather from GPU', 'Location', 'NorthWest')
```
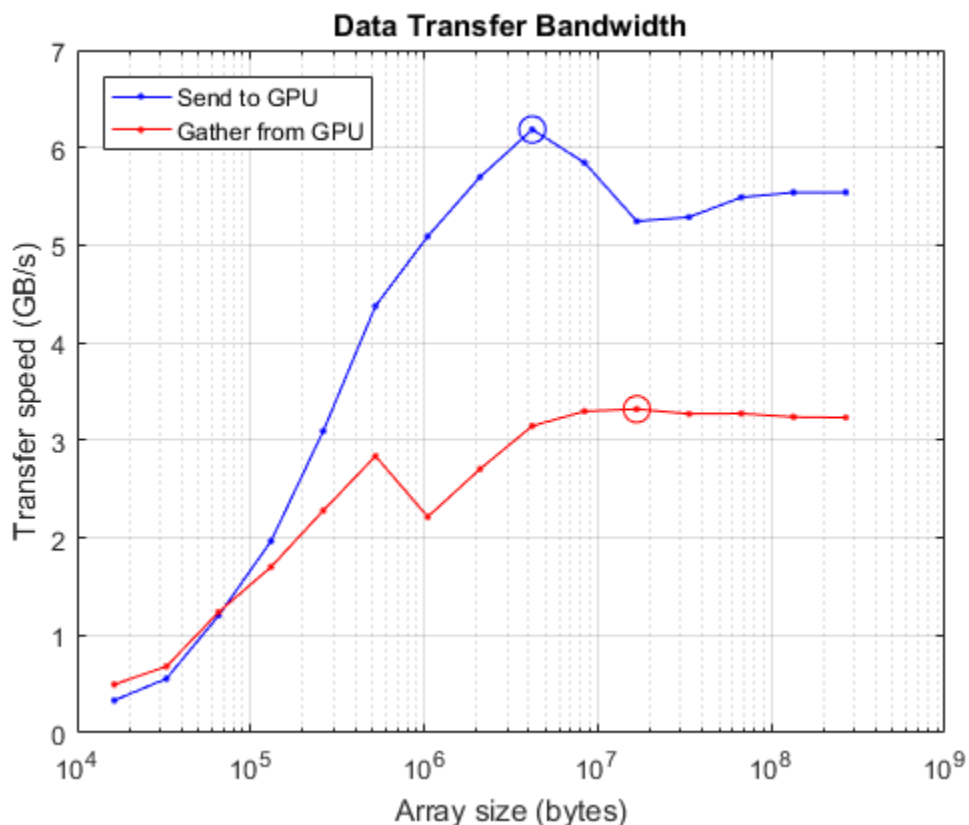

Testing memory intensive operations

Many operations do very little computation with each element of an array and are therefore dominated by the time taken to fetch the data from memory or to write it back. Functions such as `ones`, `zeros`, `nan`, `true` only write their output, whereas functions like `transpose`, `tril` both read and write but do no computation. Even simple operators like `plus`, `minus`, `mtimes` do so little computation per element that they are bound only by the memory access speed.

The function plus performs one memory read and one memory write for each floating point operation. It should therefore be limited by memory access speed and provides a good indicator of the speed of a read+write operation.

```
memoryTimesGPU = inf(size(sizes));
for ii=1:numel(sizes)
    numElements = sizes(ii)/sizeOfDouble;
    gpuData = randi([0 9], numElements, 1, 'gpuArray');
    plusFcn = @() plus(gpuData, 1.0);
    memoryTimesGPU(ii) = gputimeit(plusFcn);
end
memoryBandwidthGPU = 2*(sizes./memoryTimesGPU)/1e9;
[maxBWGPU, maxBWIdxGPU] = max(memoryBandwidthGPU);
fprintf('Achieved peak read+write speed on the GPU: %g GB/s\n',maxBWGPU)
```

Achieved peak read+write speed on the GPU: 186.494 GB/s

Now compare it with the same code running on the CPU.

```
memoryTimesHost = inf(size(sizes));
for ii=1:numel(sizes)
    numElements = sizes(ii)/sizeOfDouble;
    hostData = randi([0 9], numElements, 1);
    plusFcn = @() plus(hostData, 1.0);
    memoryTimesHost(ii) = timeit(plusFcn);
end
memoryBandwidthHost = 2*(sizes./memoryTimesHost)/1e9;
[maxBWHost, maxBWIdxHost] = max(memoryBandwidthHost);
fprintf('Achieved peak read+write speed on the host: %g GB/s\n',maxBWHost)

% Plot CPU and GPU results.
hold off
semilogx(sizes, memoryBandwidthGPU, 'b.-', ...
    sizes, memoryBandwidthHost, 'r.-')
hold on
semilogx(sizes(maxBWIdxGPU), maxBWGPU, 'bo-', 'MarkerSize', 10);
semilogx(sizes(maxBWIdxHost), maxBWHost, 'ro-', 'MarkerSize', 10);
grid on
title('Read+write Bandwidth')
xlabel('Array size (bytes)')
ylabel('Speed (GB/s)')
legend('GPU', 'Host', 'Location', 'NorthWest')
```

Achieved peak read+write speed on the host: 40.2573 GB/s
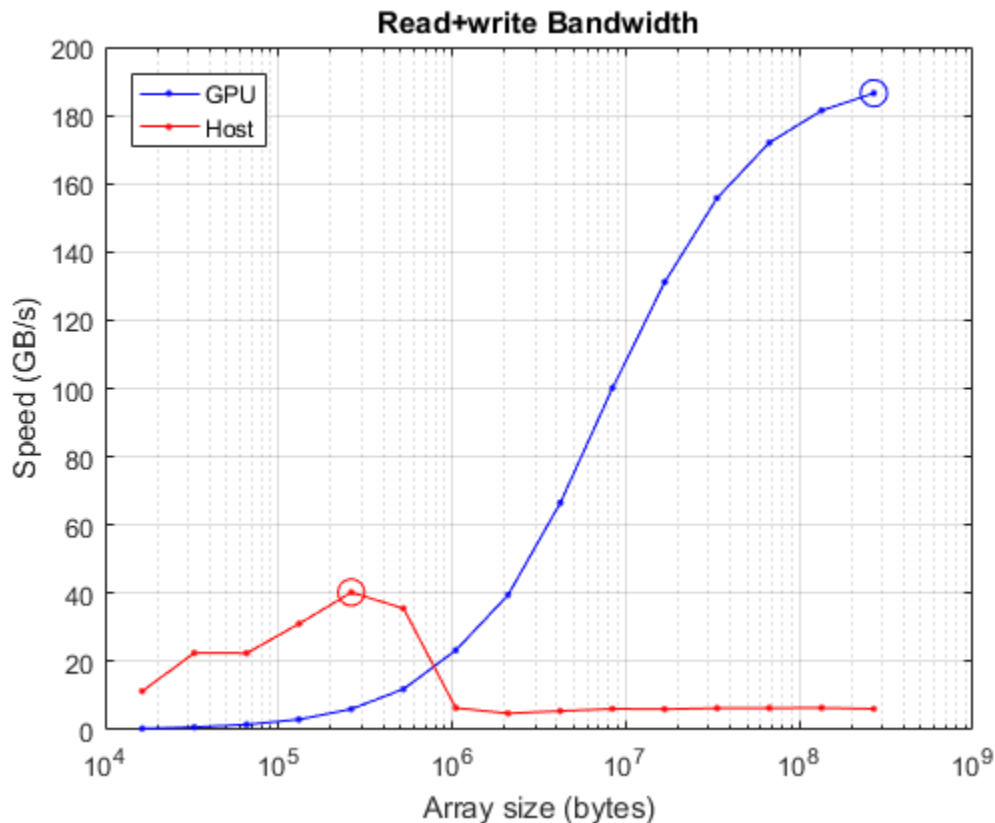
Comparing this plot with the data-transfer plot above, it is clear that GPUs can typically read from and write to their memory much faster than they can get data from the host. It is therefore important to minimize the number of host-GPU or GPU-host memory transfers. Ideally, programs should transfer the data to the GPU, then do as much with it as possible while on the GPU, and bring it back to the host only when complete. Even better would be to create the data on the GPU to start with.

Testing computationally intensive operations

For operations where the number of floating-point computations performed per element read from or written to memory is high, the memory speed is much less important. In this case the number and speed of the floating-point units is the limiting factor. These operations are said to have high "computational density".

A good test of computational performance is a matrix-matrix multiply. For multiplying two $N \times N$ matrices, the total number of floating-point calculations is

$$FLOPS(N) = 2N^3 - N^2.$$

Two input matrices are read and one resulting matrix is written, for a total of $3N^2$ elements read or written. This gives a computational density of (2N - 1)/3 FLOP/element. Contrast this with `plus` as used above, which has a computational density of 1/2 FLOP/element.

```
sizes = power(2, 12:2:24);
N = sqrt(sizes);
mmTimesHost = inf(size(sizes));
mmTimesGPU = inf(size(sizes));
```

```
for ii=1:numel(sizes)
    % First do it on the host
    A = rand( N(ii), N(ii) );
    B = rand( N(ii), N(ii) );
    mmTimesHost(ii) = timeit(@() A*B);
    % Now on the GPU
    A = gpuArray(A);
    B = gpuArray(B);
    mmTimesGPU(ii) = gputimeit(@() A*B);
end
mmGFlopsHost = (2*N.^3 - N.^2)./mmTimesHost/1e9;
[maxGFlopsHost,maxGFlopsHostIdx] = max(mmGFlopsHost);
mmGFlopsGPU = (2*N.^3 - N.^2)./mmTimesGPU/1e9;
[maxGFlopsGPU,maxGFlopsGPUIdx] = max(mmGFlopsGPU);
fprintf(['Achieved peak calculation rates of ', ...
    '%1.1f GFLOPS (host), %1.1f GFLOPS (GPU)\n'], ...
    maxGFlopsHost, maxGFlopsGPU)
```

```
Achieved peak calculation rates of 72.5 GFLOPS (host), 1153.3 GFLOPS (GPU)
```

Now plot it to see where the peak was achieved.

```
hold off
semilogx(sizes, mmGFlopsGPU, 'b.-', sizes, mmGFlopsHost, 'r.-')
hold on
semilogx(sizes(maxGFlopsGPUIdx), maxGFlopsGPU, 'bo-', 'MarkerSize', 10);
semilogx(sizes(maxGFlopsHostIdx), maxGFlopsHost, 'ro-', 'MarkerSize', 10);
grid on
title('Double precision matrix-matrix multiply')
xlabel('Matrix size (numel)')
ylabel('Calculation Rate (GFLOPS)')
legend('GPU', 'Host', 'Location', 'NorthWest')
```
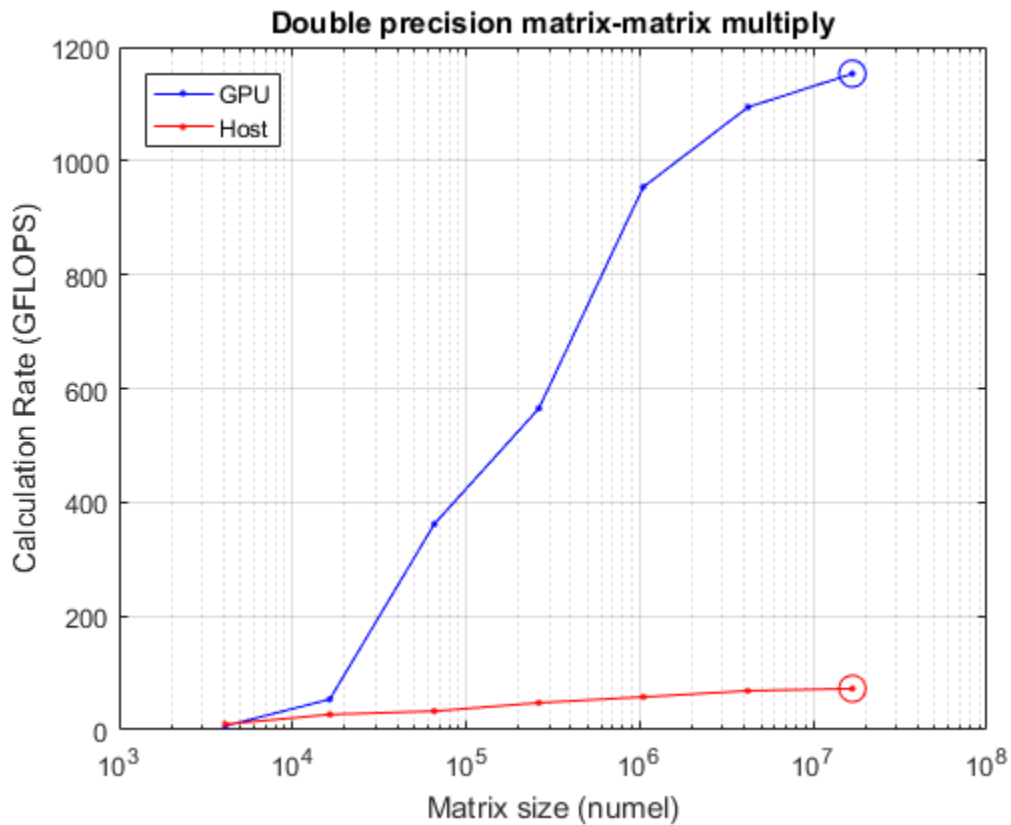
Conclusions

These tests reveal some important characteristics of GPU performance:

- Transfers from host memory to GPU memory and back are relatively slow.
- A good GPU can read/write its memory much faster than the host CPU can read/write its memory.
- Given large enough data, GPUs can perform calculations much faster than the host CPU.

It is notable that in each test quite large arrays were required to fully saturate the GPU, whether limited by memory or by computation. GPUs provide the greatest advantage when working with millions of elements at once.

More detailed GPU benchmarks, including comparisons between different GPUs, are available in GPUBench on the MATLAB® Central File Exchange.

Generating Random Numbers on a GPU

This example shows how to switch between the different random number generators that are supported on the GPU.

Random numbers form a key part of many simulation or estimation algorithms. Typically, these numbers are generated using the functions `rand`, `randi`, and `randn`. Parallel Computing Toolbox™ provides three corresponding functions for generating random numbers directly on a GPU: `rand`, `randi`, and `randn`. These functions can use one of several different number generation algorithms.

```
d = gpuDevice;
fprintf("This example is run on a " + d.Name + " GPU.")
```

This example is run on a GeForce GTX 1080 GPU.

Discovering the GPU random number generators

The function `parallel.gpu.RandStream.list` provides a short description of the available generators.

```
parallel.gpu.RandStream.list
```

The following random number generator algorithms are available:

```
MRG32K3A:        Combined multiple recursive generator (supports parallel streams)
Philox4x32_10:   Philox 4x32 generator with 10 rounds (supports parallel streams)
Threefry4x64_20: Threefry 4x64 generator with 20 rounds (supports parallel streams)
```

Each of these generators has been designed with parallel use in mind, providing multiple independent streams of random numbers. However, they each have some advantages and disadvantages:

- **CombRecursive** (also known as `MRG32k3a`): This generator was introduced in 1999 and has been widely tested and used.
- **Philox** (also known as Philox4x32_10): New generator introduced in 2011, specifically designed for high performance in highly parallel systems such as GPUs.
- **Threefry** (also known as Threefry4x64_20): New generator introduced in 2011 based on the existing cryptographic ThreeFish algorithm, which is widely tested and used. This generator was designed to give good performance in highly parallel systems such as GPUs. This is the default generator for GPU calculations.

The three generators available on the GPU are also available for use on the CPU in MATLAB®. The MATLAB generators have the same name and produce identical results given the same initial state. This is useful when you want to produce the same sets of random numbers on both the GPU and the CPU.

All of these generators pass the standard TestU01 test suite [1].

Changing the default random number generator

The function `gpurng` can store and reset the generator state for the GPU. You can also use `gpurng` to switch between the different generators that are provided. Before changing the generator, store the existing state so that it can be restored at the end of these tests.

9-177

```
oldState = gpurng;

gpurng(0, "Philox4x32-10");
disp(gpurng)

     Type: 'philox'
     Seed: 0
    State: [7×1 uint32]
```

Generating uniformly distributed random numbers

Uniformly distributed random numbers are generated on the GPU using either `rand`, or `randi`. In performance terms, these two functions behave very similarly and only `rand` is measured here. `gputimeit` is used to measure the performance to ensure accurate timing results, automatically calling the function many times and correctly dealing with synchronization and other timing issues.

To compare the performance of the different generators, use `rand` to generate a large number of random numbers on the GPU using each generator. In the following code, `rand` generates 10^7 random numbers and is called 100 times for each generator. Each run is timed using `gputimeit`. Generating large samples of random numbers can take several minutes. The results indicate a performance comparison between the three random number generators available on the GPU.

```
generators = ["Philox","Threefry","CombRecursive"];
gputimesU = nan(100,3);
for g=1:numel(generators)
    % Set the generator
    gpurng(0, generators{g});
    % Perform calculation 100 times, timing the generator
    for rep=1:100
        gputimesU(rep,g) = gputimeit(@() rand(10000,1000,"gpuArray"));
    end
end

% Plot the results
figure
hold on
histogram(gputimesU(:,1),"BinWidth",1e-4);
histogram(gputimesU(:,2),"BinWidth",1e-4);
histogram(gputimesU(:,3),"BinWidth",1e-4)

legend(generators)
xlabel("Time to generate 10^7 random numbers (sec)")
ylabel("Frequency")
title("Generating samples in U(0,1) using " + d.Name)
hold off
```
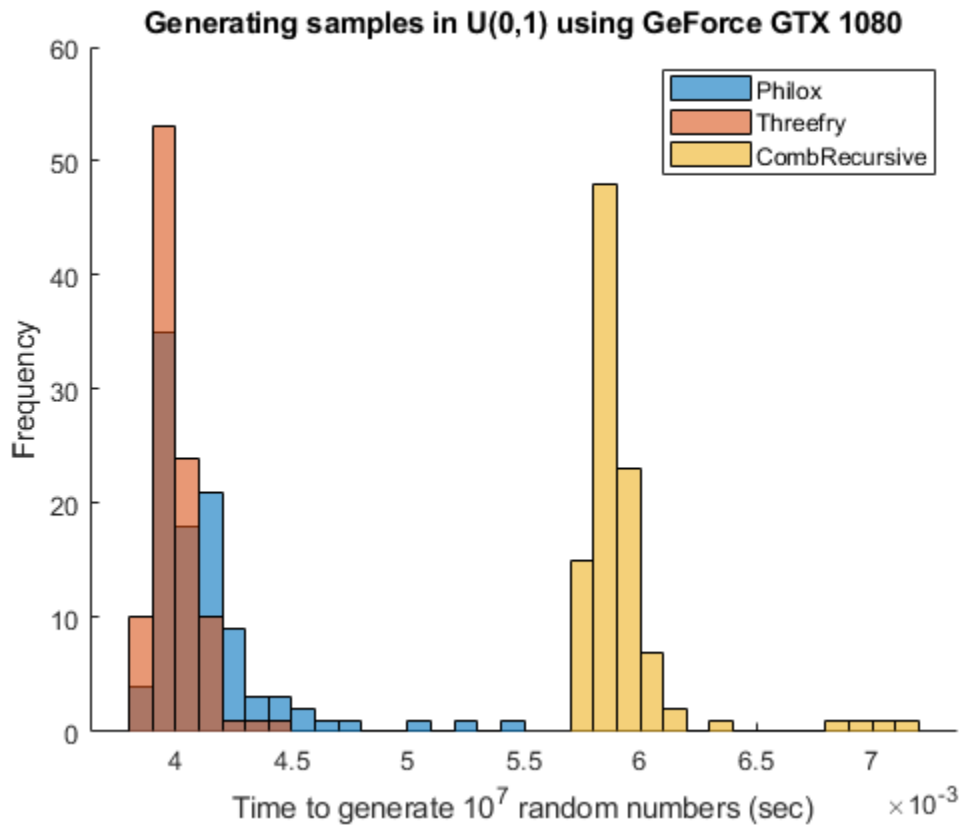
The newer generators Threefry and Philox have similar perfomance. Both are faster than CombRecursive.

Generating normally distributed random numbers

Many simulations rely on perturbations sampled from a normal distribution. Similar to the uniform test, use `randn` to compare the performance of the three generators when generating normally distributed random numbers. Generating large samples of random numbers can take several minutes.

```
generators = ["Philox","Threefry","CombRecursive"];
gputimesN = nan(100,3);
for g=1:numel(generators)
    % Set the generator
    gpurng(0, generators{g});
    % Perform calculation 100 times, timing the generator
    for rep=1:100
        gputimesN(rep,g) = gputimeit(@() randn(10000,1000,"gpuArray"));
    end
end

% Plot the results
figure
hold on
histogram(gputimesN(:,1),"BinWidth",1e-4);
histogram(gputimesN(:,2),"BinWidth",1e-4)
histogram(gputimesN(:,3),"BinWidth",1e-4)
legend(generators)
```

```
xlabel("Time to generate 10^7 random numbers (sec)")
ylabel("Frequency")
title("Generating samples in N(0,1) using " + d.Name)
hold off
```
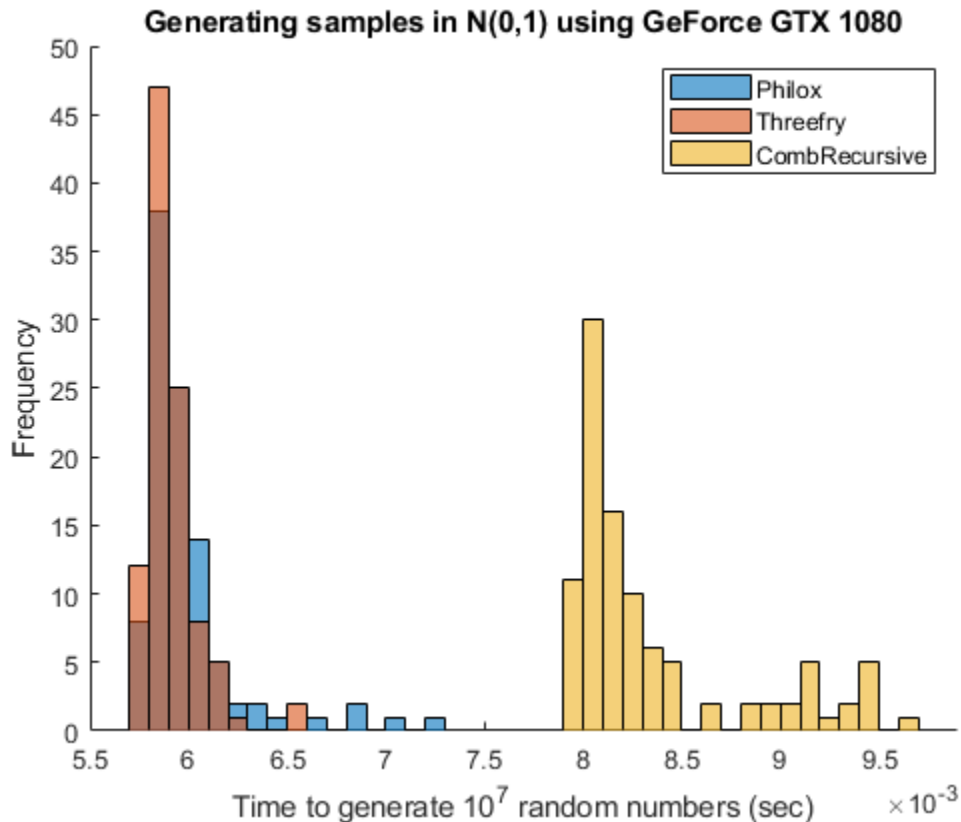

Once again, the results indicate that the Threefry and Philox generators perform similarly and are both notably faster than CombRecursive. The extra work required to produce normally distributed values reduces the rate at which values are produced by each of the generators.

Before finishing, restore the original generator state.

```
gpurng(oldState);
```

Conclusion

In this example, the three GPU random number generators are compared. The exact results vary depending on your GPU and computing platform. Each generator provides some advantages (+) and has some caveats (-).

Threefry

- (+) Fast
- (+) Based on well-known and well-tested Threefish algorithm
- (-) Relatively new in real-world usage

Philox

- (+) Fast
- (-) Relatively new in real-world usage

CombRecursive

- (+) Long track record in real-world usage
- (-) Slowest

References

[1] L'Ecuyer, P., and R. Simard. "TestU01: A C library for empirical testing of random number generators." A*CM Transactions on Mathematical Software.* Vol. 33, No. 4, 2007, article 22.

See Also
gpurng | parallel.gpu.RandStream

More About
- "Random Number Streams on a GPU" on page 8-6

Illustrating Three Approaches to GPU Computing: The Mandelbrot Set

This example shows how a simple, well-known mathematical problem, the Mandelbrot Set, can be expressed in MATLAB® code. Using Parallel Computing Toolbox™ this code is then adapted to make use of GPU hardware in three ways:

1   Using the existing algorithm but with GPU data as input
2   Using `arrayfun` to perform the algorithm on each element independently
3   Using the MATLAB/CUDA interface to run some existing CUDA/C++ code

Setup

The values below specify a highly zoomed part of the Mandelbrot Set in the valley between the main cardioid and the p/q bulb to its left.
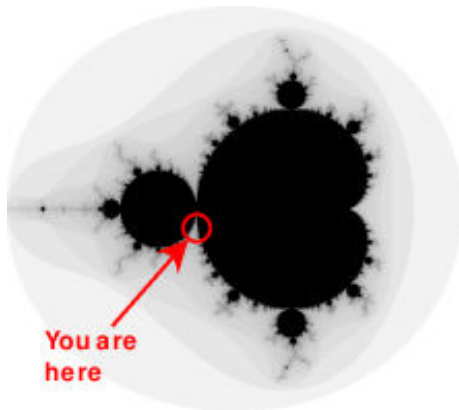
A 1000x1000 grid of real parts (X) and imaginary parts (Y) is created between these limits and the Mandelbrot algorithm is iterated at each grid location. For this particular location 500 iterations will be enough to fully render the image.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862,  0.123640851045266];
```

The Mandelbrot Set in MATLAB

Below is an implementation of the Mandelbrot Set using standard MATLAB commands running on the CPU. This is based on the code provided in Cleve Moler's "Experiments with MATLAB" e-book.

This calculation is vectorized such that every location is updated at once.

```
% Setup
t = tic();
x = linspace( xlim(1), xlim(2), gridSize );
y = linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );
z0 = xGrid + 1i*yGrid;
count = ones( size(z0) );
```

```matlab
% Calculate
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs( z )<=2;
    count = count + inside;
end
count = log( count );

% Show
cpuTime = toc( t );
fig = gcf;
fig.Position = [200 200 600 600];
imagesc( x, y, count );
colormap( [jet();flipud( jet() );0 0 0] );
axis off
title( sprintf( '%1.2fsecs (without GPU)', cpuTime ) );
```

6.84secs (without GPU)
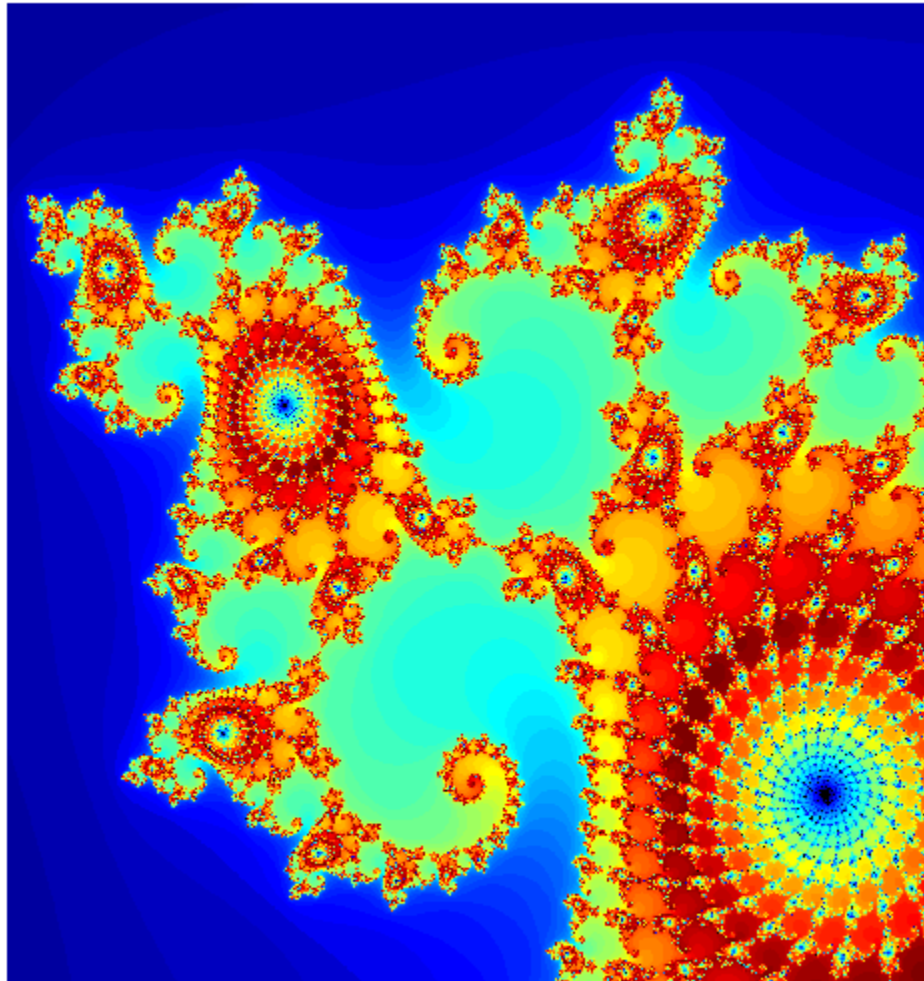
Using `gpuArray`

When MATLAB encounters data on the GPU, calculations with that data are performed on the GPU. The class `gpuArray` provides GPU versions of many functions that you can use to create data arrays, including the `linspace`, `logspace`, and `meshgrid` functions needed here. Similarly, the `count` array is initialized directly on the GPU using the function `ones`.

With these changes to the data initialization the calculations will now be performed on the GPU:

```
% Setup
t = tic();
x = gpuArray.linspace( xlim(1), xlim(2), gridSize );
y = gpuArray.linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );
z0 = complex( xGrid, yGrid );
```

```matlab
count = ones( size(z0), 'gpuArray' );

% Calculate
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs( z )<=2;
    count = count + inside;
end
count = log( count );

% Show
count = gather( count ); % Fetch the data back from the GPU
naiveGPUTime = toc( t );
imagesc( x, y, count )
axis off
title( sprintf( '%1.3fsecs (naive GPU) = %1.1fx faster', ...
    naiveGPUTime, cpuTime/naiveGPUTime ) )
```

0.249secs (naive GPU) = 27.4x faster
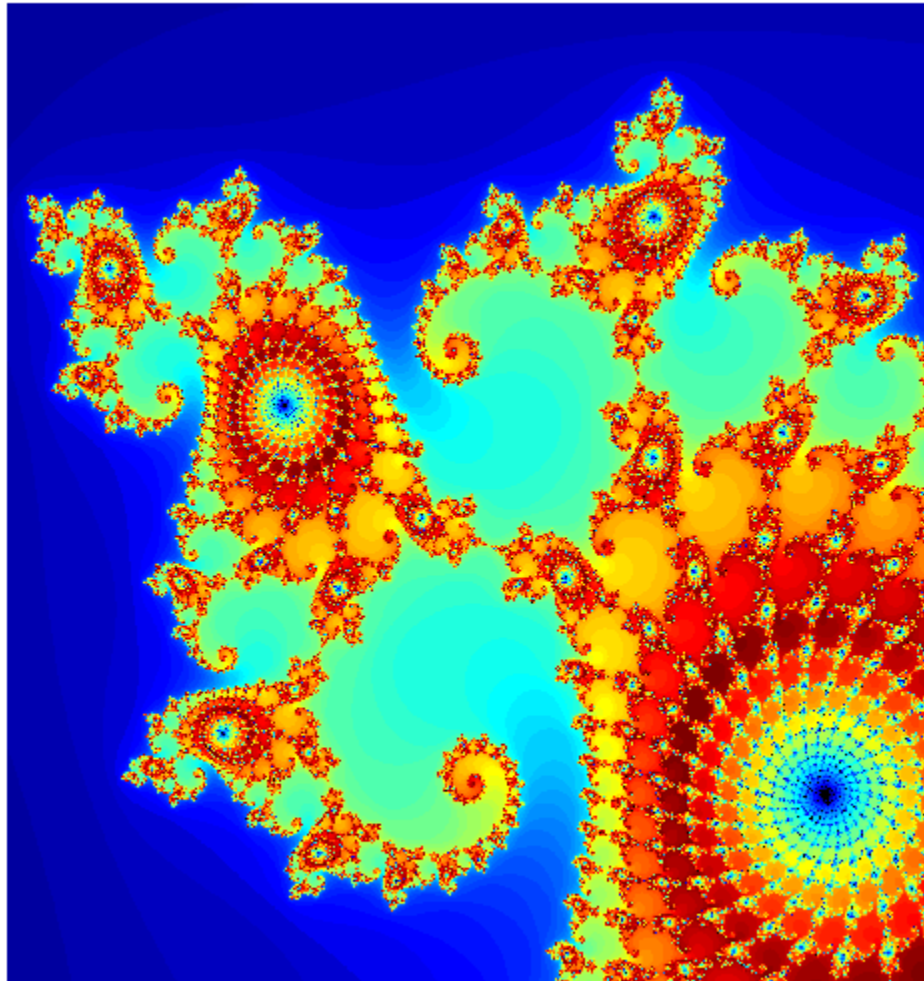
Element-wise Operation

Noting that the algorithm is operating equally on every element of the input, we can place the code in a helper function and call it using `arrayfun`. For GPU array inputs, the function used with `arrayfun` gets compiled into native GPU code. In this case we placed the loop in `pctdemo_processMandelbrotElement.m`:

```
function count = pctdemo_processMandelbrotElement(x0,y0,maxIterations)
z0 = complex(x0,y0);
z = z0;
count = 1;
while (count <= maxIterations) && (abs(z) <= 2)
    count = count + 1;
    z = z*z + z0;
```

```
end
count = log(count);
```

Note that an early abort has been introduced because this function processes only a single element. For most views of the Mandelbrot Set a significant number of elements stop very early and this can save a lot of processing. The `for` loop has also been replaced by a `while` loop because they are usually more efficient. This function makes no mention of the GPU and uses no GPU-specific features - it is standard MATLAB code.

Using `arrayfun` means that instead of many thousands of calls to separate GPU-optimized operations (at least 6 per iteration), we make one call to a parallelized GPU operation that performs the whole calculation. This significantly reduces overhead.

```
% Setup
t = tic();
x = gpuArray.linspace( xlim(1), xlim(2), gridSize );
y = gpuArray.linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );

% Calculate
count = arrayfun( @pctdemo_processMandelbrotElement, ...
                  xGrid, yGrid, maxIterations );

% Show
count = gather( count ); % Fetch the data back from the GPU
gpuArrayfunTime = toc( t );
imagesc( x, y, count )
axis off
title( sprintf( '%1.3fsecs (GPU arrayfun) = %1.1fx faster', ...
    gpuArrayfunTime, cpuTime/gpuArrayfunTime ) );
```

0.027secs (GPU arrayfun) = 258.0x faster
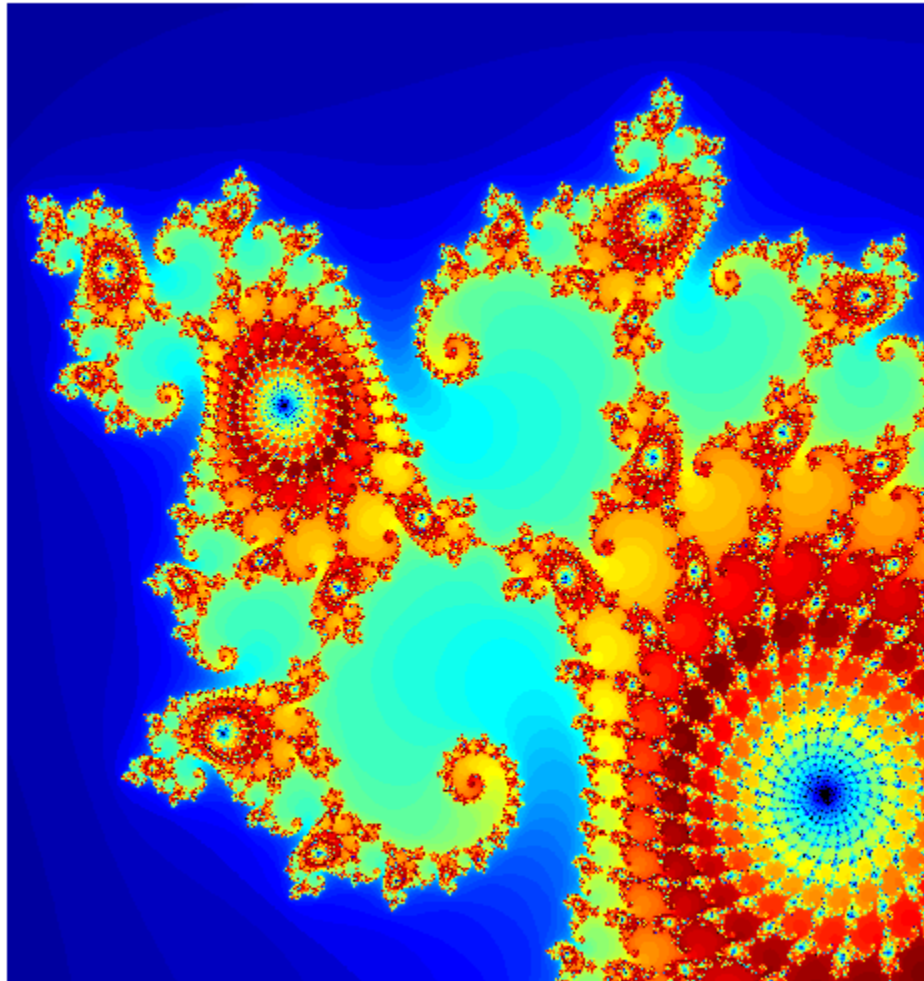
Working with CUDA

In Experiments in MATLAB improved performance is achieved by converting the basic algorithm to a C-Mex function. If you are willing to do some work in C/C++, then you can use Parallel Computing Toolbox to call pre-written CUDA kernels using MATLAB data. You do this with the `parallel.gpu.CUDAKernel` feature.

A CUDA/C++ implementation of the element processing algorithm has been hand-written in `pctdemo_processMandelbrotElement.cu`: This must then be manually compiled using nVidia's NVCC compiler to produce the assembly-level `pctdemo_processMandelbrotElement.ptx` (`.ptx` stands for "Parallel Thread eXecution language").

The CUDA/C++ code is a little more involved than the MATLAB versions we have seen so far, due to the lack of complex numbers in C++. However, the essence of the algorithm is unchanged:

```
__device__
unsigned int doIterations( double const realPart0,
                           double const imagPart0,
                           unsigned int const maxIters ) {
   // Initialize: z = z0
   double realPart = realPart0;
   double imagPart = imagPart0;
   unsigned int count = 0;
   // Loop until escape
   while ( ( count <= maxIters )
         && ((realPart*realPart + imagPart*imagPart) <= 4.0) ) {
      ++count;
      // Update: z = z*z + z0;
      double const oldRealPart = realPart;
      realPart = realPart*realPart - imagPart*imagPart + realPart0;
      imagPart = 2.0*oldRealPart*imagPart + imagPart0;
   }
   return count;
}
```

One GPU thread is required for location in the Mandelbrot Set, with the threads grouped into blocks. The kernel indicates how big a thread-block is, and in the code below we use this to calculate the number of thread-blocks required. This then becomes the `GridSize`.

```
% Load the kernel
cudaFilename = 'pctdemo_processMandelbrotElement.cu';
ptxFilename = ['pctdemo_processMandelbrotElement.',parallel.gpu.ptxext];
kernel = parallel.gpu.CUDAKernel( ptxFilename, cudaFilename );

% Setup
t = tic();
x = gpuArray.linspace( xlim(1), xlim(2), gridSize );
y = gpuArray.linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );

% Make sure we have sufficient blocks to cover all of the locations
numElements = numel( xGrid );
kernel.ThreadBlockSize = [kernel.MaxThreadsPerBlock,1,1];
kernel.GridSize = [ceil(numElements/kernel.MaxThreadsPerBlock),1];

% Call the kernel
count = zeros( size(xGrid), 'gpuArray' );
count = feval( kernel, count, xGrid, yGrid, maxIterations, numElements );

% Show
count = gather( count ); % Fetch the data back from the GPU
gpuCUDAKernelTime = toc( t );
imagesc( x, y, count )
axis off
title( sprintf( '%1.3fsecs (GPU CUDAKernel) = %1.1fx faster', ...
    gpuCUDAKernelTime, cpuTime/gpuCUDAKernelTime ) );
```

0.009secs (GPU CUDAKernel) = 761.9x faster
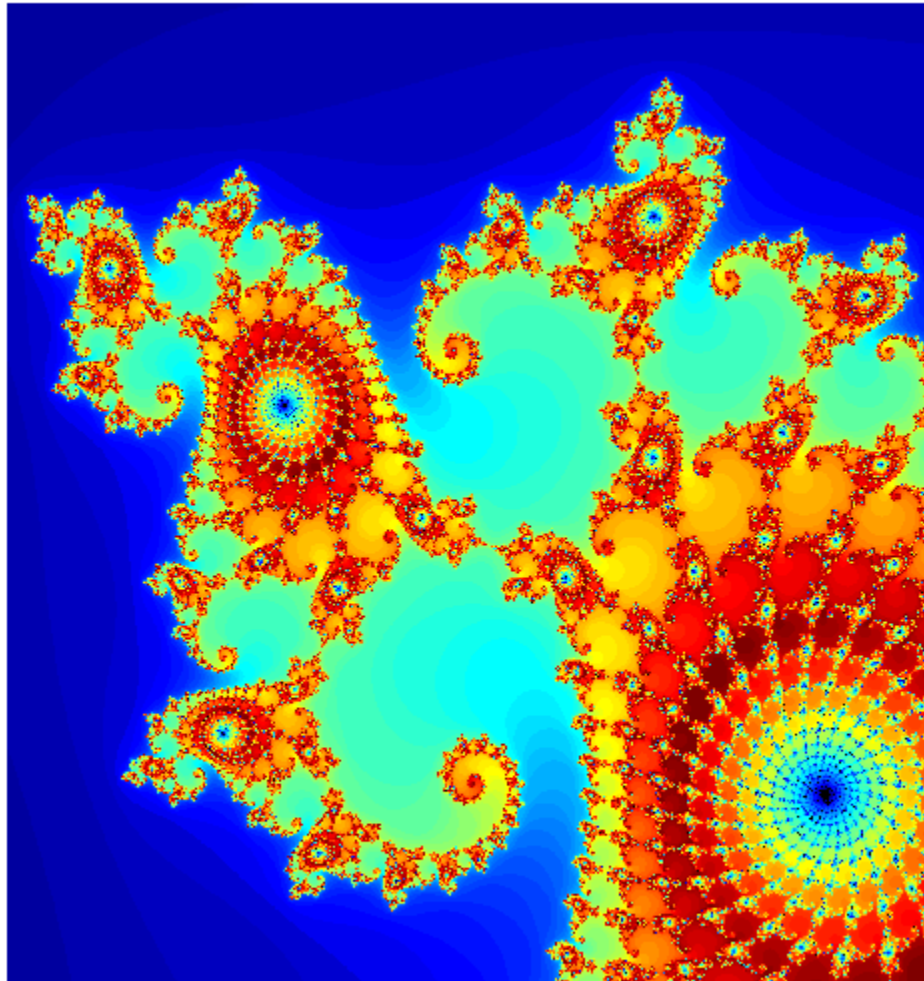
Summary

This example has shown three ways in which a MATLAB algorithm can be adapted to make use of GPU hardware:

1   Convert the input data to be on the GPU using `gpuArray`, leaving the algorithm unchanged
2   Use `arrayfun` on a `gpuArray` input to perform the algorithm on each element of the input independently
3   Use `parallel.gpu.CUDAKernel` to run some existing CUDA/C++ code using MATLAB data

```
title('The Mandelbrot Set on a GPU')
```

The Mandelbrot Set on a GPU
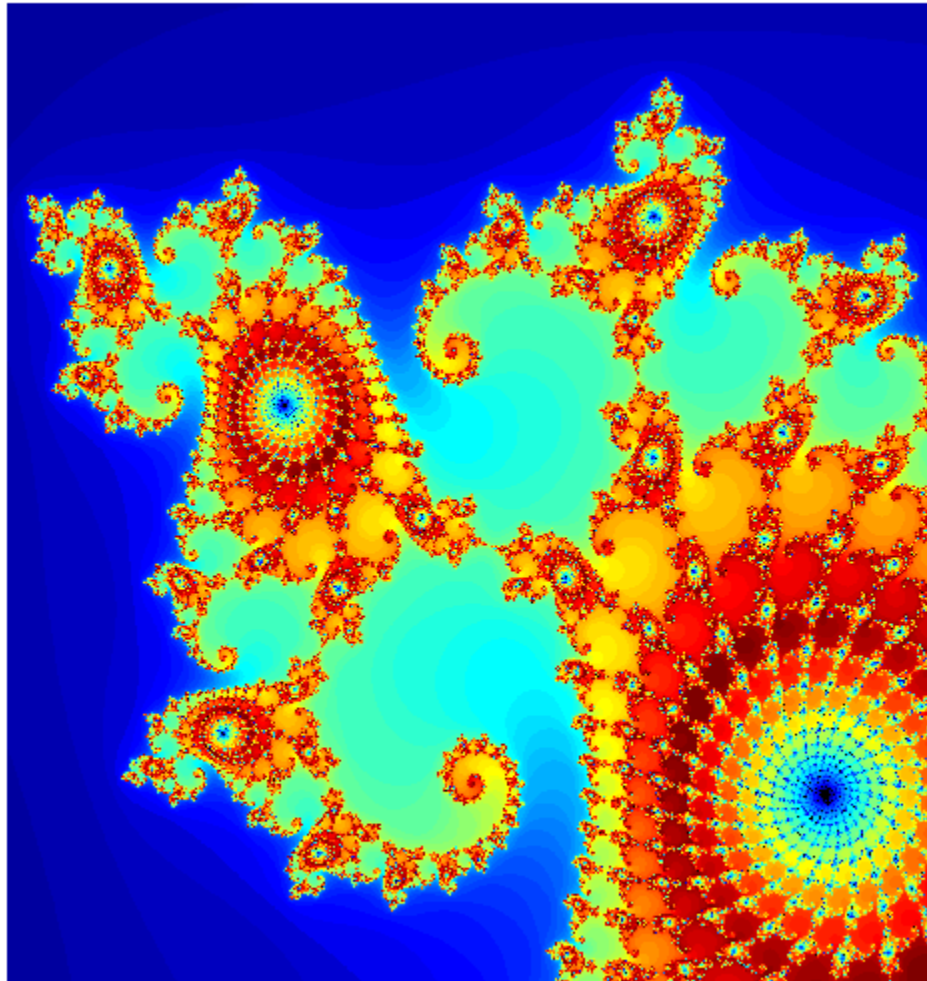
Using GPU ARRAYFUN for Monte-Carlo Simulations

This example shows how prices for financial options can be calculated on a GPU using Monte-Carlo methods. Three simple types of exotic option are used as examples, but more complex options can be priced in a similar way.

This example is a function so that the helpers can be nested inside it.

```
function paralleldemo_gpu_optionpricing
```

This example uses long-running kernels, so cannot run if kernel execution on the GPU can time-out. A time-out is usually only active if the selected GPU is also driving a display.

```
dev = gpuDevice();
if dev.KernelExecutionTimeout
    error( 'pctexample:gpuoptionpricing:KernelTimeout', ...
        ['This example cannot run if kernel execution on the GPU can ', ...
        'time-out.'] );
end
```

Stock Price Evolution

We assume that prices evolve according to a log-normal distribution related to the risk-free interest rate, the dividend yield (if any), and the volatility in the market. All of these quantities are assumed fixed over the lifetime of the option. This gives the following stochastic differential equation for the price:

$$dS = S \times \left[(r - d)dt + \sigma \epsilon \sqrt{dt} \right]$$

where S is the stock price, r is the risk-free interest rate, d is the stock's annual dividend yield, σ is the volatility of the price and ϵ represents a Gaussian white-noise process. Assuming that $(S + \Delta S)/S$ is log-normally distributed, this can be discretized to:

$$S_{t+1} = S_t \times \exp\left[\left(r - d - \frac{1}{2}\sigma^2 \right) \Delta t + \sigma \epsilon \sqrt{\Delta t} \right]$$

As an example let's use $100 of stock that yields a 1% dividend each year. The central government interest rate is assumed to be 0.5%. We examine a two-year time window sampled roughly daily. The market volatility is assumed to be 20% per annum.

```
stockPrice   = 100;   % Stock price starts at $100.
dividend     = 0.01;  % 1% annual dividend yield.
riskFreeRate = 0.005; % 0.5 percent.
timeToExpiry = 2;     % Lifetime of the option in years.
sampleRate   = 1/250; % Assume 250 working days per year.
volatility   = 0.20;  % 20% volatility.
```

We reset the random number generators to ensure repeatable results.

```
seed = 1234;
rng( seed );                % Reset the CPU random number generator.
gpurng( seed );             % Reset the GPU random number generator.
```

We can now loop over time to simulate the path of the stock price:

```
price = stockPrice;
time = 0;
hold on;
while time < timeToExpiry
    time = time + sampleRate;
    drift = (riskFreeRate - dividend - volatility*volatility/2)*sampleRate;
    perturbation = volatility*sqrt( sampleRate )*randn();
    price = price*exp(drift + perturbation);
    plot( time, price, '.' );
end
axis tight;
grid on;
xlabel( 'Time (years)' );
ylabel( 'Stock price ($)' );
```
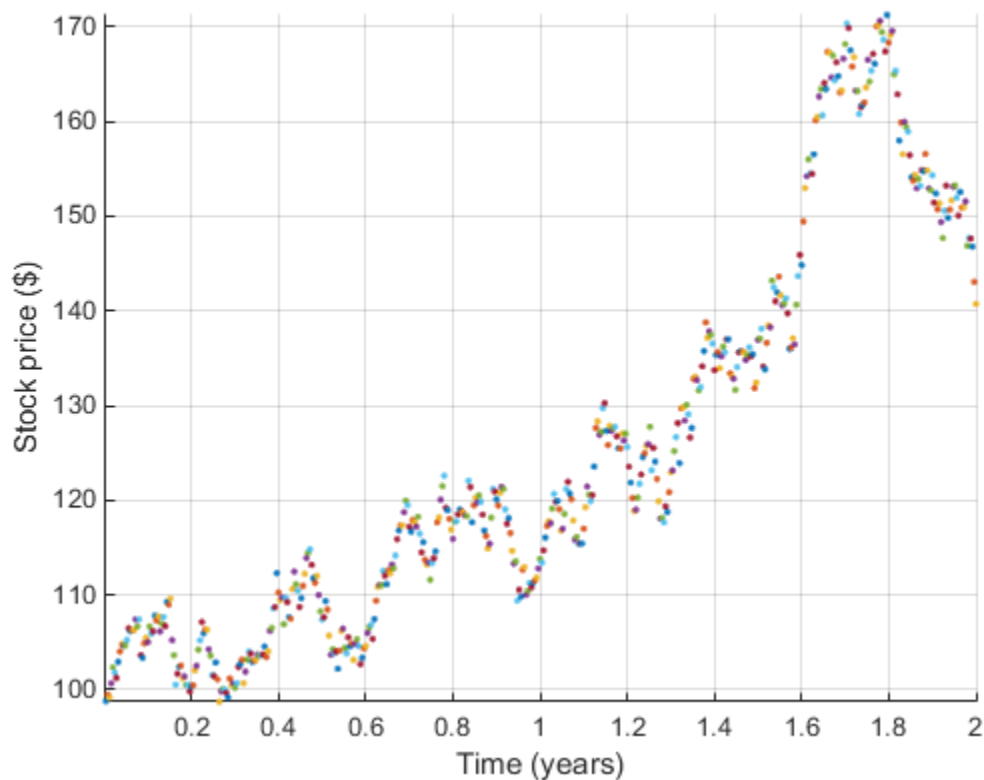

Running on the GPU

To run stock price simulations on the GPU we first need to put the simulation loop inside a helper function:

```
function finalStockPrice = simulateStockPrice(S,r,d,v,T,dT)
    t = 0;
    while t < T
        t = t + dT;
        dr = (r - d - v*v/2)*dT;
        pert = v*sqrt( dT )*randn();
        S = S*exp(dr + pert);
    end
```

```
        finalStockPrice = S;
    end
```

We can then call it thousands of times using `arrayfun`. To ensure the calculations happen on the GPU we make the input prices a GPU vector with one element per simulation. To accurately measure the calculation time on the GPU we use the `gputimeit` function.

```matlab
% Create the input data.
N = 1000000;
startStockPrices = stockPrice*ones(N,1,'gpuArray');

% Run the simulations.
finalStockPrices = arrayfun( @simulateStockPrice, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    timeToExpiry, sampleRate );
meanFinalPrice = mean(finalStockPrices);

% Measure the execution time of the function on the GPU using gputimeit.
% This requires us to store the |arrayfun| call in a function handle.
functionToTime = @() arrayfun(@simulateStockPrice, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    timeToExpiry, sampleRate );
timeTaken = gputimeit(functionToTime);

fprintf( 'Calculated average price of $%1.4f in %1.3f secs.\n', ...
    meanFinalPrice, timeTaken );

clf;
hist( finalStockPrices, 100 );
xlabel( 'Stock price ($)' )
ylabel( 'Frequency' )
grid on;
```

```
Calculated average price of $98.9563 in 0.283 secs.
```
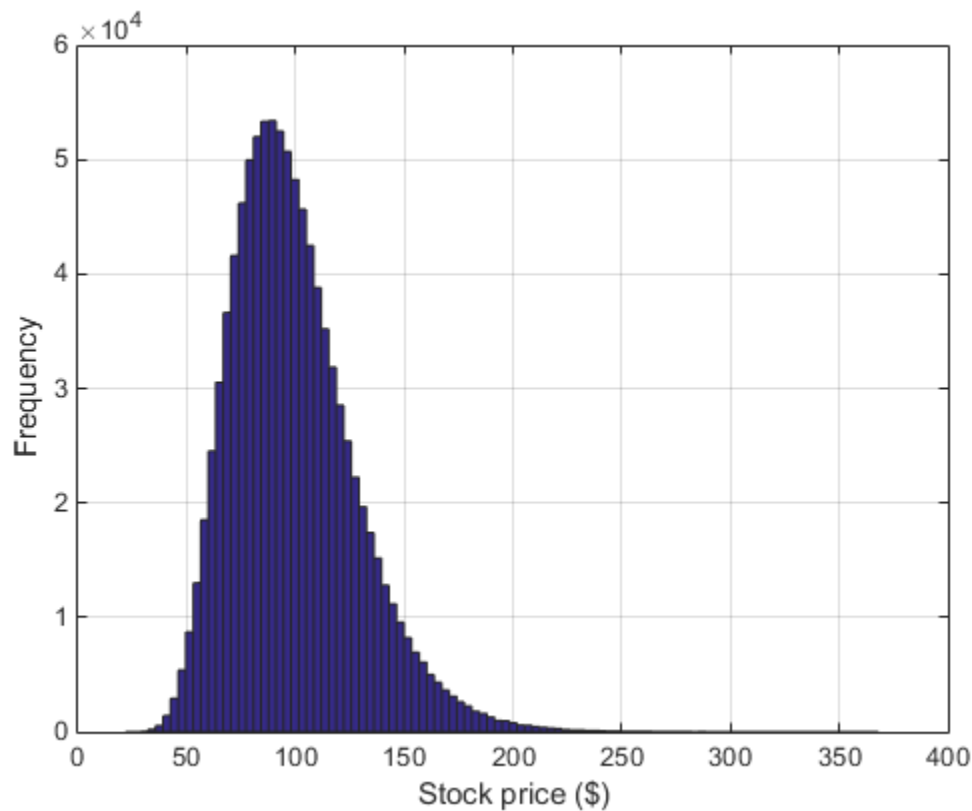
Pricing an Asian Option

As an example, let's use a European Asian Option based on the arithmetic mean of the price of the stock during the lifetime of the option. We can calculate the mean price by accumulating the price during the simulation. For a call option, the option is exercised if the average price is above the strike, and the payout is the difference between the two:

```
function optionPrice = asianCallOption(S,r,d,v,x,T,dT)
    t = 0;
    cumulativePrice = 0;
    while t < T
        t = t + dT;
        dr = (r - d - v*v/2)*dT;
        pert = v*sqrt( dT )*randn();
        S = S*exp(dr + pert);
        cumulativePrice = cumulativePrice + S;
    end
    numSteps = (T/dT);
    meanPrice = cumulativePrice / numSteps;
    % Express the final price in today's money.
    optionPrice = exp(-r*T) * max(0, meanPrice - x);
end
```

Again we use the GPU to run thousands of simulation paths using `arrayfun`. Each simulation path gives an independent estimate of the option price, and we therefore take the mean as our result.

```
strike = 95;    % Strike price for the option ($).
```

```
optionPrices = arrayfun( @asianCallOption, ...
    startStockPrices, riskFreeRate, dividend, volatility, strike, ...
    timeToExpiry, sampleRate );
meanOptionPrice = mean(optionPrices);

% Measure the execution time on the GPU and show the results.
functionToTime = @() arrayfun( @asianCallOption, ...
    startStockPrices, riskFreeRate, dividend, volatility, strike, ...
    timeToExpiry, sampleRate );
timeTaken = gputimeit(functionToTime);

fprintf( 'Calculated average price of $%1.4f in %1.3f secs.\n', ...
    meanOptionPrice, timeTaken );
```

```
Calculated average price of $8.7210 in 0.287 secs.
```

Pricing a Lookback Option

For this example we use a European-style lookback option whose payout is the difference between the minimum stock price and the final stock price over the lifetime of the option.

```
function optionPrice = euroLookbackCallOption(S,r,d,v,T,dT)
    t = 0;
    minPrice = S;
    while t < T
        t = t + dT;
        dr = (r - d - v*v/2)*dT;
        pert = v*sqrt( dT )*randn();
        S = S*exp(dr + pert);
        if S<minPrice
            minPrice = S;
        end
    end
    % Express the final price in today's money.
    optionPrice = exp(-r*T) * max(0, S - minPrice);
end
```

Note that in this case the strike price for the option is the minimum stock price. Because the final stock price is always greater than or equal to the minimum, the option is always exercised and is not really "optional".

```
optionPrices = arrayfun( @euroLookbackCallOption, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    timeToExpiry, sampleRate );
meanOptionPrice = mean(optionPrices);

% Measure the execution time on the GPU and show the results.
functionToTime = @() arrayfun( @euroLookbackCallOption, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    timeToExpiry, sampleRate );
timeTaken = gputimeit(functionToTime);

fprintf( 'Calculated average price of $%1.4f in %1.3f secs.\n', ...
    meanOptionPrice, timeTaken );
```

```
Calculated average price of $19.2711 in 0.286 secs.
```

Pricing a Barrier Option

This final example uses an "up and out" barrier option which becomes invalid if the stock price ever reaches the barrier level. If the stock price stays below the barrier level then the final stock price is used in a normal European call option calculation.

```
function optionPrice = upAndOutCallOption(S,r,d,v,x,b,T,dT)
    t = 0;
    while (t < T) && (S < b)
        t = t + dT;
        dr = (r - d - v*v/2)*dT;
        pert = v*sqrt( dT )*randn();
        S = S*exp(dr + pert);
    end
    if S<b
        % Within barrier, so price as for a European option.
        optionPrice = exp(-r*T) * max(0, S - x);
    else
        % Hit the barrier, so the option is withdrawn.
        optionPrice = 0;
    end
end
```

Note that we must now supply both a strike price for the option and the barrier price at which it becomes invalid:

```
strike  = 95;   % Strike price for the option ($).
barrier = 150;  % Barrier price for the option ($).

optionPrices = arrayfun( @upAndOutCallOption, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    strike, barrier, ...
    timeToExpiry, sampleRate );
meanOptionPrice = mean(optionPrices);

% Measure the execution time on the GPU and show the results.
functionToTime = @() arrayfun( @upAndOutCallOption, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    strike, barrier, ...
    timeToExpiry, sampleRate );
timeTaken = gputimeit(functionToTime);

fprintf( 'Calculated average price of $%1.4f in %1.3f secs.\n', ...
    meanOptionPrice, timeTaken );

Calculated average price of $6.8166 in 0.289 secs.

end
```

Stencil Operations on a GPU

This example uses Conway's "Game of Life" to demonstrate how stencil operations can be performed using a GPU.

Many array operations can be expressed as a "stencil operation", where each element of the output array depends on a small region of the input array. Examples include finite differences, convolution, median filtering, and finite-element methods. This example uses Conway's "Game of Life" to demonstrate two ways to run a stencil operation on a GPU, starting from the code in Cleve Moler's e-book *Experiments in MATLAB*.

The "Game of Life" follows a few simple rules:

- Cells are arranged in a 2D grid
- At each step, the fate of each cell is determined by the vitality of its eight nearest neighbors
- Any cell with exactly three live neighbors comes to life at the next step
- A live cell with exactly two live neighbors remains alive at the next step
- All other cells (including those with more than three neighbors) die at the next step or remain empty

The "stencil" in this case is therefore the 3x3 region around each element. Here are some examples of how a cell is updated:
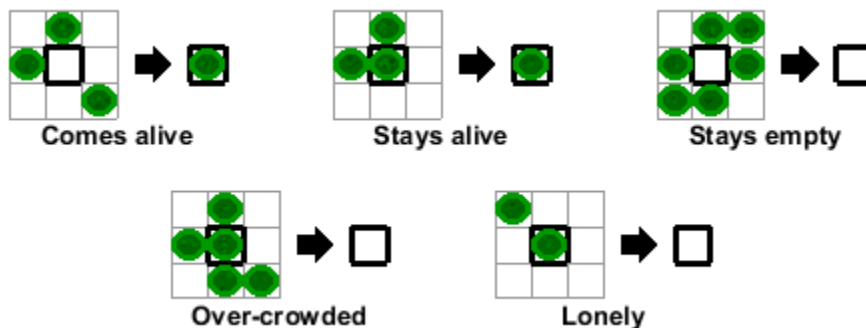
This example is a function to allow the use of nested functions:

```
function paralleldemo_gpu_stencil()
```

Generate a random initial population

An initial population of cells is created on a 2D grid with roughly 25% of the locations alive.

```
gridSize = 500;
numGenerations = 100;
initialGrid = (rand(gridSize,gridSize) > .75);
gpu = gpuDevice();

% Draw the initial grid
hold off
imagesc(initialGrid);
colormap([1 1 1;0 0.5 0]);
title('Initial Grid');
```
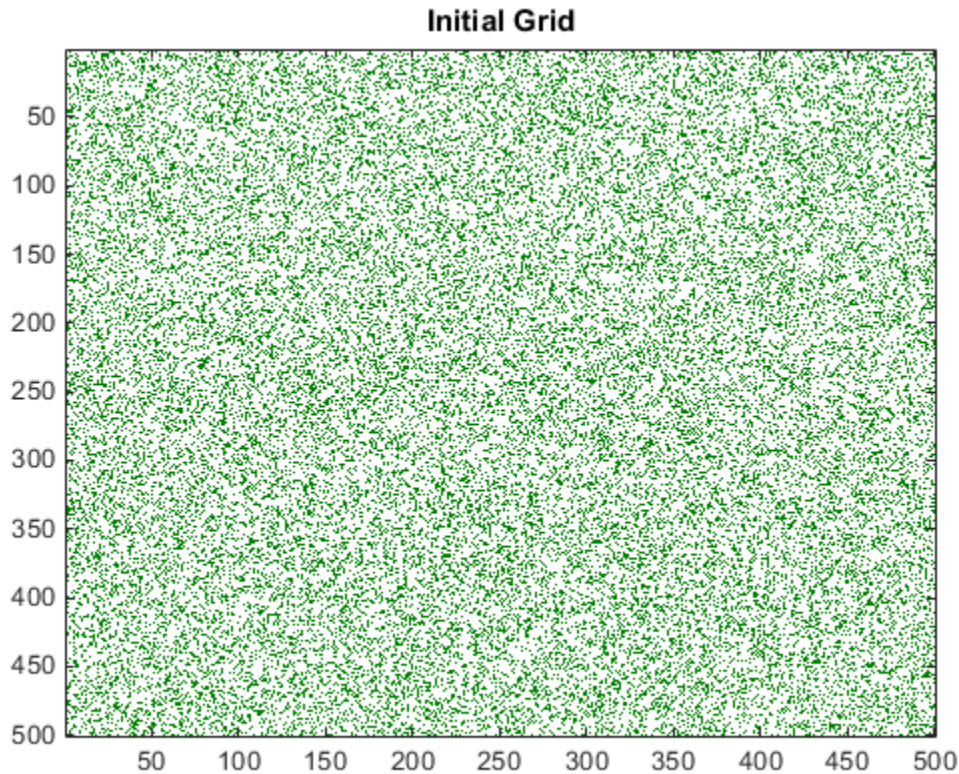
Initial Grid

Playing the Game of Life

The e-book *Experiments in MATLAB* provides an initial implementation that can be used for comparison. This version is fully vectorized, updating all cells in the grid in one pass per generation.

```matlab
function X = updateGrid(X, N)
    p = [1 1:N-1];
    q = [2:N N];
    % Count how many of the eight neighbors are alive.
    neighbors = X(:,p) + X(:,q) + X(p,:) + X(q,:) + ...
        X(p,p) + X(q,q) + X(p,q) + X(q,p);
    % A live cell with two live neighbors, or any cell with
    % three live neighbors, is alive at the next step.
    X = (X & (neighbors == 2)) | (neighbors == 3);
end

grid = initialGrid;
% Loop through each generation updating the grid and displaying it
for generation = 1:numGenerations
    grid = updateGrid(grid, gridSize);

    imagesc(grid);
    title(num2str(generation));
    drawnow;
end
```
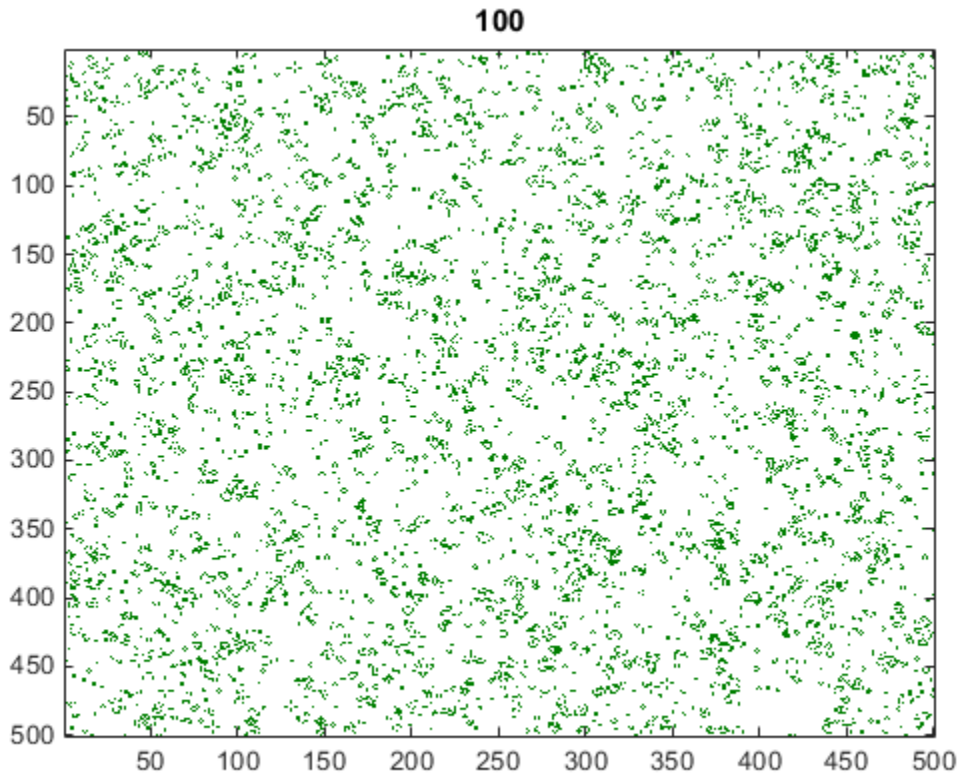
Now re-run the game and measure how long it takes for each generation.

```
grid = initialGrid;
timer = tic();

for generation = 1:numGenerations
    grid = updateGrid(grid, gridSize);
end

cpuTime = toc(timer);
fprintf('Average time on the CPU: %2.3fms per generation.\n', ...
    1000*cpuTime/numGenerations);
```

Average time on the CPU: 11.323ms per generation.

Retain this result to verify the correctness of each version below.

```
expectedResult = grid;
```

Converting the Game of Life to run on a GPU

To run the Game of Life on the GPU, the initial population is sent to the GPU using `gpuArray`. The algorithm remains unchanged. Note that `wait (GPUDevice)` is used to ensure that the GPU has finished calculating before the timer is stopped. This is required only for accurate timing.

```
grid = gpuArray(initialGrid);
timer = tic();
```

```
for generation = 1:numGenerations
    grid = updateGrid(grid, gridSize);
end

wait(gpu); % Only needed to ensure accurate timing
gpuSimpleTime = toc(timer);

% Print out the average computation time and check the result is unchanged.
fprintf(['Average time on the GPU: %2.3fms per generation ', ...
    '(%1.1fx faster).\n'], ...
    1000*gpuSimpleTime/numGenerations, cpuTime/gpuSimpleTime);
assert(isequal(grid, expectedResult));
```

```
Average time on the GPU: 1.655ms per generation (6.8x faster).
```

Creating an element-wise version for the GPU

Looking at the calculations in the `updateGrid` function, it is apparent that the same operations are applied at each grid location independently. This suggests that `arrayfun` could be used to do the evaluation. However, each cell needs to know about its eight neighbors, breaking the element-wise independence. Each element needs to be able to access the full grid while also working independently.

The solution is to use a nested function. Nested functions, even those used with `arrayfun`, can access variables declared in their parent function. This means that each cell can read the whole grid from the previous time-step and index into it.

```
grid = gpuArray(initialGrid);

    function X = updateParentGrid(row, col, N)
        % Take account of boundary effects
        rowU = max(1,row-1);   rowD = min(N,row+1);
        colL = max(1,col-1);   colR = min(N,col+1);
        % Count neighbors
        neighbors ...
            = grid(rowU,colL) + grid(row,colL) + grid(rowD,colL) ...
            + grid(rowU,col)                   + grid(rowD,col) ...
            + grid(rowU,colR) + grid(row,colR) + grid(rowD,colR);
        % A live cell with two live neighbors, or any cell with
        % three live neighbors, is alive at the next step.
        X = (grid(row,col) & (neighbors == 2)) | (neighbors == 3);
    end


timer = tic();

rows = gpuArray.colon(1, gridSize)';
cols = gpuArray.colon(1, gridSize);
for generation = 1:numGenerations
    grid = arrayfun(@updateParentGrid, rows, cols, gridSize);
end

wait(gpu); % Only needed to ensure accurate timing
gpuArrayfunTime = toc(timer);

% Print out the average computation time and check the result is unchanged.
fprintf(['Average time using GPU arrayfun: %2.3fms per generation ', ...
    '(%1.1fx faster).\n'], ...
```

```
    1000*gpuArrayfunTime/numGenerations, cpuTime/gpuArrayfunTime);
assert(isequal(grid, expectedResult));
```

```
Average time using GPU arrayfun: 0.795ms per generation (14.2x faster).
```

Note that we also used another new feature of `arrayfun` here: dimension expansion. We needed to pass only the row and column vectors, and these were automatically expanded into the full grid. The effect is as though we called:

```
[cols,rows] = meshgrid(cols,rows);
```

as part of the `arrayfun` call. This saves us both some computation and some data transfer between CPU memory and GPU memory.

Conclusion

In this example, a simple stencil operation, Conway's "Game of Life", has been implemented on the GPU using `arrayfun` and variables declared in the parent function. This technique can be used to implement a range of stencil operations including finite-element algorithms, convolutions, and filters. It can also be used to access elements in a look-up table defined in the parent function.

```
fprintf('CPU:         %2.3fms per generation.\n', ...
    1000*cpuTime/numGenerations);
fprintf('Simple GPU:  %2.3fms per generation (%1.1fx faster).\n', ...
    1000*gpuSimpleTime/numGenerations, cpuTime/gpuSimpleTime);
fprintf('Arrayfun GPU: %2.3fms per generation (%1.1fx faster).\n', ...
    1000*gpuArrayfunTime/numGenerations, cpuTime/gpuArrayfunTime);
```

```
CPU:          11.323ms per generation.
Simple GPU:   1.655ms per generation (6.8x faster).
Arrayfun GPU: 0.795ms per generation (14.2x faster).
```

```
end
```

Accessing Advanced CUDA Features Using MEX

This example demonstrates how advanced features of the GPU can be accessed using MEX files. It builds on the example "Stencil Operations on a GPU" on page 9-198. The previous example uses Conway's "Game of Life" to demonstrate how stencil operations can be performed using MATLAB® code that runs on a GPU. The present example demonstrates how you can further improve the performance of stencil operations using two advanced features of the GPU: shared memory and texture memory. You do this by writing your own CUDA code in a MEX file and calling the MEX file from MATLAB. You can find an introduction to the use of the GPU in MEX files in "Run MEX-Functions Containing CUDA Code" on page 8-28.

As defined in the previous example, in a "stencil operation", each element of the output array depends on a small region of the input array. Examples include finite differences, convolution, median filtering, and finite-element methods. If we assume that the stencil operation is a key part of our work-flow, we could take the time to convert it to a hand-written CUDA kernel in the hope of getting maximum benefit from the GPU. This example uses Conway's "Game of Life" as our stencil operation, and moves the calculation into a MEX file.

The "Game of Life" follows a few simple rules:

- Cells are arranged in a 2D grid
- At each step, the fate of each cell is determined by the vitality of its eight nearest neighbors
- Any cell with exactly three live neighbors comes to life at the next step
- A live cell with exactly two live neighbors remains alive at the next step
- All other cells (including those with more than three neighbors) die at the next step or remain empty

The "stencil" in this case is therefore the 3x3 region around each element. For more details, view the code for paralleldemo_gpu_stencil.

This example is a function to allow us to use sub-functions:

```
function paralleldemo_gpu_mexstencil()
```

Generate a random initial population

An initial population of cells is created on a 2D grid with approximately 25% of the locations alive.

```
gridSize = 500;
numGenerations = 100;
initialGrid = (rand(gridSize,gridSize) > .75);

hold off
imagesc(initialGrid);
colormap([1 1 1;0 0.5 0]);
title('Initial Grid');
```

Initial Grid
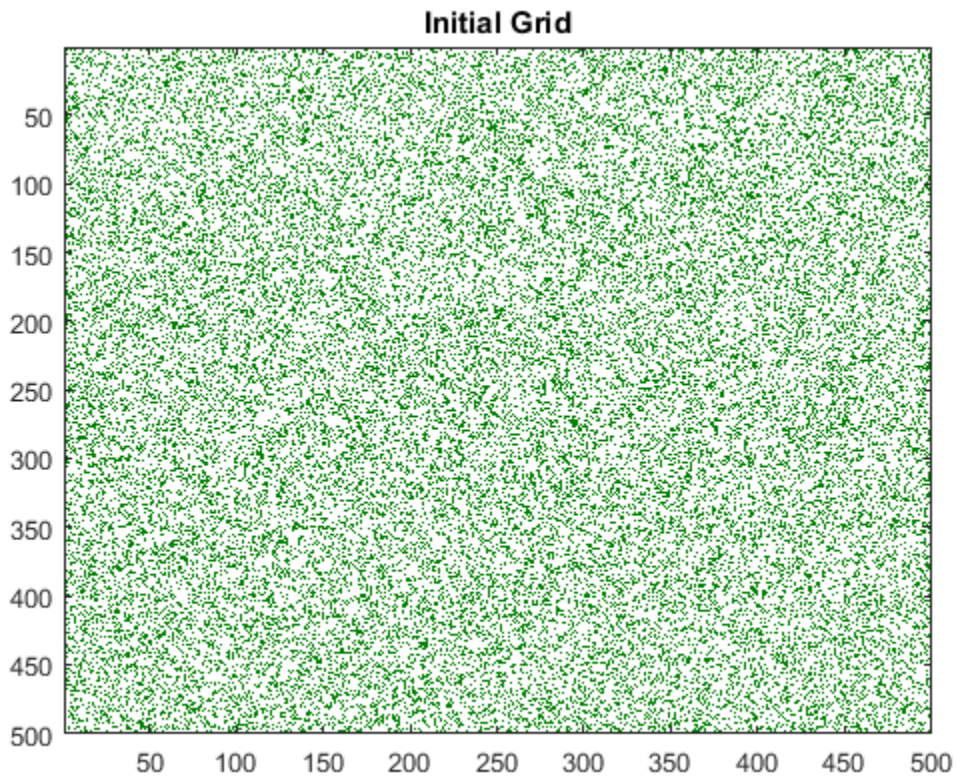
Create a baseline GPU version in MATLAB

To get a performance baseline, we start with the initial implementation described in *Experiments in MATLAB*. This version can run on the GPU by simply making sure the initial population is on the GPU using `gpuArray`.

```
function X = updateGrid(X, N)
    p = [1 1:N-1];
    q = [2:N N];
    % Count how many of the eight neighbors are alive.
    neighbors = X(:,p) + X(:,q) + X(p,:) + X(q,:) + ...
        X(p,p) + X(q,q) + X(p,q) + X(q,p);
    % A live cell with two live neighbors, or any cell with
    % three live neighbors, is alive at the next step.
    X = (X & (neighbors == 2)) | (neighbors == 3);
end

currentGrid = gpuArray(initialGrid);
% Loop through each generation updating the grid and displaying it
for generation = 1:numGenerations
    currentGrid = updateGrid(currentGrid, gridSize);

    imagesc(currentGrid);
    title(num2str(generation));
    drawnow;
end
```
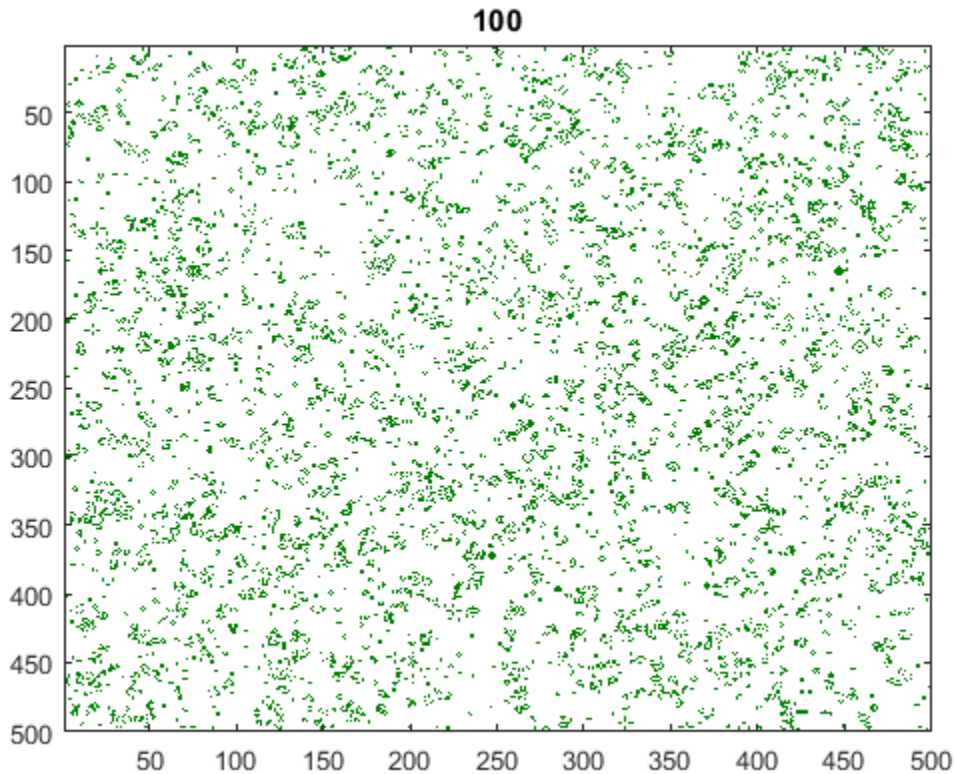
Now re-run the game and measure how long it takes for each generation.

```matlab
% This function defines the outer loop that calls each generation, without
% doing the display.
function grid=callUpdateGrid(grid, gridSize, N)
    for gen = 1:N
        grid = updateGrid(grid, gridSize);
    end
end

gpuInitialGrid = gpuArray(initialGrid);

% Retain this result to verify the correctness of each version below.
expectedResult = callUpdateGrid(gpuInitialGrid, gridSize, numGenerations);

gpuBuiltinsTime = gputimeit(@() callUpdateGrid(gpuInitialGrid, ...
                                        gridSize, numGenerations));

fprintf('Average time on the GPU: %2.3fms per generation \n', ...
        1000*gpuBuiltinsTime/numGenerations);
```

```
Average time on the GPU: 1.528ms per generation
```

Create a MEX version that uses shared memory

When writing a CUDA kernel version of the stencil operation, we have to split the input data into blocks on which each thread block can operate. Each thread in the block will be reading data that is also needed by other threads in the block. One way to minimize the number of read operations is to

9-205

copy the required input data into shared memory before processing. This copy must include some neighboring elements to allow correct calculation of the block edges. For the Game of Life, where our stencil is just a 3x3 square of elements, we need a one element boundary. For example, for a 9x9 grid processed using 3x3 blocks, the fifth block would operate on the highlighted region, where the yellow elements are the "halo" it must also read.
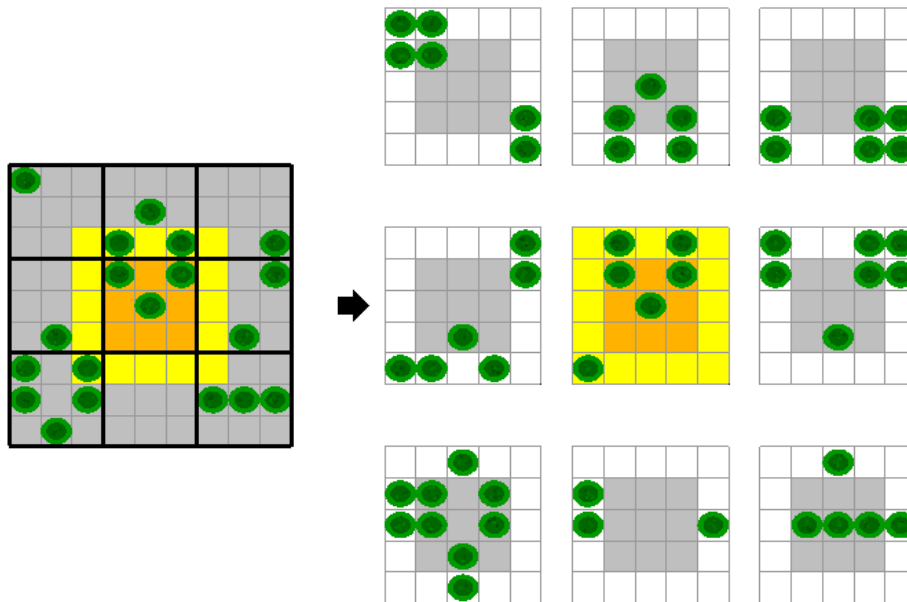
We have put the CUDA code that illustrates this approach into the file pctdemo_life_cuda_shmem.cu. The CUDA device function in this file operates as follows:

1 All threads copy the relevant part of the input grid into shared memory, including the halo.

2 The threads synchronize with one another to ensure shared memory is ready.

3 Threads that fit in the output grid perform the Game of Life calculation.

The host code in this file invokes the CUDA device function once for each generation, using the CUDA runtime API. It uses two different writable buffers for the input and output. At every iteration, the MEX-file swaps the input and output pointers so that no copying is required.

In order to call the function from MATLAB, we need a MEX gateway that unwraps the input arrays from MATLAB, builds a workspace on the GPU, and returns the output. The MEX gateway function can be found in the file pctdemo_life_mex_shmem.cpp.

Before we can call the MEX-file, we must compile it using mexcuda, which requires installing the nvcc compiler. You can compile these two files into a single MEX function using a command like

```
mexcuda -output pctdemo_life_mex_shmem ...
        pctdemo_life_cuda_shmem.cu pctdemo_life_mex_shmem.cpp
```

which will produce a MEX file named pctdemo_life_mex_shmem.

```
% Calculate the output value using the MEX file with shared memory. The
% initial input value is copied to the GPU inside the MEX file.
```

```
grid = pctdemo_life_mex_shmem(initialGrid, numGenerations);
gpuMexTime = gputimeit(@()pctdemo_life_mex_shmem(initialGrid, ...
                                                 numGenerations));
% Print out the average computation time and check the result is unchanged.
fprintf('Average time of %2.3fms per generation (%1.1fx faster).\n', ...
        1000*gpuMexTime/numGenerations, gpuBuiltinsTime/gpuMexTime);
assert(isequal(grid, expectedResult));
```

```
Average time of 0.055ms per generation (27.7x faster).
```

Create a MEX version that uses texture memory

A second way to deal with the issue of repeated read operations is to use the GPU's texture memory. Texture accesses are cached in a way that attempts to provide good performance when several threads access overlapping 2-D data. This is exactly the access pattern that we have in a stencil operation.

There are two CUDA APIs that can be used to read texture memory. We choose to use the CUDA texture reference API, which is supported on all CUDA devices. The maximum number of elements that can be in an array bound to a texture is 2^{27}, so the texture approach will not work if the input has more elements.

The CUDA code that illustrates this approach is in `pctdemo_life_cuda_texture.cu`. As in the previous version, this file contains both host code and device code. Three features of this file enable the use of texture memory in the device function.

1   The texture variable is declared at the top of the MEX-file.
2   The CUDA device function fetches the input from the texture reference.
3   The MEX-file binds the texture reference to the input buffer.

In this file, the CUDA device function is simpler than before. It only needs to perform the Game of Life calculations. There is no need to copy into shared memory or to synchronize the threads.

As in the shared-memory version, the host code invokes the CUDA device function once for each generation, using the CUDA runtime API. It again uses two writable buffers for the input and output and swaps their pointers at every iteration. Before each call to the device function, it binds the texture reference to the appropriate buffer. After the device function has executed, it unbinds the texture reference.

There is a MEX gateway file for this version, `pctdemo_life_mex_texture.cpp` that takes care of the input and output arrays and of workspace allocation. These files can be built into a single MEX file using a command like the following.

```
mexcuda -output pctdemo_life_mex_texture ...
        pctdemo_life_cuda_texture.cu pctdemo_life_mex_texture.cpp
```

```
% Calculate the output value using the MEX file with textures.
grid = pctdemo_life_mex_texture(initialGrid, numGenerations);
gpuTexMexTime = gputimeit(@()pctdemo_life_mex_texture(initialGrid, ...
                                                      numGenerations));
% Print out the average computation time and check the result is unchanged.
fprintf('Average time of %2.3fms per generation (%1.1fx faster).\n', ...
        1000*gpuTexMexTime/numGenerations, gpuBuiltinsTime/gpuTexMexTime);
assert(isequal(grid, expectedResult));
```

```
Average time of 0.025ms per generation (61.5x faster).
```

9-207

Conclusions

In this example, we have illustrated two different ways to deal with copying inputs for stencil operations: explicitly copy blocks into shared memory, or take advantage of the GPU's texture cache. The best approach will depend on the size of the stencil, the size of the overlap region, and the hardware generation of your GPU. The important point is that you can use each of these approaches in conjunction with your MATLAB code to optimize your application.

```
fprintf('First version using gpuArray:  %2.3fms per generation.\n', ...
        1000*gpuBuiltinsTime/numGenerations);
fprintf(['MEX with shared memory: %2.3fms per generation ',...
         '(%1.1fx faster).\n'], 1000*gpuMexTime/numGenerations, ...
        gpuBuiltinsTime/gpuMexTime);
fprintf(['MEX with texture memory: %2.3fms per generation '...
         '(%1.1fx faster).\n']', 1000*gpuTexMexTime/numGenerations, ...
        gpuBuiltinsTime/gpuTexMexTime);
```

```
First version using gpuArray:  1.528ms per generation.
MEX with shared memory: 0.055ms per generation (27.7x faster).
MEX with texture memory: 0.025ms per generation (61.5x faster).
```

```
end
```

Improve Performance of Small Matrix Problems on the GPU using PAGEFUN

This example shows how to use `pagefun` to improve the performance of applying a large number of independent rotations and translations to objects in a 3-D environment. This is typical of a range of problems which involve a large batch of calculations on small arrays.

GPUs are most effective when carrying out calculations on very large matrices. In MATLAB® this is usually achieved by vectorizing code to maximize the work done in each instruction. When you have a large data set but your calculations are divided into many small matrix operations, it can be challenging to maximize performance by running simultaneously on the many hundreds of GPU cores.

The `arrayfun` and `bsxfun` functions allow scalar operations to be carried out in parallel on the GPU. The `pagefun` function adds the capability of carrying out matrix operations in batch in a similar way. The `pagefun` function is available in Parallel Computing Toolbox™ for use with gpuArrays.

In this example, a robot is navigating a known map containing a large number of features that the robot can identify using its sensors. The robot locates itself in the map by measuring the relative position and orientation of those objects and comparing them to the map locations. Assuming the robot is not completely lost, it can use any difference between the two to correct its position, for instance by using a Kalman Filter. We will focus on the first part of the algorithm.

This example is a function so that the helper functions can be nested inside it.

```
function paralleldemo_gpu_pagefun
```

Set up the map

Let's create a map of objects with randomized positions and orientations in a large room.

```
numObjects = 1000;
roomDimensions = [50 50 5]; % Length * breadth * height in meters
```

We represent positions and orientations using 3-by-1 vectors T and 3-by-3 rotation matrices R. When we have N of these *transforms* we pack the translations into a 3-by-N matrix, and the rotations into a 3-by-3-by-N array. The following function initializes N transforms with random values, providing a structure as output:

```
    function Tform = randomTransforms(N)
        Tform.T = zeros(3, N);
        Tform.R = zeros(3, 3, N);
        for i = 1:N
            Tform.T(:,i) = rand(3, 1) .* roomDimensions';
            % To get a random orientation, we can extract an orthonormal
            % basis for a random 3-by-3 matrix.
            Tform.R(:,:,i) = orth(rand(3, 3));
        end
    end
```

Now use this to set up the map of object transforms, and a start location for the robot.

```
Map = randomTransforms(numObjects);
Robot = randomTransforms(1);
```

Define the equations

To correctly identify map features the robot needs to transform the map to put its sensors at the origin. Then it can find map objects by comparing what it sees with what it expects to see.

For a map object i we can find its position relative to the robot $T_{rel}(i)$ and orientation $\mathbf{R}_{rel}(i)$ by transforming its global map location:

$$
\begin{aligned}
\mathbf{R}_{rel}(i) &= \mathbf{R}_{bot}^\top \mathbf{R}_{map}(i) \\
T_{rel}(i) &= \mathbf{R}_{bot}^\top (T_{map}(i) - T_{bot})
\end{aligned}
$$

where T_{bot} and $\mathbf{R}_{bot}$ are the position and orientation of the robot, and $T_{map}(i)$ and $\mathbf{R}_{map}(i)$ represent the map data. The equivalent MATLAB code looks like this:

```
Rrel(:,:,i) = Rbot' * Rmap(:,:,i)
Trel(:,i) = Rbot' * (Tmap(:,i) - Tbot)
```

Perform many matrix transforms on the CPU using a `for` loop

We need to transform every map object to its location relative to the robot. We can do this serially by looping over all the transforms in turn. Note the 'like' syntax for `zeros` which will allow us to use the same code on the GPU in the next section.

```
function Rel = loopingTransform(Robot, Map)
    Rel.R = zeros(size(Map.R), 'like', Map.R); % Initialize memory
    Rel.T = zeros(size(Map.T), 'like', Map.T); % Initialize memory
    for i = 1:numObjects
        Rel.R(:,:,i) = Robot.R' * Map.R(:,:,i);
        Rel.T(:,i) = Robot.R' * (Map.T(:,i) - Robot.T);
    end
end
```

To time the calculation we use the `timeit` function, which will call `loopingTransform` multiple times to get an average timing. Since it requires a function with no arguments, we use the `@()` syntax to create an anonymous function of the right form.

```
cpuTime = timeit(@()loopingTransform(Robot, Map));
fprintf('It takes %3.4f seconds on the CPU to execute %d transforms.\n', ...
        cpuTime, numObjects);
```

```
It takes 0.0104 seconds on the CPU to execute 1000 transforms.
```

Try the same code on the GPU

To run this code on the GPU is merely a matter of copying the data into a `gpuArray`. When MATLAB encounters data stored on the GPU it will run any code using it on the GPU as long as it is supported.

```
gMap.R = gpuArray(Map.R);
gMap.T = gpuArray(Map.T);
gRobot.R = gpuArray(Robot.R);
gRobot.T = gpuArray(Robot.T);
```

Now we call `gputimeit`, which is the equivalent of `timeit` for code that includes GPU computation. It makes sure all GPU operations have finished before recording the time.

```
fprintf('Computing...\n');
gpuTime = gputimeit(@()loopingTransform(gRobot, gMap));
```

```
fprintf('It takes %3.4f seconds on the GPU to execute %d transforms.\n', ...
        gpuTime, numObjects);
fprintf(['Unvectorized GPU code is %3.2f times slower ',...
    'than the CPU version.\n'], gpuTime/cpuTime);
```

```
Computing...
It takes 0.5588 seconds on the GPU to execute 1000 transforms.
Unvectorized GPU code is 53.90 times slower than the CPU version.
```

Batch process using pagefun

The GPU version above was very slow because, although all calculations were independent, they ran in series. Using `pagefun` we can run all the computations in parallel. We also employ `bsxfun` to calculate the translations, since these are element-wise operations.

```
function Rel = pagefunTransform(Robot, Map)
    Rel.R = pagefun(@mtimes, Robot.R', Map.R);
    Rel.T = Robot.R' * bsxfun(@minus, Map.T, Robot.T);
end
```

```
gpuPagefunTime = gputimeit(@()pagefunTransform(gRobot, gMap));
fprintf(['It takes %3.4f seconds on the GPU using pagefun ',...
    'to execute %d transforms.\n'], gpuPagefunTime, numObjects);
fprintf(['Vectorized GPU code is %3.2f times faster ',...
    'than the CPU version.\n'], cpuTime/gpuPagefunTime);
fprintf(['Vectorized GPU code is %3.2f times faster ',...
    'than the unvectorized GPU version.\n'], gpuTime/gpuPagefunTime);
```

```
It takes 0.0008 seconds on the GPU using pagefun to execute 1000 transforms.
Vectorized GPU code is 13.55 times faster than the CPU version.
Vectorized GPU code is 730.18 times faster than the unvectorized GPU version.
```

Explanation

The first computation was the calculation of the rotations. This involved a matrix multiply, which translates to the function `mtimes` (*). We pass this to `pagefun` along with the two sets of rotations to be multiplied:

```
Rel.R = pagefun(@mtimes, Robot.R', Map.R);
```

`Robot.R'` is a 3-by-3 matrix, and `Map.R` is a 3-by-3-by-N array. The `pagefun` function matches each independent matrix from the map to the same robot rotation, and gives us the required 3-by-3-by-N output.

The translation calculation also involves a matrix multiply, but the normal rules of matrix multiplication allow this to come outside the loop without any changes. However, it also involves subtracting `Robot.T` from `Map.T`, which are different sizes. Since this is an element-by-element operation, we can use `bsxfun` to match up dimensions in the same way as `pagefun` did for the rotations:

```
Rel.T = Robot.R' * bsxfun(@minus, Map.T, Robot.T);
```

This time we needed to use the element-wise operator which maps to the function `minus` (-).

More advanced GPU vectorization - Solving a "lost robot" problem

If our robot was in an unknown part of the map, it might use a global search algorithm to locate itself. The algorithm would test a number of possible locations by carrying out the above computation and

looking for good correspondence between the objects seen by the robot's sensors and what it would expect to see at that position.

Now we have got multiple robots as well as multiple objects. N objects and M robots should give us N*M transforms out. To distinguish 'robot space' from 'object space' we use the 4th dimension for rotations and the 3rd for translations. That means our robot rotations will be 3-by-3-by-1-by-M, and the translations will be 3-by-1-by-M.

We initialize our search with random robot locations. A good search algorithm would use topological or other clues to seed the search more intelligently.

```
numRobots = 10;
Robot = randomTransforms(numRobots);
Robot.R = reshape(Robot.R, 3, 3, 1, []); % Spread along the 4th dimension
Robot.T = reshape(Robot.T, 3, 1, []); % Spread along the 3rd dimension
gRobot.R = gpuArray(Robot.R);
gRobot.T = gpuArray(Robot.T);
```

Our new looping transform function requires two nested loops, to loop over the robots as well as over the objects.

```
    function Rel = loopingTransform2(Robot, Map)
        Rel.R = zeros(3, 3, numObjects, numRobots, 'like', Map.R);
        Rel.T = zeros(3, numObjects, numRobots, 'like', Map.T);
        for i = 1:numObjects
            for j = 1:numRobots
                Rel.R(:,:,i,j) = Robot.R(:,:,1,j)' * Map.R(:,:,i);
                Rel.T(:,i,j) = ...
                    Robot.R(:,:,1,j)' * (Map.T(:,i) - Robot.T(:,1,j));
            end
        end
    end
```

```
cpuTime = timeit(@()loopingTransform2(Robot, Map));
fprintf('It takes %3.4f seconds on the CPU to execute %d transforms.\n', ...
        cpuTime, numObjects*numRobots);
```

```
It takes 0.1493 seconds on the CPU to execute 10000 transforms.
```

For our GPU timings we use `tic` and `toc` this time, because otherwise the calculation would take too long. This will be precise enough for our purposes. To ensure any cost associated with creating the output data is included, we are calling `loopingTransform2` with a single output variable, just as `timeit` and `gputimeit` do by default.

```
fprintf('Computing...\n');
tic;
gRel = loopingTransform2(gRobot, gMap); %#ok<NASGU> Suppress unused variable warning
gpuTime = toc;

fprintf('It takes %3.4f seconds on the GPU to execute %d transforms.\n', ...
        gpuTime, numObjects*numRobots);
fprintf(['Unvectorized GPU code is %3.2f times slower ',...
    'than the CPU version.\n'], gpuTime/cpuTime);
```

```
Computing...
It takes 7.0564 seconds on the GPU to execute 10000 transforms.
Unvectorized GPU code is 47.26 times slower than the CPU version.
```

As before, the looping version runs much slower on the GPU because it is not doing calculations in parallel.

The new `pagefun` version needs to incorporate the `transpose` operator as well as `mtimes` into a call to `pagefun`. We also need to `squeeze` the transposed robot orientations to put the spread over robots into the 3rd dimension, to match the translations. Despite this, the resulting code is considerably more compact.

```
function Rel = pagefunTransform2(Robot, Map)
    Rt = pagefun(@transpose, Robot.R);
    Rel.R = pagefun(@mtimes, Rt, Map.R);
    Rel.T = pagefun(@mtimes, squeeze(Rt), ...
                    bsxfun(@minus, Map.T, Robot.T));
end
```

Once again, `pagefun` and `bsxfun` expand dimensions appropriately. So where we multiply 3-by-3-by-1-by-M matrix `Rt` with 3-by-3-by-N-by-1 matrix `Map.R`, we get a 3-by-3-by-N-by-M matrix out.

```
gpuPagefunTime = gputimeit(@()pagefunTransform2(gRobot, gMap));
fprintf(['It takes %3.4f seconds on the GPU using pagefun ',...
    'to execute %d transforms.\n'], gpuPagefunTime, numObjects*numRobots);
fprintf(['Vectorized GPU code is %3.2f times faster ',...
    'than the CPU version.\n'], cpuTime/gpuPagefunTime);
fprintf(['Vectorized GPU code is %3.2f times faster ',...
    'than the unvectorized GPU version.\n'], gpuTime/gpuPagefunTime);
```

```
It takes 0.0025 seconds on the GPU using pagefun to execute 10000 transforms.
Vectorized GPU code is 59.97 times faster than the CPU version.
Vectorized GPU code is 2834.45 times faster than the unvectorized GPU version.
```

Conclusion

The `pagefun` function supports a number of 2-D operations, as well as most of the scalar operations supported by `arrayfun` and `bsxfun`. Together, these functions allow you to vectorize a range of computations involving matrix algebra and array manipulation, eliminating the need for loops and making huge performance gains.

Wherever you are doing small calculations on GPU data in a loop, you should consider converting to a batch implementation in this way. This can also be an opportunity to make use of the GPU to improve performance where previously it gave no performance gains.

```
end
```

Profiling Explicit Parallel Communication

This example shows how to profile explicit communication to the nearest neighbor lab. It illustrates the use of `labSend`, `labReceive`, and `labSendReceive`, showing both the slow (incorrect) and the fast (optimal) way of implementing this algorithm. The problem is explored using the parallel profiler. For getting started with parallel profiling, see "Profiling Parallel Code" on page 5-32.

The figures in this example are produced from a 12-node cluster.

The example code involves explicit communication. In MATLAB® explicit communication is synonymous with directly using Parallel Computing Toolbox communication primitives (e.g. `labSend`, `labReceive`, `labSendReceive`, `labBarrier`). Performance problems involving this type of communication, if not related to the underlying hardware, can be difficult to trace. With the parallel profiler many of these problems can be interactively identified. It is important to remember you can separate the various parts of your program into separate functions. This can help when profiling, because some data is collected only for each function.

The Algorithm

The algorithm we are profiling is a nearest neighbor communication pattern. Each MATLAB worker needs data only from itself and one neighboring lab. This type of data parallel pattern lends itself well to many matrix problems, but when done incorrectly, can be needlessly slow. In other words, each lab depends on data that is *already* available on an adjacent lab. For example, on a four-lab cluster, lab 1 wants to send some data to lab 2 and needs some data from lab 4 so each lab depends on only one other lab:

1 depends on -> 4

2 depends on -> 1

3 depends on -> 2

4 depends on -> 3

It is possible to implement any given communication algorithm using `labSend` and `labReceive`. `labReceive` always blocks your program until the communication is complete, while `labSend` might not if the data is small. Using `labSend` first, though, doesn't help in most cases.

One way to accomplish this algorithm is to have every lab wait for a receive, and only one lab start the communication chain by completing a send and then a receive. Alternatively, we can use `labSendReceive`, and at first glance it may not be apparent that there should be a major difference in performance.

You can view the code for pctdemo_aux_profbadcomm and pctdemo_aux_profcomm to see the complete implementations of this algorithm. Look at the first file and notice that it uses `labSend` and `labReceive` for communication.

It is a common mistake to start thinking in terms of `labSend` and `labReceive` when it is not necessary. Looking at how this `pctdemo_aux_profbadcomm` implementation performs will give us a better idea of what to expect.

Profiling the `labSend` Implementation

```
spmd
    labBarrier; % to ensure the labs all start at the same time
```

```
    mpiprofile reset;
    mpiprofile on;
    pctdemo_aux_profbadcomm;
end

Lab  1:
  sending to 2
Lab  2:
  receive from 1
Lab  3:
  receive from 2
Lab  4:
  receive from 3
Lab  5:
  receive from 4
Lab  6:
  receive from 5
Lab  7:
  receive from 6
Lab  8:
  receive from 7
Lab  9:
  receive from 8
Lab 10:
  receive from 9
Lab 11:
  receive from 10
Lab 12:
  receive from 11
Lab  1:
  receive from 12
Lab  6:
  sending to 7
Lab  7:
  sending to 8
Lab  8:
  sending to 9
Lab  9:
  sending to 10
Lab 10:
  sending to 11
Lab 11:
  sending to 12
Lab 12:
  sending to 1
Lab  2:
  sending to 3
Lab  3:
  sending to 4
Lab  4:
  sending to 5
Lab  5:
  sending to 6

mpiprofile viewer
```

The **Function Summary Report** is displayed. On this page, you can see time spent waiting in communications as an orange bar under the **Total Time Plot** heading. The data below shows that

considerable amount of time was spent waiting. Let's see how the parallel profiler helps to identify the causes of these waits.

| pctdemo_aux_profbadcomm>iRecFromPrevLab | 1 | 0.142 s | 0.142 s | 0.141 s | 0.131 s | 18.00 Mb | 0.7% | |
| pctdemo_aux_profbadcomm>iRecFromPrevLab | 1 | 0.018 s | 0.018 s | 0.014 s | 0.000 s | 18.00 Mb | 21.6% | |

Quickstart Steps

1 In the profiler Function Summary Report, look at the pctdemo_aux_profbadcomm entry and click **Compare max vs. min TotalTime**. Observe the large orange waiting time indicated under the function `iRecFromPrevLab`. This is an early indication that there is something wrong with a corresponding send, either because of network problems or algorithm problems.

2 Use the top toolbar table to click `Plot All Per Worker Communication`. The first figure in this view shows all the data received by each lab. In this example each lab is receiving the same amount of data from the previous lab, so it doesn't seem to be a data distribution problem. The second figure shows the various communication times including the time spent waiting for communication. In the third figure, the **Comm Waiting Time Per Worker** plot shows a stepwise increase in waiting time. An example **Comm Waiting Time Per Worker** plot can be seen below using a 12-node cluster. It is good to go back and check what is happening on the source lab.

3 Browse what's happening on lab 1. a.) In the Profiler click **Home**. b.) Click the top-level `pctdemo_aux_profbadcomm` function to go to the Function Detail Report. c.) Be sure that **Show function listing** is selected. d.) Scroll down and see where lab 1 spends time and which lines are covered. e.) For comparison, look at the **Lines where the most time was spent** table and select the last lab usinAAg the **Go to worker** listbox.

To see all the profiled lines of code, scroll down to the last item in the page. An example of this annotated code listing can be seen below.

```
Color highlight code according to:  comm waiting time  ⌄

  time    calls    (sent/rec/waiting)   line
                                        1  function pctdemo_aux_profbadcomm
                                        2  %PCTDEMO_AUX_PROFBADCOMM Demonstrate poor communication patterns.
                                        3  %     This is a sample parallel program where the communication pattern causes
                                        4  %     the program actually runs in serial. You can see the problem using the
                                        5  %     parallel profiler.
                                        6
                                        7  %     Copyright 2007 The MathWorks, Inc.
                                        8
              1                         9  N = iGetComplexityByNumLabs();
  0.03        1                        10  mydata = rand(N);
                                       11
              1                        12  if labindex == 1
  0.03        1         18m/0b/0s      13      iSendDataToNextLab(mydata);
              1                        14  end
  0.14        1         0b/18m/0.131s  15  otherLabData = iRecFromPrevLab(); %#ok Don't need return data.
                                       16  % use the data received
                                       17  % e.g. myresult = otherLabData*mydata;
              1                        18  if labindex~=1
                                       19      iSendDataToNextLab(mydata);
< 0.01        1                        20  end
```

Communication Plots Using a Larger Non-local Cluster

To clearly see the problem with our usage of `labSend` and `labReceive`, look at the following **Receive Comm Waiting Time** plot from a 12-node cluster.

No Plot
Plot Time Histograms
Plot All Per Worker Communication
Plot Communication Time Per Worker

All workers Comm Waiting Time Per Worker
for all functions
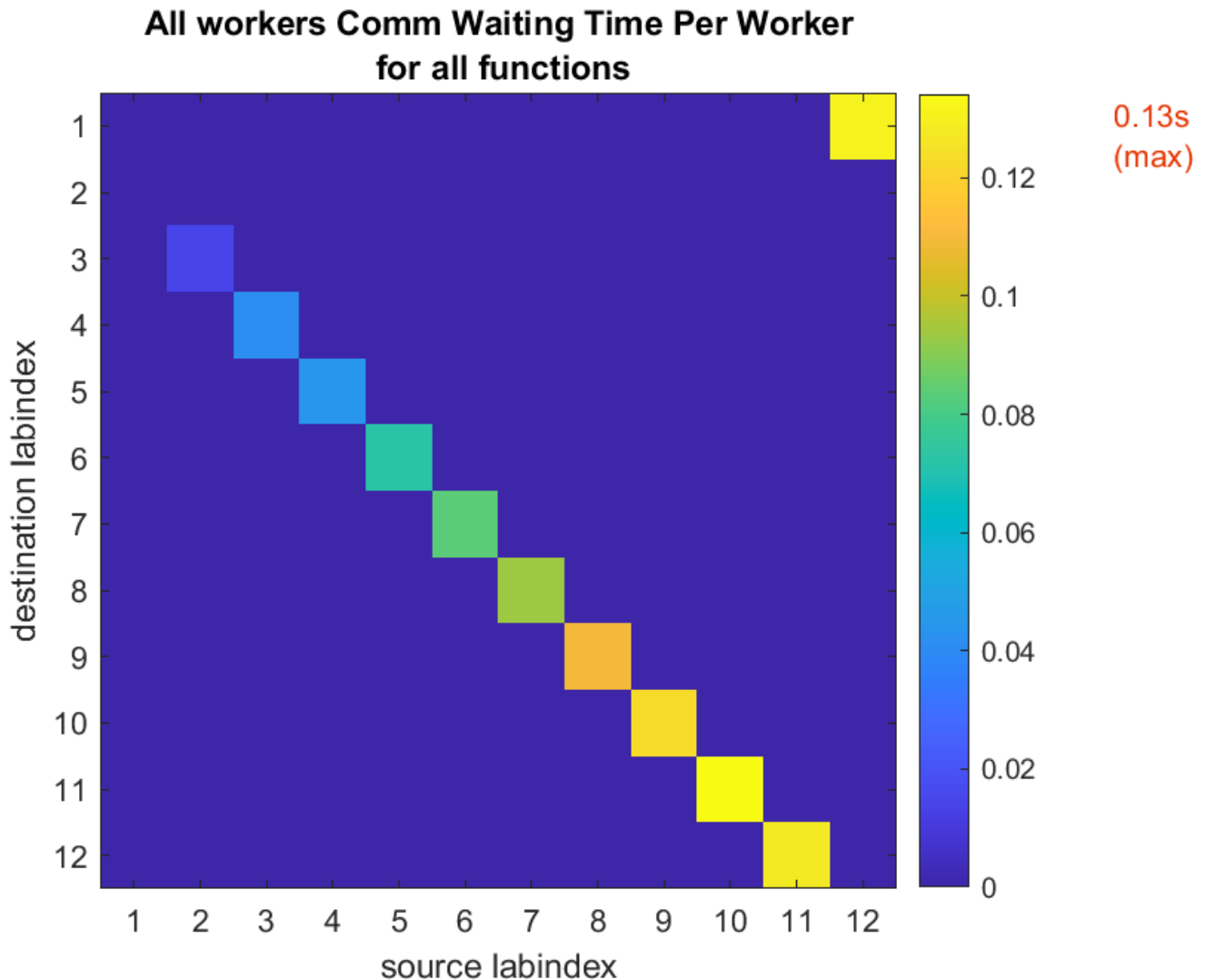
In the plot above, you can see the unnecessary waiting using **Plot All Per Worker Communication**. This waiting time increases because `labReceive` blocks until the corresponding paired `labSend` has completed. Hence, you get sequential communication even though subsequent labs only need the data that is originating in the immediate neighbor `labindex`.

Using `labSendReceive` to Implement this Algorithm

You can use `labSendReceive` to send and receive data simultaneously from the lab that you depend on to get minimal waiting time. See this in the corrected version of the communication pattern implemented in `pctdemo_aux_profcomm`. Clearly, using `labSendReceive` is not possible if you

need to receive data before you can send it. In such cases, use `labSend` and `labReceive` to ensure chronological order. However, in cases like this example, when there is no need to receive data before sending, use `labSendReceive`. Let's profile this version without resetting the data collected on the previous version (use `mpiprofile resume`).

```
spmd
    labBarrier;
    mpiprofile resume;
    pctdemo_aux_profcomm;
end

Lab  1:
  sending to 2 receiving from 12
Lab  2:
  sending to 3 receiving from 1
Lab  3:
  sending to 4 receiving from 2
Lab  4:
  sending to 5 receiving from 3
Lab  5:
  sending to 6 receiving from 4
Lab  6:
  sending to 7 receiving from 5
Lab  7:
  sending to 8 receiving from 6
Lab  8:
  sending to 9 receiving from 7
Lab  9:
  sending to 10 receiving from 8
Lab 10:
  sending to 11 receiving from 9
Lab 11:
  sending to 12 receiving from 10
Lab 12:
  sending to 1 receiving from 11

mpiprofile viewer
```

This corrected version reduces the waiting time to effectively zero. To see this, click **Plot All Per Worker Communication** *after* selecting `pctdemo_aux_profcomm`. The same communication pattern, described above, now spends nearly no time waiting using `labSendReceive` (see the **Comm Waiting Time Per Worker** plot below).
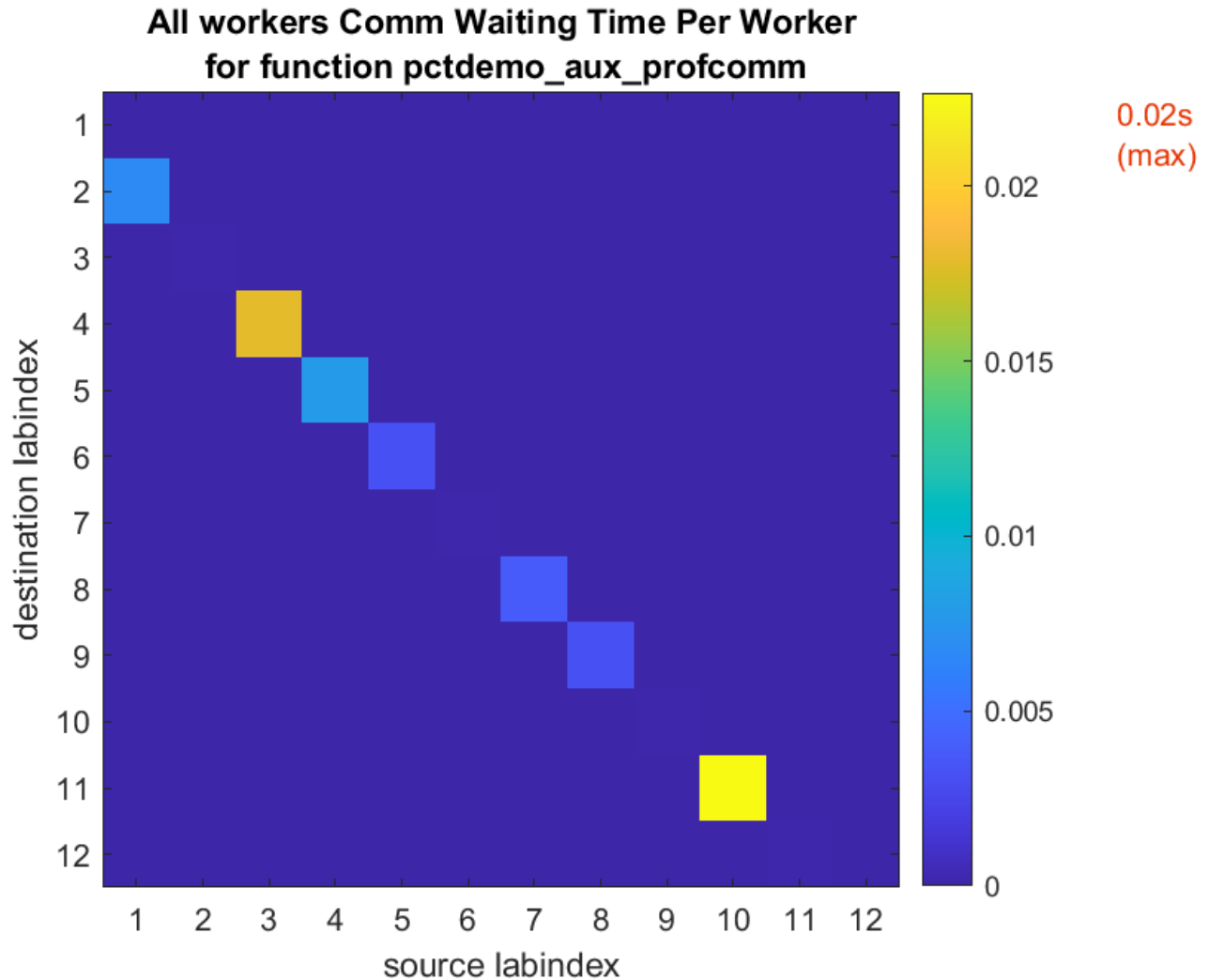
The Plot Color Scheme

For each 2-D image plot, the coloring scheme is normalized to the task at hand. Therefore, do not use the coloring scheme in the plot shown above to compare with other plots, since colors are normalized and are dependent on the maximum value (seen in the top right in brown). For this example, using the max value is the best way to compare the huge difference in waiting times when we use `pctdemo_aux_profcomm` instead of `pctdemo_aux_profbadcomm`.

Profiling Load Unbalanced Codistributed Arrays

This example shows how to profile the implicit communication that occurs when using an unevenly distributed array. For getting started with parallel profiling, see "Profiling Parallel Code" on page 5-32.

This example shows how to use the parallel profiler in the case of an unevenly distributed array. The easiest way to create a codistributed array is to pass a `codistributor` as an argument, such as in `rand(N, codistributor)`. This evenly distributes your matrix of size N between your MATLAB® workers. To get an unbalanced data distribution, you can get some number of columns of a codistributed array as a function of `labindex`.

The plots in this example are produced from a 12-node MATLAB cluster. Everything else is shown running on a four-node local cluster.

The Algorithm

The algorithm we chose for this codistributed array is relatively simple. We generate a large matrix such that each lab gets an approximately 512-by-512 submatrix, except for the first lab. The first lab receives only one column of the matrix and the other columns are assigned to the last lab. Thus, on a four-lab cluster, lab 1 keeps only a 1-by-512 column, labs 2 and 3 have their allotted partitions, and lab 4 has its allotted partition plus the additional columns (left over from lab 1). The end result is an unbalanced workload when doing zero communication element-wise operations (such as `sin`) and communication delays with data parallel operations (such as `codistributed/mtimes`). We start with a data parallel operation first (`codistributed/mtimes`). We then perform, in a loop, `sqrt`, `sin`, and inner product operations, all of which only operate on individual elements of the matrix.

The MATLAB file code for this example can be found in: `pctdemo_aux_profdistarray`

In this example, the size of the matrix differs depending on the number of MATLAB workers (`numlabs`). However, it takes approximately the same amount of computation time (not including communication) to run this example on any cluster, so you can try using a larger cluster without having to wait a long time.

```
spmd
    labBarrier; % synchronize all the labs
    mpiprofile reset
    mpiprofile on
    pctdemo_aux_profdistarray();
end
```

```
Analyzing and transferring files to the workers ...done.
Lab 1:
  This lab has 1024 rows and 1 columns of a codistributed array
Lab 2:
  This lab has 1024 rows and 256 columns of a codistributed array
Lab 3:
  This lab has 1024 rows and 256 columns of a codistributed array
Lab 4:
  This lab has 1024 rows and 511 columns of a codistributed array
Lab 1:
  Calling mtimes on codistributed arrays
  Calling embarrassingly parallel math functions (i.e. no communication is required)
  on a codistributed array.
  Done
```

```
Lab 2:
  Calling mtimes on codistributed arrays
  Calling embarrassingly parallel math functions (i.e. no communication is required)
  on a codistributed array.
  Done
Lab 3:
  Calling mtimes on codistributed arrays
  Calling embarrassingly parallel math functions (i.e. no communication is required)
  on a codistributed array.
  Done
Lab 4:
  Calling mtimes on codistributed arrays
  Calling embarrassingly parallel math functions (i.e. no communication is required)
  on a codistributed array.
  Done
```

```
mpiprofile viewer
```

First, browse the Function Summary Report, making sure it is sorted by the execution time by clicking the **Total Time** column. Then follow the link for the function `pctdemo_aux_profdistarray` to see the Function Detail Report.

The Busy Line Table in the Function Detail Report

Each MATLAB function entry has its own **Busy Line** table, which is useful if you want to profile multiple programs or examples at the same time.

- In the **Function Detail Report**, observe the communication information for the executed MATLAB code on a line-by-line basis.

- Compare profiling information using the Busy Line table. Click **Compare max vs. min TotalTime**. Observe the Busy Line table and check to see which line numbers took the most time by sorting the time field using the drop-down list. There are no for-loops in this code and no increasing complexity. However, there still is a large difference in computation load between the labs. Look at the `sqrt( sin( D .* D ) );` line.

Lines where the most time was spent including the top 5 code lines from the comparison worker (maroon)

Line Number (for worker 4 and *1*)	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
35	D2 = sqrt(sin(D .* D));	50 *50*	0.810 s *0.261 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	75.6% *49.0%*	
27	D1 = D * D * D;	1 *1*	0.199 s *0.189 s*	12.00 Mb *8.02 Mb*	8.02 Mb *15.98 Mb*	0.000 s *0.071 s*	0.026 s *0.027 s*	18.6% *35.5%*	
22	D = codistributed.rand(n, n, c...	1 *1*	0.043 s *0.064 s*	81 b *6.46 Kb*	3.23 Kb *162 b*	0.003 s *0.018 s*	0.001 s *0.001 s*	4.0% *12.0%*	
21	codistr = codistributor1d(2, p...	1 *1*	0.014 s *0.010 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	1.3% *1.9%*	
37	fprintf('Done\n');	1 *1*	0.002 s *0 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	0.2% *0%*	
29	labBarrier;	1 *1*	0 s *0.006 s*	0 b *0 b*	0 b *0 b*	0.000 s *0.006 s*	0 s *0 s*	0% *1.1%*	
All other lines			0.003 s	0 b	0 b	0.000 s	0 s	0.3%	
Totals			1.071 s *0.533 s*	12.00 Mb *8.02 Mb*	8.02 Mb *15.98 Mb*	0.004 s *0.094 s*	0.027 s *0.028 s*	100%	

Despite the fact that no communication is required for this element-wise operation, the performance is not optimal, because some labs do more work than others. In the second row, (D*D*D), the total time taken is the same on both labs. However, the **Data Rec** and **Data Sent** columns show a large difference in the amount of data sent and received. The time taken for this `mtimes` is similar on all labs, because the codistributed array communication implicitly synchronizes all the labs.

In the ninth column (from the left) of the Busy Line table, a bar shows the percentage for the selected field (using the **Sort busy lines** list box). These bars can also be used to visually compare **Total Time**, and **Data Sent** or **Data Received** of the main and comparison labs.

Observing Codistributed Array Operations in Plot View

If you click the relevant function name and are in the **Function Detail Report**, you get more specific information about a codistributed array operation.

- To get the inter-lab communication data click `Plot All Per Worker Communication`. In the first figure, you can see lab 1 transferring the most amount of data, and the last lab (lab 12) transferring the least amount of data.

- To go back to the **Function Summary Report**, click **Home** and then click on the `pctdemo_aux_profdistarray` link to view the Busy Line table again.

Using the comparisons, you can also see the amount of data communicated between each lab. This is constant for all labs except for the first and last labs. When there is no explicit communication, this indicates a distribution problem. In a typical codistributed array `mtimes` operation, labs that have the least amount of data (e.g., lab 1) receive all the required data from their neighboring labs (e.g., lab 2).

The Data Received Plot
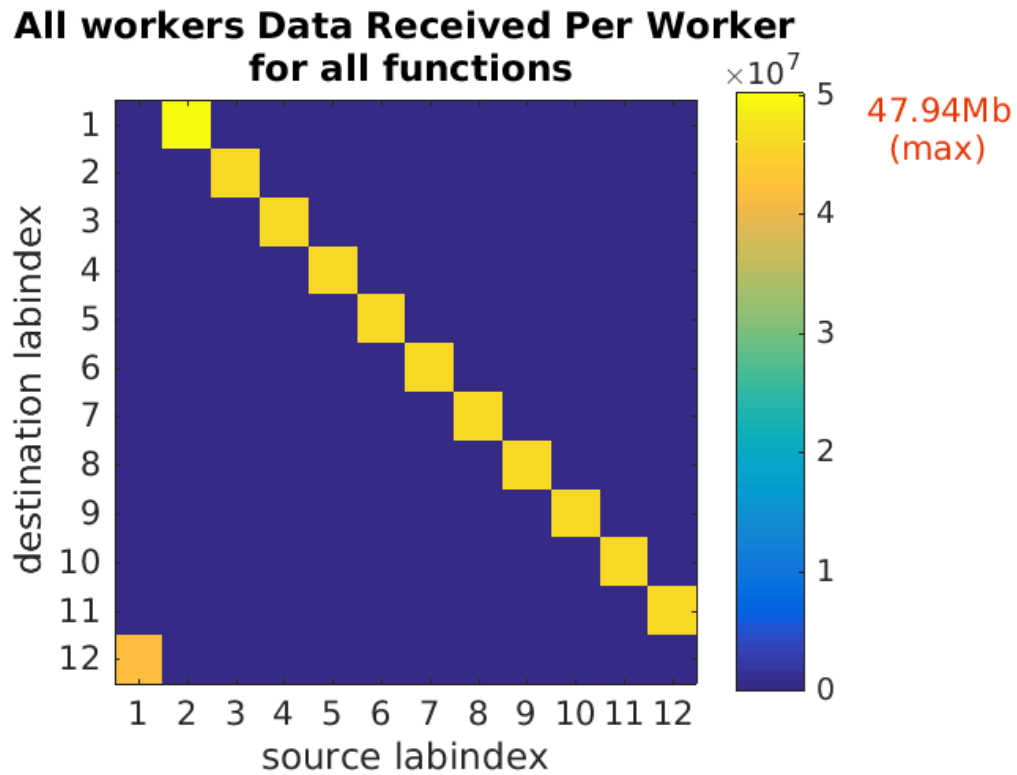
In this **Data Received Per Worker** plot, there is a significant decrease in the amount of data transferred by the last lab and an increase in the amount transferred by the first lab. Observing the Receive Communication Time plot (not shown) further illustrates that there is something different going on in the first lab. That is, the first lab is spending the longest amount of time in communication.

As you can see, the uneven distribution of a matrix causes unnecessary communication delays when using data parallel codistributed array operations and uneven work distribution with task parallel (no communication) operations. In addition, labs (like the first lab in this example) that are receiving more data start with the least amount of data prior to the codistributed array operation.

Sequential Blackjack

This example plays the card game of blackjack, also known as 21. We simulate a number of players that are independently playing thousands of hands at a time, and display payoff statistics. Simulating the playing of blackjack is representative of Monte Carlo analysis of financial instruments. The simulation can be done completely in parallel, except for the data collection at the end.

For details about the computations, view the code for pctdemo_setup_blackjack.

Related examples:

- "Distributed Blackjack" on page 9-226

Load the Example Settings and the Data

We start by getting the example difficulty level. If you want to use a different example difficulty level, use `paralleldemoconfig` and then run this example again.

```
difficulty = pctdemo_helper_getDefaults();
```

We get the number of players and the number of hands each player plays from `pctdemo_setup_blackjack`. The `difficulty` parameter controls the number of players that we simulate. You can view the code for pctdemo_setup_blackjack for full details.

```
[fig, numHands, numPlayers] = pctdemo_setup_blackjack(difficulty);
```

Run the Simulation

We use `pctdemo_task_blackjack` to simulate a single player who plays `numHands` hands, and we call that function `numPlayers` times to simulate all the players. Because the separate invocations of the function are independent one of another, we can easily use the Parallel Computing Toolbox to perform these simulations. You can view the code for pctdemo_task_blackjack for full details.

```
startTime = clock;
S = zeros(numHands, numPlayers); % Preallocate for the results.
for i = 1:numPlayers
    S(:, i) = pctdemo_task_blackjack(numHands, 1);
end
```

Measure the Elapsed Time

The time used for the sequential simulations should be compared against the time it takes to perform the same set of calculations using the Parallel Computing Toolbox in the "Distributed Blackjack" on page 9-226 example. The elapsed time varies with the underlying hardware.

```
elapsedTime = etime(clock, startTime);
fprintf('Elapsed time is %2.1f seconds\n', elapsedTime);
```

```
Elapsed time is 34.7 seconds
```

Plot the Results

We display the expected fraction of the bet that is won or lost in each hand, along with the confidence interval. We also show the evolution of the winnings and losses of each of the players we simulate. You can view the code for pctdemo_plot_blackjack for full details.

```
pctdemo_plot_blackjack(fig, S);
```

Observed probability of payoffs

Mean fraction of bet that is lost/won: -0.0023 ± 0.0066

Total winnings: $ -2740

6 players playing 20000 hands each
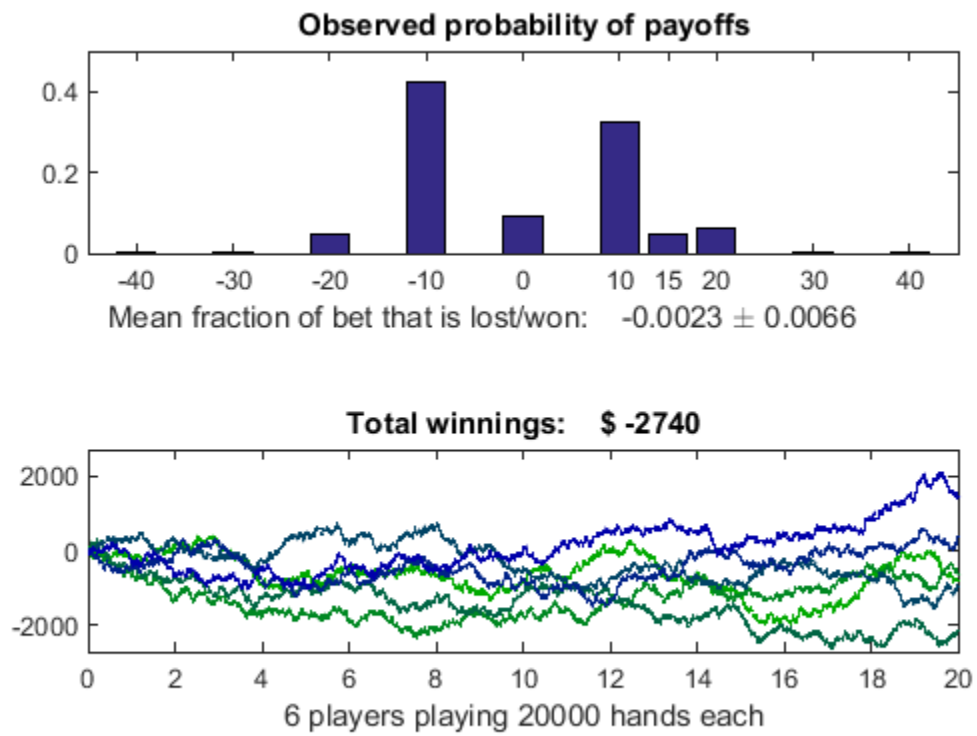
Distributed Blackjack

This example uses the Parallel Computing Toolbox™ to play the card game of blackjack, also known as 21. We simulate a number of players that are independently playing thousands of hands at a time, and display payoff statistics. Simulating the playing of blackjack is representative of Monte Carlo analysis of financial instruments. The simulation can be done completely in parallel, except for the data collection at the end.

For details about the computations, view the code for pctdemo_setup_blackjack

Related examples:

- "Sequential Blackjack" on page 9-224

Analyze the Sequential Problem

Because the blackjack players are independent one of another, we can simulate them in parallel. We do this by dividing the problem up into a number of smaller tasks.

Load the Example Settings and the Data

The example uses the default profile when identifying the cluster to use. "Discover Clusters and Use Cluster Profiles" on page 5-11 explains how to create new profiles and how to change the default profile. If you want to use a different example difficulty level or number of tasks, use `paralleldemoconfig` and then run this example again.

```
[difficulty, myCluster, numTasks] = pctdemo_helper_getDefaults();
```

We get the number of players and the number of hands each player plays from `pctdemo_setup_blackjack`. The `difficulty` parameter controls the number of players that we simulate. You can view the code for pctdemo_setup_blackjack for full details.

```
[fig, numHands, numPlayers] = pctdemo_setup_blackjack(difficulty);
```

Divide the Work into Smaller Tasks

We divide the simulation of the `numPlayers` players among the `numTasks` tasks. Thus, task `i` simulates `splitPlayers{i}` players.

```
[splitPlayers, numTasks] = pctdemo_helper_split_scalar(numPlayers, ...
                                                        numTasks);
fprintf(['This example will submit a job with %d task(s) ' ...
         'to the cluster.\n'], numTasks);
```

```
This example will submit a job with 4 task(s) to the cluster.
```

Create and Submit the Job

We create a job and one task in the job for each split. Notice that the task function is the same function that was used in the sequential example. You can view the code for pctdemo_task_blackjack for full details.

```
startTime = clock;
job = createJob(myCluster);
for i = 1:numTasks
    createTask(job, @pctdemo_task_blackjack, 1, ...
```

```
            {numHands, splitPlayers(i)});
end
```

We can now submit the job and wait for it to finish.

```
submit(job);
wait(job);
```

Retrieve the Results

Let us verify that we received all the results that we expected. `fetchOutputs` will throw an error if the tasks did not complete successfully, in which case we need to delete the job before throwing the error.

```
try
    jobResults = fetchOutputs(job);
catch err
    delete(job);
    rethrow(err);
end
```

Collect the task results into a numHands-by-numPlayers matrix.

```
S = cell2mat(jobResults');
```

We have now finished all the verifications, so we can delete the job.

```
delete(job);
```

Measure the Elapsed Time

The time used for the distributed simulations should be compared against the time it takes to perform the same set of calculations in the "Sequential Blackjack" on page 9-224 example. The elapsed time varies with the underlying hardware and network infrastructure.

```
elapsedTime = etime(clock, startTime);
fprintf('Elapsed time is %2.1f seconds\n', elapsedTime);
```

```
Elapsed time is 27.5 seconds
```

Plot the Results

We display the expected fraction of the bet that is won or lost in each hand, along with the confidence interval. We also show the evolution of the winnings and losses of each of the players we simulate. You can view the code for pctdemo_plot_blackjack for full details.

```
pctdemo_plot_blackjack(fig, S);
```

Observed probability of payoffs

Mean fraction of bet that is lost/won: -0.0065 ± 0.0066

Total winnings: $ -7790

6 players playing 20000 hands each
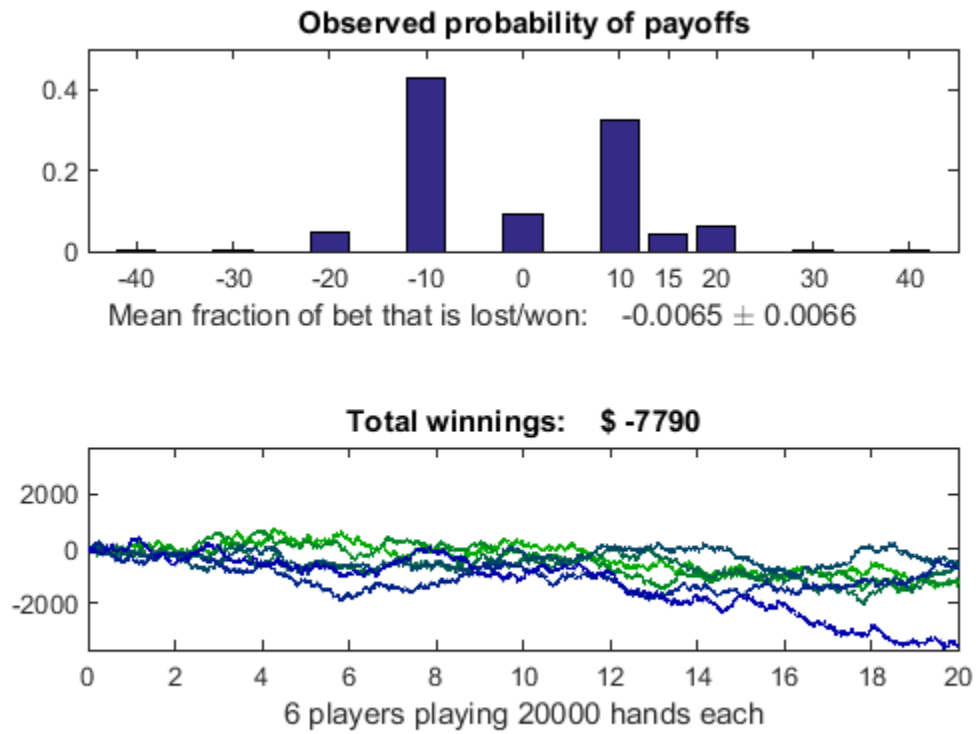
Parfeval Blackjack

This example uses Parallel Computing Toolbox™ to play the card game of blackjack, also known as 21. We simulate a number of players that are independently playing thousands of hands at a time, and display payoff statistics. This example runs the simulations asynchronously on a parallel pool of workers, using `parfeval`. In this way, we can update a display of the results as they become available.

Related examples:

- "Sequential Blackjack" on page 9-224
- "Simple Benchmarking of PARFOR Using Blackjack" on page 9-102

You can find the code shown in this example in the function:

```
function paralleldemo_blackjack_parfeval
```

Analyze the sequential problem

Because the blackjack players are independent of one another, we can simulate them in parallel. We do this by dividing the problem up into a number of function evaluations. We run a maximum of `numPlayers` simulations, where each player plays `numHands` hands of blackjack. We plot the results as soon as they become available, and we terminate the simulation if the elapsed time exceeds `maxSimulationTime` seconds, or if the user cancels execution.

```
numPlayers        = 100;
numHands          = 5000;
maxSimulationTime = 20;
```

Divide the work into individual function evaluations

We call the `parfeval` function to request evaluation of the simulation on the parallel pool workers. The parallel pool will be created automatically if necessary. The `parfeval` function returns a `parallel.Future` object, which we use to access results when they become available. You can view the code for pctdemo_task_blackjack for full details.

```
for idx = numPlayers:-1:1
    futures(idx) = parfeval(@pctdemo_task_blackjack, 1, numHands, 1);
end
% Create an onCleanup to ensure we do not leave any futures running when we exit
% this example.
cancelFutures = onCleanup(@() cancel(futures));
```

Set up for collecting results and monitoring progress

The parallel pool workers immediately start running `pctdemo_task_blackjack`, and we can collect and display results as soon as they are available by using the `fetchNext` method. We use `resultsSoFar` to accumulate results. We update the array `completed` to indicate which elements of `futures` have completed, and increment the counter `numCompleted`. We supply the optional argument `timeout` to the `fetchNext` method so that it returns quickly if no new results are available.

```
resultsSoFar = zeros(numHands, numPlayers); % Allocate space for all results
completed    = false(1, numPlayers);        % Has a given future completed yet
timeout      = 2;                            % fetchNext timeout in seconds
```

```
numCompleted = 0;                              % How many simulations have completed
fig          = pctdemo_setup_blackjack(1);  % Create a figure to display results

% Build a waitbar with a cancel button, using appdata to track
% whether the cancel button has been pressed.
hWaitBar = waitbar(0, 'Blackjack progress', 'CreateCancelBtn', ...
                   @(src, event) setappdata(gcbf(), 'Cancelled', true));
setappdata(hWaitBar, 'Cancelled', false);
```
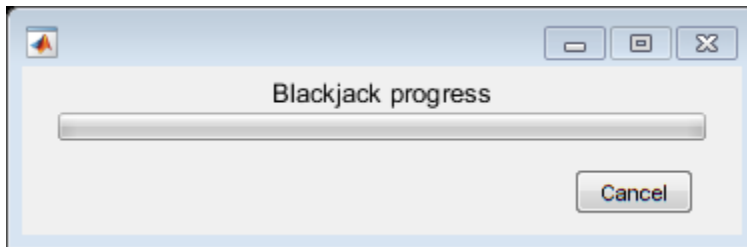

Collect and display results as they become available

We collect and display results by calling `fetchNext` in a loop until we have seen `numPlayers` results. When `fetchNext` returns new results, we assign the results into `resultsSoFar`, update the `completed` array and the `numCompleted` counter, and update the plot. We abort the loop early if the user presses the cancel button on the waitbar, or the `maxSimulationTime` expires.

```
startTime = clock();
while numCompleted < numPlayers

    % fetchNext blocks execution until one element of futures has completed.  It
    % then returns the index into futures of the element that has now completed,
    % and the results from execution.
    [completedIdx, resultThisTime] = fetchNext(futures, timeout);

    % If fetchNext timed out returning an empty completedIdx, do not attempt to
    % process results.
    if ~isempty(completedIdx)
        numCompleted = numCompleted + 1;
        % Update list of completed futures.
        completed(completedIdx) = true;
        % Fill out portion of results.
        resultsSoFar(:, completedIdx) = resultThisTime;
        % Update plot.
        pctdemo_plot_blackjack(fig, resultsSoFar(:, completed), false);
    end

    % Check to see if we have run out of time.
    timeElapsed = etime(clock(), startTime);
    if timeElapsed > maxSimulationTime
        fprintf('Simulation terminating: maxSimulationTime exceeded.\n');
        break;
    end

    % Check to see if the cancel button was pressed.
    if getappdata(hWaitBar, 'Cancelled')
        fprintf('Simulation cancelled.\n');
        break;
    end
```

```
    % Update the waitbar.
    fractionTimeElapsed     = timeElapsed / maxSimulationTime;
    fractionPlayersCompleted = numCompleted / numPlayers;
    fractionComplete        = max(fractionTimeElapsed, fractionPlayersCompleted);
    waitbar(fractionComplete, hWaitBar);
end
fprintf('Number of simulations completed: %d\n', numCompleted);

% Now the simulation is complete, we can cancel the futures and delete
% the waitbar.
cancel(futures);
delete(hWaitBar);
```

```
Simulation terminating: maxSimulationTime exceeded.
Number of simulations completed: 74
```


Observed probability of payoffs

Mean fraction of bet that is lost/won: -0.0057 ± 0.0037

Total winnings: $ -21270

74 players playing 5000 hands each
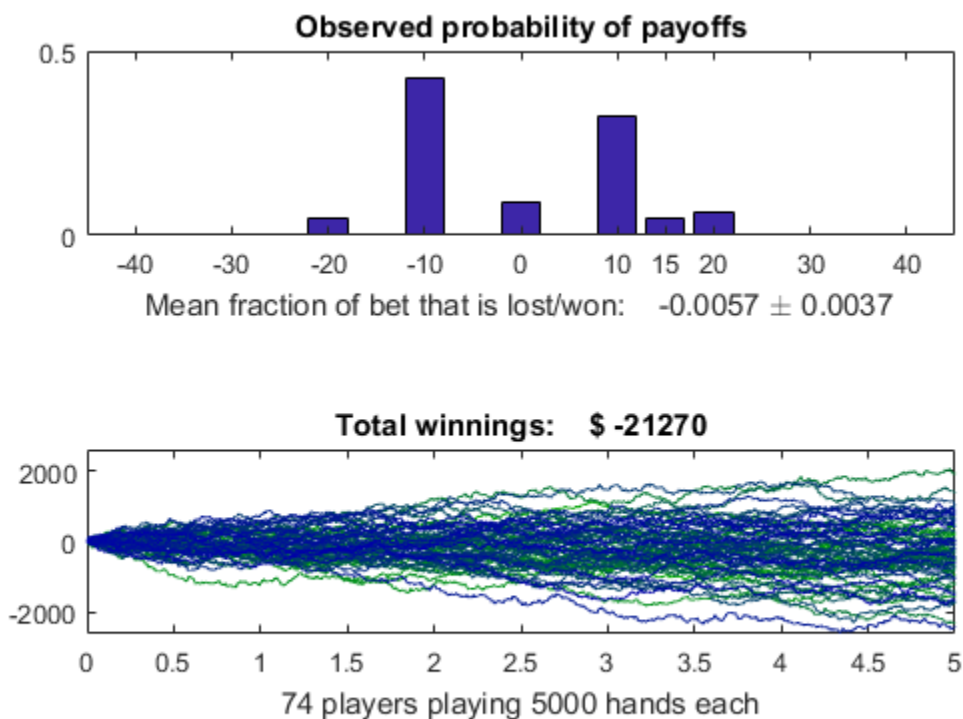
```
end
```

Numerical Estimation of Pi Using Message Passing

This example shows the basics of working with spmd statements, and how they provide an interactive means of performing parallel computations. We do this by performing relatively simple computations to approximate pi.

Related Documentation:

- `spmd` in the Parallel Computing Toolbox™ User's Guide

Related Examples:

- "Using GOP to Achieve MPI_Allreduce Functionality" on page 9-119

The code shown in this example can be found in this function:

```
function paralleldemo_quadpi_mpi
```

Introduction

We intend to use the fact that

$$\int_0^1 \frac{4}{1+x^2}\,dx = 4(atan(1) - atan(0)) = \pi$$

to approximate pi by approximating the integral on the left.

We intend to have the parallel pool perform the calculations in parallel, and to use the `spmd` keyword to mark the parallel blocks of code. We first look at the size of the parallel pool that is currently open.

```
p = gcp;
p.NumWorkers
```

```
ans =

    12
```

Parallelize the Computations

We approximate pi by the numerical integral of `4/(1 + x^2)` from 0 to 1.

```
type pctdemo_aux_quadpi.m
```

```
function y = pctdemo_aux_quadpi(x)
%PCTDEMO_AUX_QUADPI Return data to approximate pi.
%   Helper function used to approximate pi.  This is the derivative
%   of 4*atan(x).

%   Copyright 2008 The MathWorks, Inc.
y = 4./(1 + x.^2);
```

We divide the work between the workers (labs) by having each worker calculate the integral of the function over a subinterval of [0, 1] as shown in the picture.

$$F(x) = \frac{4}{1 + x^2}$$
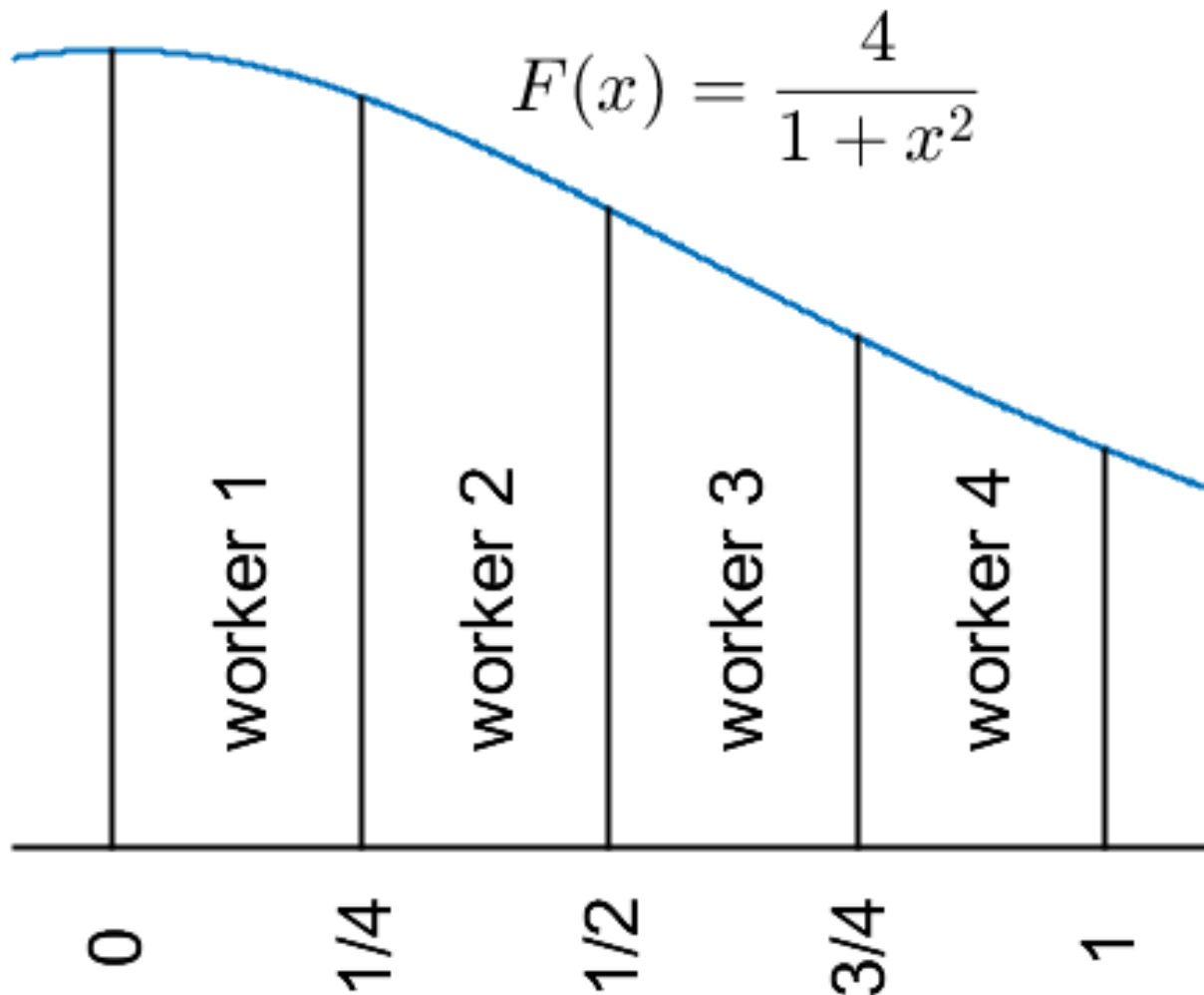
We define the variables a and b on all the workers, but let their values depend on labindex so that the intervals [a, b] correspond to the subintervals shown in the figure. We then verify that the intervals are correct. Note that the code in the body of the spmd statement is executed in parallel on all the workers in the parallel pool.

```
spmd
    a = (labindex - 1)/numlabs;
    b = labindex/numlabs;
    fprintf('Subinterval: [%-4g, %-4g]\n', a, b);
end
```

```
Lab  1:
  Subinterval: [0    , 0.0833333]
Lab  2:
  Subinterval: [0.0833333, 0.166667]
Lab  3:
  Subinterval: [0.166667, 0.25]
```

```
Lab  4:
  Subinterval: [0.25, 0.333333]
Lab  5:
  Subinterval: [0.333333, 0.416667]
Lab  6:
  Subinterval: [0.416667, 0.5 ]
Lab  7:
  Subinterval: [0.5 , 0.583333]
Lab  8:
  Subinterval: [0.583333, 0.666667]
Lab  9:
  Subinterval: [0.666667, 0.75]
Lab 10:
  Subinterval: [0.75, 0.833333]
Lab 11:
  Subinterval: [0.833333, 0.916667]
Lab 12:
  Subinterval: [0.916667, 1   ]
```

We let all the workers now use a MATLAB quadrature method to approximate each integral. They all operate on the same function, but on the different subintervals of [0,1] shown in the figure above.

```
spmd
    myIntegral = integral(@pctdemo_aux_quadpi, a, b);
    fprintf('Subinterval: [%-4g, %-4g]   Integral: %4g\n', ...
            a, b, myIntegral);
end
```

```
Lab  1:
  Subinterval: [0   , 0.0833333]   Integral: 0.332565
Lab  2:
  Subinterval: [0.0833333, 0.166667]   Integral: 0.32803
Lab  3:
  Subinterval: [0.166667, 0.25]   Integral: 0.31932
Lab  4:
  Subinterval: [0.25, 0.333333]   Integral: 0.307088
Lab  5:
  Subinterval: [0.333333, 0.416667]   Integral: 0.292162
Lab  6:
  Subinterval: [0.416667, 0.5 ]   Integral: 0.275426
Lab  7:
  Subinterval: [0.5 , 0.583333]   Integral: 0.257707
Lab  8:
  Subinterval: [0.583333, 0.666667]   Integral: 0.239713
Lab  9:
  Subinterval: [0.666667, 0.75]   Integral: 0.221994
Lab 10:
  Subinterval: [0.75, 0.833333]   Integral: 0.204949
Lab 11:
  Subinterval: [0.833333, 0.916667]   Integral: 0.188836
Lab 12:
  Subinterval: [0.916667, 1   ]   Integral: 0.173804
```

Add the Results

The workers have all calculated their portions of the integral of the function, and we add the results together to form the entire integral over [0, 1]. We use the `gplus` function to add `myIntegral` across all the workers and return the sum on all the workers.

```matlab
spmd
    piApprox = gplus(myIntegral);
end
```

Inspect Results in the Client

Since the variable `piApprox` was assigned to inside an spmd statement, it is accessible on the client as a Composite. Composite objects resemble cell arrays with one element for each worker. Indexing into a Composite brings back the corresponding value from the worker to the client.

```matlab
approx1 = piApprox{1};    % 1st element holds value on worker 1.
fprintf('pi           : %.18f\n', pi);
fprintf('Approximation: %.18f\n', approx1);
fprintf('Error        : %g\n', abs(pi - approx1))
```

```
pi           : 3.141592653589793100
Approximation: 3.141592653589792700
Error        : 4.44089e-16
```

Query and Cancel parfeval Futures

When you use `parfeval` or `parfevalOnAll` to run computations in the background, you create objects called futures. You can use the `State` property of a future to find out whether it is running, queued or finished. You can also use the `FevalQueue` property of a parallel pool to access running and queued futures. To cancel futures, you can use the `cancel` function. In this example, you:

- Use `cancel` to cancel futures directly.
- Check completion errors on completed futures.
- Use the `FevalQueue` property to access futures.

Add Work to Queue

Create a parallel pool `p` with two workers.

```
p = parpool(2);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

When you use `parfeval` to run computations in the background, the function creates and adds a future for each computation to the pool queue. Tasks remain in the queue until a worker becomes idle. When a worker becomes idle, it starts to compute a task if the queue is not empty. When a worker completes a task, the task is removed from the queue and the worker becomes idle.

Use `parfeval` to create an array of futures `f` by instructing workers to execute the function `pause`. Use an argument of `1` for the third future, and an argument of `Inf` for all other futures.

```
for n = 1:5
    if n == 3
        f(n) = parfeval(@pause,0,1);
    else
        f(n) = parfeval(@pause,0,Inf);
    end
end
```

Each use of `parfeval` returns a future object that represents the execution of a function on a worker. Except for the third future, every future will take an infinite amount of time to compute. The future created by `parfeval(@pause,0,Inf)` is an extreme case of a future which can slow down a queue.

Cancel Futures Directly

You can use the `State` property to obtain the status of futures. Construct a cell array of the state of each future in `f`.

```
{f.State}

ans = 1×5 cell
    {'running'}    {'running'}    {'queued'}    {'queued'}    {'queued'}
```

Every task except for the third pauses forever.

Cancel the second future directly with `cancel`.

```
cancel(f(2));
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'running'}    {'queued'}    {'queued'}
```

After you cancel the second future, the third future runs. Wait until the third future completes, then examine the states again.

```
wait(f(3));
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'finished'}    {'running'}    {'queued'}
```

The third future now has the state `'finished'`.

Check Completion Errors

When a future completes, its `State` property becomes `'finished'`. To distinguish between futures which are cancelled and complete normally, use the `Error` property.

```
fprintf("f(2): %s\n", f(2).Error.message)
```

```
f(2): Execution of the future was cancelled.
```

```
fprintf("f(3): %s\n", f(3).Error.message)
```

```
f(3):
```

The code cancels the second future, as the message property indicates. The second future was cancelled, as stated in the `message` property. The third future completes without error, and therefore does not have an error message.

Cancel Futures in Pool Queue

You can use the `FevalQueue` property to access the futures in the pool queue.

```
p.FevalQueue
```

```
ans =
 FevalQueue with properties:

       Number Queued: 1
      Number Running: 2
```

The queue has two properties: `RunningFutures` and `QueuedFutures`. The `RunningFutures` property is an array of futures corresponding to tasks that are currently running.

```
disp(p.FevalQueue.RunningFutures)
```

```
 1x2 FevalFuture array:

        ID          State  FinishDateTime  Function  Error
     -------------------------------------------------------
   1   12           running                 @pause
   2   15           running                 @pause
```

The `QueuedFutures` property is an array of futures corresponding to tasks that are currently queued and not running.

```
disp(p.FevalQueue.QueuedFutures)
```

```
 FevalFuture with properties:

                   ID: 16
             Function: @pause
       CreateDateTime: 15-Jul-2020 17:29:37
        StartDateTime:
     Running Duration: 0 days 0h 0m 0s
                State: queued
                Error: none
```

You can cancel a single future or an array of futures. Cancel all the futures in `QueuedFutures`.

```
cancel(p.FevalQueue.QueuedFutures);
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'finished'}    {'running'}    {'finished'}
```

`RunningFutures` and `QueuedFutures` are sorted from newest to oldest, regardless of whether `f` is in order from newest to oldest. Each future has a unique `ID` property for the lifetime of the client. Check the `ID` property of each of the futures in `f`.

```
disp(f)
```

```
 1x5 FevalFuture array:

        ID                State           FinishDateTime  Function  Error
     ----------------------------------------------------------------------
  1    12                running                          @pause
  2    13  finished (unread)  15-Jul-2020 17:29:37        @pause   Error
  3    14  finished (unread)  15-Jul-2020 17:29:39        @pause
  4    15                running                          @pause
  5    16  finished (unread)  15-Jul-2020 17:29:39        @pause   Error
```

Compare the result against the `ID` property of each of the `RunningFutures`.

```
for j = 1:length(p.FevalQueue.RunningFutures)
    rf = p.FevalQueue.RunningFutures(j);
    fprintf("p.FevalQueue.RunningFutures(%i): ID = %i\n", j, rf.ID)
end
```

```
p.FevalQueue.RunningFutures(1): ID = 12
p.FevalQueue.RunningFutures(2): ID = 15
```

Here, `RunningFutures` is an array containing `f(1)` and `f(4)`. If you cancel `RunningFutures(2)`, you cancel the fourth future `f(4)`.

Sometimes, futures are not available in the workspace, for example, if you execute the same piece of code twice before it finishes, or if you use `parfeval` in a function. You can cancel futures that are not available in the workspace.

Clear `f` from the workspace.

```
clear f
```

You can use `RunningFutures` and `QueuedFutures` to access futures that have not yet completed. Use `RunningFutures` to cancel `f(4)`.

```
rf2 = p.FevalQueue.RunningFutures(2);
cancel(rf2)
rf2.State
```

```
ans =
'finished'
```

To cancel all the futures still in the queue, use the following code.

```
cancel([p.FevalQueue.RunningFutures p.FevalQueue.QueuedFutures])
```

Objects

ClusterPool

Parallel pool of workers on a cluster of machines

Description

Use `parpool` to create a parallel pool of workers on a cluster of machines. After you create the pool, parallel pool features, such as `parfor` or `parfeval`, run on the workers. With the `ClusterPool` object, you can interact with the parallel pool.

Creation

Create a parallel pool on a cluster of machines by using the `parpool` function.

```
pool = parpool("myCluster")
```

where `myCluster` is the name of a cluster profile for a cluster of machines.

Properties

AttachedFiles — Files and folders copied to workers
cell array of character vectors

Files and folders copied to workers, specified as a cell array of character vectors. To attach files and folders to the pool, use `addAttachedFiles`.

AutoAddClientPath — Indication whether user-added entries on client path are added to worker paths
`true` (default) | `false`

This property is read-only.

Indication whether user-added entries on client path are added to worker paths, specified as a logical value.

Data Types: `logical`

Cluster — Cluster on which the parallel pool is running
cluster object

This property is read-only.

Cluster on which the parallel pool is running, specified as a `parallel.Cluster` object.

Connected — Flag that indicates whether the parallel pool is running
`true` | `false`

This property is read-only.

Flag that indicates whether the parallel pool is running, specified as a logical value.

Data Types: logical

EnvironmentVariables — Environment variables copied to the workers
cell array of character vectors

This property is read-only.

Environment variables copied to the workers, specified as a cell array of character vectors.

FevalQueue — Queue of FevalFutures to run on the parallel pool
FevalQueue

This property is read-only.

Queue of FevalFutures to run on the parallel pool, specified as an FevalQueue object. You can use this property to check the pending and running future variables of the parallel pool. To create future variables, use parfeval and parfevalOnAll. For more information on future variables, see Future.

Data Types: FevalQueue

IdleTimeout — Time after which the pool shuts down if idle
nonnegative integer

Time in minutes after which the pool shuts down if idle, specified as an integer greater than zero. A pool is idle if it is not running code on the workers. By default 'IdleTimeout' is the same as the value in your parallel preferences. For more information on parallel preferences, see "Specify Your Parallel Preferences" on page 5-9.

NumWorkers — Number of workers comprising the parallel pool
integer

This property is read-only.

Number of workers comprising the parallel pool, specified as an integer.

SpmdEnabled — Indication if pool can run spmd code
true (default) | false

This property is read-only.

Indication if pool can run spmd code, specified as a logical value.

Data Types: logical

Object Functions

addAttachedFiles   Attach files or folders to parallel pool
delete   Shut down parallel pool
listAutoAttachedFiles   List of files automatically attached to job, task, or parallel pool
parfeval   Execute function asynchronously on parallel pool worker
parfevalOnAll   Execute function asynchronously on all workers in parallel pool
ticBytes   Start counting bytes transferred within parallel pool
tocBytes   Read how many bytes have been transferred since calling ticBytes
updateAttachedFiles   Update attached files or folders on parallel pool

See Also

`parpool`

Topics
"Run Code on Parallel Pools" on page 2-55
"Choose Between Thread-Based and Process-Based Environments" on page 2-60

Introduced in R2020a

codistributed

Access elements of arrays distributed among workers in parallel pool

Constructor

`codistributed, codistributed.build`

You can also create a codistributed array explicitly from spmd code or a communicating job task with any of several MATLAB functions.

`eye(___,'codistributed')`	`randi(___,'codistributed')`
`false(___,'codistributed')`	`randn(___,'codistributed')`
`Inf(___,'codistributed')`	`codistributed.cell`
`NaN(___,'codistributed')`	`codistributed.spalloc`
`ones(___,'codistributed')`	`codistributed.speye`
`true(___,'codistributed')`	`codistributed.sprand`
`zeros(___,'codistributed')`	`codistributed.sprandn`
`rand(___,'codistributed')`	

Description

Arrays partitioned among the workers in a pool, are accessible from the workers as codistributed array objects.

Codistributed arrays on workers that you create inside `spmd` statements or from within task functions of communicating jobs can be accessed as distributed arrays on the client.

Methods

classUnderlying	(Not recommended) Class of elements within gpuArray or distributed array
codistributed.cell	Create codistributed cell array
codistributed.colon	Distributed colon operation
codistributed.spalloc	Allocate space for sparse codistributed matrix
codistributed.speye	Create codistributed sparse identity matrix
codistributed.sprand	Create codistributed sparse array of uniformly distributed pseudo-random values
codistributed.sprandn	Create codistributed sparse array of uniformly distributed pseudo-random values
eye	Identity matrix
false	Array of logical 0 (false)
gather	Transfer distributed array or gpuArray to local workspace
getCodistributor	Codistributor object for existing codistributed array
getLocalPart	Local portion of codistributed array
globalIndices	Global indices for local part of codistributed array
Inf	Array of infinity
isaUnderlying	(Not recommended) True if distributed array's underlying elements are of specified class
iscodistributed	True for codistributed array
rand	Array of rand values
redistribute	Redistribute codistributed array with another distribution scheme
sparse	Create sparse distributed or codistributed matrix
true	Array of logical 1 (true)

The methods for codistributed arrays are too numerous to list here. Most resemble and behave the same as built-in MATLAB functions. See "Run MATLAB Functions with Distributed Arrays" on page 4-18.

Also among the methods there are several for examining the characteristics of the array itself. Most behave like the MATLAB functions of the same name:

Function	Description
iscodistributed	Determine if array is codistributed
isreal	Determine if array elements are real
isUnderlyingType	Determine if underlying data in the array is of specified type
length	Length of vector or largest array dimension
ndims	Number of dimensions in the array
size	Size of array dimensions
underlyingType	Class (data type) of the underlying data in the array

See Also

distributed | gather | getLocalPart | parpool | spmd

Topics
"Create and Use Distributed Arrays" on page 1-4
"Working with Codistributed Arrays" on page 4-4
"Run MATLAB Functions with Distributed Arrays" on page 4-18
"Nondistributed Versus Distributed Arrays" on page 4-2

Introduced in R2008b

codistributor1d

1-D distribution scheme for codistributed array

Constructor

codistributor1d

Description

A codistributor1d object defines the 1-D distribution scheme for a codistributed array. The 1-D codistributor distributes arrays along a single specified dimension, the distribution dimension, in a noncyclic, partitioned manner.

For help on codistributor1d, including a list of links to individual help for its methods and properties, type

help codistributor1d

Methods

codistributor1d.defaultPartition	
	Default partition for codistributed array
globalIndices	Global indices for local part of codistributed array
isComplete	True if codistributor object is complete

Properties

Property	Description
Dimension	Distributed dimension of codistributor1d object
Partition	Partition scheme of codistributor1d object

Introduced in R2009b

codistributor2dbc

2-D block-cyclic distribution scheme for codistributed array

Constructor

codistributor2dbc

Description

A codistributor2dbc object defines the 2-D block-cyclic distribution scheme for a codistributed array. The 2-D block-cyclic codistributor can only distribute two-dimensional matrices. It distributes matrices along two subscripts over a rectangular computational grid of labs in a blocked, cyclic manner. The parallel matrix computation software library called ScaLAPACK uses the 2-D block-cyclic codistributor.

For help on codistributor2dbc, including a list of links to individual help for its methods and properties, type

```
help codistributor2dbc
```

Methods

codistributor2dbc.defaultLabGrid	
	Default computational grid for 2-D block-cyclic distributed arrays
globalIndices	Global indices for local part of codistributed array
isComplete	True if codistributor object is complete

Properties

Property	Description
BlockSize	Block size of codistributor2dbc object
LabGrid	Lab grid of codistributor2dbc object
Orientation	Orientation of codistributor2dbc object

Introduced in R2009b

Composite

Access nondistributed variables on multiple workers from client

Constructor

`Composite`

Description

Variables that exist on the workers running an `spmd` statement are accessible on the client as a Composite object. A Composite resembles a cell array with one element for each worker. So for Composite `C`:

```
C{1} represents value of C on worker1
C{2} represents value of C on worker2
etc.
```

`spmd` statements create Composites automatically, which you can access after the statement completes. You can also create a Composite explicitly with the `Composite` function.

Methods

exist	Check whether Composite is defined on workers
subsasgn	Subscripted assignment for Composite
subsref	Subscripted reference for Composite

Other methods of a Composite object behave similarly to these MATLAB array functions:

`disp`, `display`	Display Composite
`end`	Indicate last Composite index
`isempty`	Determine whether Composite is empty
`length`	Length of Composite
`ndims`	Number of Composite dimensions
`numel`	Number of elements in Composite
`size`	Composite dimensions

Introduced in R2008b

CUDAKernel

Kernel executable on GPU

Constructor

`parallel.gpu.CUDAKernel`

Description

A `CUDAKernel` object represents a CUDA kernel, that can execute on a GPU. You create the kernel when you compile PTX or CU code, as described in "Run CUDA or PTX Code on GPU" on page 8-20.

Methods

existsOnGPU	Determine if gpuArray or CUDAKernel is available on GPU
feval	Evaluate kernel on GPU
setConstantMemory	Set some constant memory on GPU

Properties

A `CUDAKernel` object has the following properties:

Property Name	Description
`ThreadBlockSize`	Size of block of threads on the kernel. This can be an integer vector of length 1, 2, or 3 (since thread blocks can be up to 3-dimensional). The product of the elements of `ThreadBlockSize` must not exceed the `MaxThreadsPerBlock` for this kernel, and no element of `ThreadBlockSize` can exceed the corresponding element of the `GPUDevice` property `MaxThreadBlockSize`.
`MaxThreadsPerBlock`	Maximum number of threads permissible in a single block for this CUDA kernel. The product of the elements of `ThreadBlockSize` must not exceed this value.
`GridSize`	Size of grid (effectively the number of thread blocks that will be launched independently by the GPU). This is an integer vector of length 3. None of the elements of this vector can exceed the corresponding element in the vector of the `MaxGridSize` property of the `GPUDevice` object.
`SharedMemorySize`	The amount of dynamic shared memory (in bytes) that each thread block can use. Each thread block has an available shared memory region. The size of this region is limited in current cards to ~16 kB, and is shared with registers on the multiprocessors. As with all memory, this needs to be allocated before the kernel is launched. It is also common for the size of this shared memory region to be tied to the size of the thread block. Setting this value on the kernel ensures that each thread in a block can access this available shared memory region.

Property Name	Description
EntryPoint	(read-only) A character vector containing the actual entry point name in the PTX code that this kernel is going to call. An example might look like `'_Z13returnPointerPKfPy'`.
MaxNumLHSArguments	(read-only) The maximum number of left hand side arguments that this kernel supports. It cannot be greater than the number of right hand side arguments, and if any inputs are constant or scalar it will be less.
NumRHSArguments	(read-only) The required number of right hand side arguments needed to call this kernel. All inputs need to define either the scalar value of an input, the elements for a vector input/output, or the size of an output argument.
ArgumentTypes	(read-only) Cell array of character vectors, the same length as `NumRHSArguments`. Each of the character vectors indicates what the expected MATLAB type for that input is (a numeric type such as `uint8`, `single`, or `double` followed by the word `scalar` or `vector` to indicate if we are passing by reference or value). In addition, if that argument is only an input to the kernel, it is prefixed by `in`; and if it is an input/output, it is prefixed by `inout`. This allows you to decide how to efficiently call the kernel with both MATLAB arrays and gpuArray, and to see which of the kernel inputs are being treated as outputs.

See Also

gpuArray, GPUDevice

Introduced in R2011b

distributed

Access elements of distributed arrays from client

Constructor

`distributed`

You can also create a distributed array explicitly from the client with any of several MATLAB functions, shown in the table.

`eye(___,'distributed')`	`true(___,'distributed')`
`false(___,'distributed')`	`zeros(___,'distributed')`
`Inf(___,'distributed')`	`distributed.cell`
`NaN(___,'distributed')`	`distributed.spalloc`
`ones(___,'distributed')`	`distributed.speye`
`rand(___,'distributed')`	`distributed.sprand`
`randi(___,'distributed')`	`distributed.sprandn`
`randn(___,'distributed')`	

Description

Distributed arrays represent those arrays which are partitioned out among the workers in a parallel pool. A distributed array resembles a normal MATLAB array in the way you index and manipulate its elements, but none of its elements exists on the client.

Codistributed arrays that you create inside `spmd` statements are accessible as distributed arrays from the client.

Use the `gather` function to retrieve distributed arrays into the client work space.

Methods

classUnderlying	(Not recommended) Class of elements within gpuArray or distributed array
distributed.cell	Create distributed cell array
distributed.spalloc	Allocate space for sparse distributed matrix
distributed.speye	Create distributed sparse identity matrix
distributed.sprand	Create distributed sparse array of uniformly distributed pseudo-random values
distributed.sprandn	Create distributed sparse array of normally distributed pseudo-random values
eye	Identity matrix
false	Array of logical 0 (false)
gather	Transfer distributed array or gpuArray to local workspace
Inf	Array of infinity
isaUnderlying	(Not recommended) True if distributed array's underlying elements are of specified class
isdistributed	True for distributed array
rand	Array of rand values
sparse	Create sparse distributed or codistributed matrix
true	Array of logical 1 (true)
write	Write distributed data to an output location

The methods for distributed arrays are too numerous to list here. Most resemble and behave the same as built-in MATLAB functions. See "Run MATLAB Functions with Distributed Arrays" on page 4-18.

Also among the methods are several for examining the characteristics of the array itself. Most behave like the MATLAB functions of the same name:

Function	Description
isdistributed	Indication if array is distributed
isreal	Indication if array elements are real
isUnderlyingType	Determine if underlying data in the array is of specified type
length	Length of vector or largest array dimension
ndims	Number of dimensions in the array
size	Size of array dimensions
underlyingType	Class (data type) of the underlying data in the array

See Also
codistributed | gather | parpool | spmd

Topics
"Create and Use Distributed Arrays" on page 1-4
"Run MATLAB Functions with Distributed Arrays" on page 4-18
"Nondistributed Versus Distributed Arrays" on page 4-2

Introduced in R2008a

Future

Request function execution on parallel pool workers or MATLAB client

Description

A `Future` object represents a function to be executed on parallel pool workers or the MATLAB client.

Creation

There are several ways to create a `Future` object:

- Specify a function to be executed on a worker in a parallel pool using `parfeval`. The `parfeval` function creates a `FevalFuture` object to represent the function execution and hold the results. To create multiple `FevalFutures`, call `parfeval` multiple times; for example, you can create a vector of `FevalFutures` in a `for`-loop.
- Specify a function to be executed on every worker in a parallel pool using `parfevalOnAll`. The `parfevalOnAll` function creates a `FevalOnAllFuture` object to represent the function execution and hold the results.
- Specify a function to be executed on the MATLAB client after all `Future` objects complete using `afterAll`. The `afterAll` function creates an `AfterAllFuture` object to represent the function execution and hold the results.
- Specify a function to be executed on the MATLAB client after each `Future` object complete using `afterEach`. The `afterEach` function creates an `AfterEachFuture` object to represent the function execution and hold the results.

In summary, the following table describes the available types of future objects:

Future Object	Description
`FevalFuture`	Single `parfeval` Future instance
`FevalOnAllFuture`	`parfevalOnAll` Future instance
`AfterAllFuture`	`afterAll` Future instance
`AfterEachFuture`	`afterEach` Future instance

`Future` objects are local objects and can be accessed only in the MATLAB session that created it.

Properties

General Options

`CreateDateTime` — Date and time when this future was created
`datetime`

This property is read-only.

Date and time when this future was created, specified as a `datetime` object.

Data Types: `datetime`

`Error` — Error information
exception

This property is read-only.

Error information, specified as an exception. If the `Future` completes with no error, then this field is empty.

`FinishDateTime` — Date and time when this future finished running
datetime

This property is read-only.

Date and time when this future finished running, specified as a `datetime` object.

Data Types: `datetime`

`Function` — Function to evaluate
function_handle

This property is read-only.

Function to evaluate, specified as a function handle.

Example: `@rand`

Data Types: `function_handle`

`ID` — Numeric identifier for the future
integer

This property is read-only.

Numeric identifier for the future, specified as an integer.

Data Types: `double`

`InputArguments` — Input arguments to function
cell array

This property is read-only.

Input arguments to the function to execute, specified as a cell array.

Example: `{[1]}`

Example: `{[1,2], [2,1]}`

Data Types: `cell`

`NumOutputArguments` — Number of arguments returned by function
integer

This property is read-only.

Number of arguments returned by the function to execute, specified as an integer.

Data Types: `double`

`OutputArguments` — Output arguments from running function
cell array

This property is read-only.

Output arguments, specified as a cell array of results from the running function after it finishes execution. If the `Future` completes with errors, this field is empty. To see the error, check the `Error` property.

Example: `{[3.14]}`

Data Types: `cell`

`StartDateTime` — Date and time when this future started running
datetime

This property is read-only.

Date and time when this future started running, specified as a `datetime` object.

Data Types: `datetime`

`State` — Current state of future
`'pending'` | `'queued'` | `'running'` | `'finished'` | `'failed'` | `'unavailable'`

This property is read-only.

Current state of the future, specified as one of these values: `'pending'`, `'queued'`, `'running'`, `'finished'`, `'failed'`, or `'unavailable'`.

Data Types: `char`

FevalFuture Options

`Diary` — Text produced by execution of function
character array

This property is read-only.

Text produced by execution of function, specified as a `char` array.

Data Types: `char`

`Parent` — Queue of Future objects that contains the future
FevalQueue

This property is read-only.

Queue of `Future` objects that contains this `Future`, specified as a `FevalQueue`. Check this queue to identify the number of `Future` objects running or queued.

Data Types: `FevalQueue`

`Read` — Indication if outputs have been read
logical

This property is read-only.

Indication if outputs have been read by `fetchNext` or `fetchOutputs`, specified as a logical.

Data Types: `logical`

FevalOnAllFuture Options

`Diary` — Text produced by execution of function
cell array of character arrays

This property is read-only.

Text produced by execution of the function, specified as a cell array of character arrays that contains the text for each worker.

Data Types: `cell`

`Parent` — Queue of Future objects that contains the future
`FevalQueue`

This property is read-only.

Queue of `Future` objects that contains this `Future`, specified as a `FevalQueue`. Check this queue to identify the number of `Future` objects running or queued.

Data Types: `FevalQueue`

Object Functions

General Functions
afterAll	Specify a function to invoke after all parallel.Futures complete
afterEach	Specify a function to invoke after each parallel.Future completes
cancel	Cancel queued or running future
fetchOutputs	Retrieve all output arguments from Future
isequal	True if futures have same ID
wait	Wait for futures to complete

FevalFuture Only
fetchNext	Retrieve next available unread FevalFuture outputs

Examples

Combine `afterEach` and `afterAll`

You can combine `afterEach` and `afterAll` to automatically invoke more functions on the results of futures. Both `afterEach` and `afterAll` generate future variables that can be used again in `afterEach` and `afterAll`.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx= 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 8 workers.
```

Compute the largest element in each of those vectors when they become ready. `afterEach` executes the function handle on the output of each future when they become ready and creates another future to hold the results.

```
maxFuture = afterEach(f, @(r) max(r), 1);
```

To compute the minimum value among them, call `afterAll` on this new future. `afterAll` executes a function on the combined output arguments of all the futures after they all complete. In this case, `afterAll` executes the function `min` on the outputs of `maxFuture` after completing and creates another future to hold the result.

```
minFuture = afterAll(maxFuture, @(r) min(r), 1);
```

You can fetch the result using `fetchOutputs`. `fetchOutput` waits until the future completes to gather the results.

```
fetchOutputs(minFuture)
```

```
ans = 0.9973
```

You can check the result of `afterEach` by calling `fetchOutputs` on its future variable.

```
fetchOutputs(maxFuture)
```

```
ans = 10×1

    0.9996
    0.9989
    0.9994
    0.9973
    1.0000
    1.0000
    0.9989
    0.9994
    0.9998
    0.9999
```

See Also
`afterAll` | `afterEach` | `parallel.Pool` | `parfeval` | `parfevalOnAll`

Introduced in R2013b

gpuArray

Array stored on GPU

Description

A `gpuArray` object represents an array stored on the GPU. To work with `gpuArray` objects, use any GPU-enabled MATLAB function. You can use the array for direct calculations or in CUDA kernels that execute on the GPU. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

If you want to retrieve the array from the GPU, for example when using a function that does not support `gpuArray` objects, use the `gather` function.

Note You can load MAT files containing gpuArray data as in-memory arrays when a GPU is not available. A gpuArray loaded without a GPU is limited and you cannot use it for computations. To use a gpuArray loaded without a GPU, retrieve the contents using `gather`.

Creation

Use `gpuArray` to convert an array in the MATLAB workspace into a `gpuArray` object. Many MATLAB functions also allow you to create `gpuArray` objects directly. For more information, see "Establish Arrays on a GPU" on page 8-3.

Syntax

G = gpuArray(X)

Description

`G = gpuArray(X)` copies the array X to the GPU and returns a `gpuArray` object.

Input Arguments

X — Array
numeric array | logical array

Array to transfer to the GPU, specified as a numeric or logical array. The GPU device must have sufficient free memory to store the data. If X is already a `gpuArray` object, `gpuArray` outputs X unchanged.

You can also transfer sparse arrays to the GPU. `gpuArray` supports only sparse arrays of double-precision.

Example: G = gpuArray(magic(3));

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Object Functions

arrayfun   Apply function to each element of array on GPU
gather   Transfer distributed array or gpuArray to local workspace
pagefun   Apply function to each page of distributed array or gpuArray

There are several methods for examining the characteristics of a `gpuArray` object. Most behave like the MATLAB functions of the same name.

existsOnGPU   Determine if gpuArray or CUDAKernel is available on GPU
isequal   Determine array equality
isnumeric   Determine whether input is numeric array
issparse   Determine whether input is sparse
isUnderlyingType   Determine whether input has specified underlying data type
length   Length of largest array dimension
ndims   Number of array dimensions
size   Array size
underlyingType   Type of underlying data determining array behavior

Other methods for `gpuArray` objects are too numerous to list here. Most resemble and behave the same as the MATLAB functions of the same name. See "Run MATLAB Functions on a GPU" on page 8-9.

Examples

Use MATLAB Functions with a GPU

This example shows how to use GPU-enabled MATLAB functions to operate with gpuArrays. You can check the properties of your GPU using the `gpuDevice` function.

```
gpuDevice

ans =
  CUDADevice with properties:

                      Name: 'GeForce GTX 1080'
                     Index: 1
         ComputeCapability: '6.1'
             SupportsDouble: 1
             DriverVersion: 10.1000
            ToolkitVersion: 10.1000
         MaxThreadsPerBlock: 1024
           MaxShmemPerBlock: 49152
          MaxThreadBlockSize: [1024 1024 64]
                MaxGridSize: [2.1475e+09 65535 65535]
                  SIMDWidth: 32
                TotalMemory: 8.5899e+09
            AvailableMemory: 6.9012e+09
        MultiprocessorCount: 20
               ClockRateKHz: 1733500
                ComputeMode: 'Default'
         GPUOverlapsTransfers: 1
     KernelExecutionTimeout: 1
           CanMapHostMemory: 1
```

```
       DeviceSupported: 1
        DeviceSelected: 1
```

Create a row vector that repeats values from -15 to 15. To transfer it to the GPU and create a gpuArray, use the `gpuArray` function.

```
X = [-15:15 0 -15:15 0 -15:15];
gpuX = gpuArray(X);
whos gpuX
```

```
  Name       Size            Bytes  Class       Attributes

  gpuX       1x95                4  gpuArray
```

To operate with gpuArrays, use any GPU-enabled MATLAB function. MATLAB automatically runs calculations on the GPU. For more information, see "Run MATLAB Functions on a GPU" on page 8-9. For example, use a combination of `diag`, `expm`, `mod`, `round`, `abs`, and `fliplr`.

```
gpuE = expm(diag(gpuX,-1)) * expm(diag(gpuX,1));
gpuM = mod(round(abs(gpuE)),2);
gpuF = gpuM + fliplr(gpuM);
```

Plot the results.

```
imagesc(gpuF);
colormap(flip(gray));
```

If you need to transfer the data back from the GPU, use `gather`. Gathering back to the CPU can be costly, and is generally not necessary unless you need to use your result with functions that do not support gpuArray.

```
result = gather(gpuF);
whos result
```

```
  Name        Size              Bytes  Class     Attributes

  result      96x96             73728  double
```

In general there can be differences in the results if you run the code on the CPU, due to numerical precision and algorithmic differences between GPU and CPU. Answers on CPU and GPU are both equally valid floating point approximations to the true analytical result, having been subjected to different roundoff during computation. In this example, the results are integers and `round` eliminates the roundoff errors.

Perform a Monte Carlo Integration Using GPU-Enabled Functions

This example shows how to use MATLAB functions and operators with gpuArrays to compute the integral of a function, using the Monte Carlo integration method.

Define the number of points to sample. Sample points in the domain of the function, the interval `[-1,1]` in both x and y coordinates, by creating random points with the `rand` function. To create a random array directly on the GPU, use the `rand` function and specify `'gpuArray'`. For more information, see "Establish Arrays on a GPU" on page 8-3.

```
n = 1e6;
x = 2*rand(n,1,'gpuArray')-1;
y = 2*rand(n,1,'gpuArray')-1;
```

Define the function to integrate, and use the Monte Carlo integration formula on it. This function approximates the value of π by sampling points within the unit circle. Because the code uses GPU-enabled functions and operators on gpuArrays, the computations automatically run on the GPU. You can perform binary operations such as element-wise multiplication using the same syntax as MATLAB arrays use. To learn more about GPU-enabled functions, see "Run MATLAB Functions on a GPU" on page 8-9.

```
f = x.^2 + y.^2 <= 1;
result = 4*1/n*f'*ones(n,1,'gpuArray')
```

```
result =

    3.1403
```

Compute the Mandelbrot Set using GPU-Enabled Functions

This example shows how to use GPU-enabled MATLAB functions to compute a well-known mathematical construction: the Mandelbrot set. Check your GPU using the `gpuDevice` function.

Define the parameters. The Mandelbrot algorithm iterates over a grid of real and imaginary parts. The following code defines the number of iterations, grid size, and grid limits.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862,  0.123640851045266];
```

You can use the `gpuArray` function to transfer data to the GPU and create a `gpuArray`, or you can create an array directly on the GPU. `gpuArray` provides GPU versions of many functions that you can use to create data arrays, such as `linspace`. For more information, see "Create GPU Arrays Directly" on page 8-4.

```
x = gpuArray.linspace(xlim(1),xlim(2),gridSize);
y = gpuArray.linspace(ylim(1),ylim(2),gridSize);
whos x y
```

```
  Name      Size              Bytes  Class       Attributes

  x         1x1000                4  gpuArray
  y         1x1000                4  gpuArray
```

Many MATLAB functions support gpuArrays. When you supply a gpuArray argument to any GPU-enabled function, the function runs automatically on the GPU. For more information, see "Run MATLAB Functions on a GPU" on page 8-9. Create a complex grid for the algorithm, and create the array `count` for the results. To create this array directly on the GPU, use the `ones` function, and specify `'gpuArray'`.

```
[xGrid,yGrid] = meshgrid(x,y);
z0 = complex(xGrid,yGrid);
count = ones(size(z0),'gpuArray');
```

The following code implements the Mandelbrot algorithm using GPU-enabled functions. Because the code uses gpuArrays, the calculations happen on the GPU.

```
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end
count = log(count);
```

When computations are done, plot the results.

```
imagesc(x,y,count)
colormap([jet();flipud(jet());0 0 0]);
axis off
```

Tips

- If you need increased performance, or if a function is not available for GPU, `gpuArray` supports the following options:

  - To precompile and run purely element-wise code on `gpuArray` objects, use the `arrayfun` function.
  - To run C++ code containing CUDA device code or library calls, use a MEX-function. For more information, see "Run MEX-Functions Containing CUDA Code" on page 8-28.
  - To run existing GPU kernels written in CUDA C++, use the MATLAB CUDAKernel interface. For more information, see "Run CUDA or PTX Code on GPU" on page 8-20.
  - To generate CUDA code from MATLAB code, use GPU Coder™. For more information, see "Get Started with GPU Coder" (GPU Coder).

- You can control the random number stream on the GPU using `gpurng`.
- None of the following can exceed `intmax('int32')`:

  - The number of elements of a dense array.
  - The number of nonzero elements of a sparse array.
  - The size in any given dimension. For example, `zeros(0,3e9,'gpuArray')` is not allowed.

Alternatives

You can also create a `gpuArray` object using some MATLAB functions by specifying a `gpuArray` output. The following table lists the available MATLAB functions that can create `gpuArray` objects directly.

`eye(___,'gpuArray')`	`rand(___,'gpuArray')`
`false(___,'gpuArray')`	`randi(___,'gpuArray')`
`Inf(___,'gpuArray')`	`randn(___,'gpuArray')`
`NaN(___,'gpuArray')`	`gpuArray.colon`
`ones(___,'gpuArray')`	`gpuArray.freqspace`
`true(___,'gpuArray')`	`gpuArray.linspace`
`zeros(___,'gpuArray')`	`gpuArray.logspace`
	`gpuArray.speye`

For class-specific help on the functions with the `gpuArray` prefix, type

`help gpuArray.`*`functionname`*

where *`functionname`* is the name of the method. For example, to get help on `colon`, type

`help gpuArray.colon`

See Also
`arrayfun` | `existsOnGPU` | `gather` | `gpuDevice` | `gputimeit` | `pagefun` | `reset`

Topics
"Establish Arrays on a GPU" on page 8-3
"Run MATLAB Functions on a GPU" on page 8-9
"Identify and Select a GPU Device" on page 8-19

Introduced in R2010b

gpuDevice

Query or select a GPU device

Description

A `GPUDevice` object represents a graphic processing unit (GPU) in your computer. You can use the GPU to execute CUDA kernels or MATLAB code.

You can use a `GPUDevice` object to inspect the properties of your GPU device, reset the GPU device, or wait for your GPU to finish executing a computation. To obtain a `GPUDevice` object, use the `gpuDevice` function. You can also select or deselect your GPU device using the `gpuDevice` function. If you have access to multiple GPUs, use the `gpuDevice` function to choose a specific GPU device on which to execute your code.

You do not need to use a `GPUDevice` object to run functions on a GPU. For more information on how to use GPU-enabled functions, see "Run MATLAB Functions on a GPU" on page 8-9.

Creation

Syntax

```
gpuDevice
D = gpuDevice
D = gpuDevice(IDX)
gpuDevice([])
```

Description

`gpuDevice` displays the properties of the currently selected GPU device. If there is no currently selected device, `gpuDevice` selects the default device without clearing it. Use this syntax when you want to inspect the properties of your GPU device.

`D = gpuDevice` returns a `GPUDevice` object representing the currently selected device. If there is no currently selected device, `gpuDevice` selects the default device and returns a `GPUDevice` object representing that device without clearing it.

`D = gpuDevice(IDX)` selects the GPU device specified by index `IDX`. If the specified GPU device is not supported, an error occurs. This syntax resets the specified device and clears its memory, even if the device is already currently selected (equivalent to the `reset` function). All workspace variables representing `gpuArray` or `CUDAKernel` variables are now invalid and must be cleared from the workspace or redefined.

`gpuDevice([])`, with an empty argument (as opposed to no argument), deselects the GPU device and clears its memory of `gpuArray` and `CUDAKernel` variables. This syntax leaves no GPU device selected as the current device.

Input Arguments

IDX — Index of the GPU device
integer

Index of the GPU device, specified as an integer in the range 1 to `gpuDeviceCount`.

Example: `gpuDevice(1);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Properties

Name — Name of the GPU device
character array

Name of the GPU device, specified as a character array. The name assigned to the device is derived from the GPU device model.

Index — Index of the GPU device
integer

Index of the GPU device, specified as an integer in the range 1 to `gpuDeviceCount`. Use this index to select a particular GPU device.

ComputeCapability — Computational capability of the GPU device
character array

Computational capability of the GPU device, specified as a character array. To use the selected GPU device in MATLAB, `ComputeCapability` must meet the required specification in "GPU Support by Release" on page 8-38.

SupportsDouble — Support for double precision
0 | 1

Support for double precision operations, specified as the logical values 0 for `false` or 1 for `true`.

DriverVersion — Driver version
scalar

GPU device driver version currently in use, specified as a scalar value. To use the selected GPU device in MATLAB, `DriverVersion` must meet the required specification in "GPU Support by Release" on page 8-38.

ToolkitVersion — CUDA toolkit version
scalar

CUDA toolkit version used by the current release of MATLAB, specified as a scalar value.

MaxThreadsPerBlock — Maximum supported number of threads per block
scalar

Maximum supported number of threads per block during CUDAKernel execution, specified as a scalar value.

Example: 1024

MaxShmemPerBlock — Maximum supported amount of shared memory
scalar

Maximum supported amount of shared memory that a thread block can use during CUDAKernel execution, specified as a scalar value.

Example: 49152

MaxThreadBlockSize — Maximum size in each dimension for thread block
vector

Maximum size in each dimension for thread block, specified as a vector. Each dimension of a thread block must not exceed these dimensions. Also, the product of the thread block size must not exceed `MaxThreadsPerBlock`.

MaxGridSize — Maximum size of grid of thread blocks
vector

Maximum size of grid of thread blocks, specified as a vector.

SIMDWidth — Number of simultaneously executing threads
scalar

Number of simultaneously executing threads, specified as a scalar value.

TotalMemory — Total memory
scalar

Total memory (in bytes) on the device, specified as a scalar value.

AvailableMemory — Total memory available for data
scalar

Total memory (in bytes) available for data, specified as a scalar value. This property is available only for the currently selected device. This value can differ from the value reported by the NVIDIA System Management Interface due to memory caching.

MultiprocessorCount — Number of streaming multiprocessors
scalar

The number of streaming multiprocessors present on the device, specified as a scalar value.

ClockRateKHz — Peak clock rate
scalar

Peak clock rate of the GPU in kHz, specified as a scalar value.

ComputeMode — Compute mode
character array

The compute mode of the device, specified as one of the following values.

'Default'	The device is not restricted, and multiple applications can use it simultaneously. MATLAB can share the device with other applications, including other MATLAB sessions or workers.
'Exclusive thread' or 'Exclusive process'	Only one application at a time can use the device. While the device is selected in MATLAB, other applications cannot use it, including other MATLAB sessions or workers.
'Prohibited'	The device cannot be used.

GPUOverlapsTransfers — Support for overlapped transfers
0 | 1

Support for overlapped transfers, specified as the logical values 0 or 1.

KernelExecutionTimeout — Timeout for long-running kernels
0 | 1

Timeout for long-running kernels, specified as the logical values 0 or 1. If 1, the operating system places an upper bound on the time allowed for the CUDA kernel to execute. After this time, the CUDA driver times out the kernel and returns an error.

CanMapHostMemory — Support for mapping host memory
0 | 1

Support for mapping host memory into the CUDA address space, specified as the logical values 0 or 1.

DeviceSupported — Supported device
0 | 1

Supported device, specified by the logical values 0 or 1. Not all devices are supported; for example, devices with insufficient ComputeCapability.

DeviceSelected — Currently selected device
0 | 1

Currently selected device, specified by the logical values 0 or 1.

Object Functions

You can identify, select, reset, or wait for a GPU device using the following functions:

gpuDeviceCount	Number of GPU devices present
reset	Reset GPU device and clear its memory
wait (GPUDevice)	Wait for GPU calculation to complete

The following functions are also available:

parallel.gpu.GPUDevice.isAvailable(idx)	Returns true if the GPU specified by index idx is supported and capable of being selected. idx can be an integer or a vector of integers; the default index is the current device.
parallel.gpu.GPUDevice.getDevice(idx)	Returns a GPUDevice object without selecting it.

For a complete list of functions, use the methods function on the GPUDevice object:

methods('parallel.gpu.GPUDevice')

You can get help on any of the object functions with the following command:

help parallel.gpu.GPUDevice.*functionname*

where *functionname* is the name of the function. For example, to get help on isAvailable, type:

help parallel.gpu.GPUDevice.isAvailable

Examples

Identify and Select a GPU

To determine how many GPU devices are available in your computer, use the gpuDeviceCount function.

gpuDeviceCount

    2

When there are multiple devices, the first is the default. You can examine its properties with the gpuDevice function to determine if that is the one you want to use.

d = gpuDevice

d =

  CUDADevice with properties:

                      Name: 'GeForce GTX 1080'
                     Index: 1
         ComputeCapability: '6.1'
             SupportsDouble: 1
             DriverVersion: 10
            ToolkitVersion: 10
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
              MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 8.5899e+09
           AvailableMemory: 7.0053e+09
        MultiprocessorCount: 20
               ClockRateKHz: 1733500
               ComputeMode: 'Default'
        GPUOverlapsTransfers: 1

```
    KernelExecutionTimeout: 1
         CanMapHostMemory: 1
          DeviceSupported: 1
           DeviceSelected: 1
```

If `d` is the device you want to use, you can proceed. To run computations on the GPU, use `gpuArray` enabled functions. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

To use another device, call `gpuDevice` with the index of the other device.

```
gpuDevice(2)
```

Query Compute Capabilities

Create an object representing the default GPU device.

```
g = gpuDevice;
```

Query the compute capabilities of all available GPU devices.

```
for ii = 1:gpuDeviceCount
    g = gpuDevice(ii);
    fprintf(1,'Device %i has ComputeCapability %s \n', ...
            g.Index,g.ComputeCapability)
end
```

```
Device 1 has ComputeCapability 3.5
Device 2 has ComputeCapability 2.0
```

Use Multiple GPUs in a Parallel Pool

If you have access to several GPUs, you can perform your calculations on multiple GPUs in parallel using a parallel pool.

Start a parallel pool with as many workers as GPUs. To determine the number of GPUs available, use the `gpuDeviceCount` function. By default, MATLAB assigns a different GPU to each worker for best performance.

```
parpool('local',gpuDeviceCount);
```

To identify which GPU each worker is using, call `gpuDevice` inside an `spmd` block. The `spmd` block runs `gpuDevice` on every worker.

```
spmd
    gpuDevice
end
```

Use parallel language features, such as `parfor` or `parfeval`, to distribute your computations to workers in the parallel pool. If you use `gpuArray` enabled functions in your computations, these functions run on the GPU of the worker. For more information, see "Run MATLAB Functions on a GPU" on page 8-9. For an example, see "Run MATLAB Functions on Multiple GPUs" on page 9-82.

When you are done with your computations, shut down the parallel pool. You can use the `gcp` function to obtain the current parallel pool.

```
delete(gcp('nocreate'));
```

If you want to use a different choice of GPUs, then can use `gpuDevice` to select a particular GPU on each worker. Define an array, for example `gpuIndices`, that contains the indices of the GPUs to activate on each worker. Then, start a parallel pool with as many workers as GPUs to select, and use an `spmd` block to run `gpuDevice` on each worker. The `labindex` function identifies each worker. Use this function to associate a worker with a GPU index.

```
gpuIndices = [1 3];
parpool(numel(gpuIndices));
spmd
    gpuDevice(gpuIndices(labindex));
end
```

As a best practice, and for best performance, assign a different GPU to each worker.

See Also

GPUDeviceManager | arrayfun | gpuArray | gpuDeviceCount | reset | wait (GPUDevice)

Topics
"Identify and Select a GPU Device" on page 8-19
"Run MATLAB Functions on a GPU" on page 8-9
"Run MATLAB Functions on Multiple GPUs" on page 9-82
"GPU Support by Release" on page 8-38

External Websites
Deep Learning with GPUs and MATLAB

Introduced in R2010b

GPUDeviceManager

Manager for GPU Devices

Constructor

parallel.gpu.GPUDeviceManager.instance

Description

`parallel.gpu.GPUDeviceManager` provides events that indicate when a GPU device has been selected or deselected. These events also fire when a GPU device is reset. There is only a single instance of the `parallel.gpu.GPUDeviceManager` available in a given MATLAB session, and it is obtained using the method `parallel.gpu.GPUDeviceManager.instance`.

Events

Events of the class include the following:

Event Name	Description
DeviceSelected	Fired after a GPU device is selected.
DeviceDeselecting	Fired just before a GPU device is deselected.

Properties

A `GPUDeviceManager` object has one property:

Property Name	Description
SelectedDevice	Contains the currently selected GPU Device.

Methods

Methods of the class include the following:

Method Name	Description
getDeviceCount	Returns the number of GPU devices available.
selectDevice	Selects a GPU device.

See Also

gpuDevice, gpuDeviceCount, gpuArray

Introduced in R2016a

mxGPUArray

Type for MATLAB gpuArray

Description

mxGPUArray is an opaque C language type that allows a MEX function access to the elements in a MATLAB gpuArray. Using the mxGPU API, you can perform calculations on a MATLAB gpuArray, and return gpuArray results to MATLAB.

All MEX functions receive inputs and pass outputs as mxArrays. A gpuArray in MATLAB is a special kind of mxArray that represents an array stored on the GPU. In your MEX function, you use mxGPUArray objects to access an array stored on the GPU: these objects correspond to MATLAB gpuArrays.

The mxGPU API contains functions that manipulate mxGPUArray objects. These functions allow you to extract mxGPUArrays from input mxArrays, to wrap output mxGPUArrays as mxArrays for return to MATLAB, to determine the characteristics of the arrays, and to get pointers to the underlying elements. You can perform calculations by passing the pointers to CUDA functions that you write or that are available in external libraries.

The basic structure of a GPU MEX function is:

1 Call `mxInitGPU` to initialize MathWorks GPU library.
2 Determine which `mxArray` inputs contain GPU data.
3 Create `mxGPUArray` objects from the input `mxArray` arguments, and get pointers to the input elements on the device.
4 Create `mxGPUArray` objects to hold the outputs, and get the pointers to the output elements on the device.
5 Call a CUDA function, passing it the device pointers.
6 Wrap the output `mxGPUArray` as an `mxArray` for return to MATLAB.
7 Destroy the `mxGPUArray` objects you created.

The header file that contains this type is `mxGPUArray.h`. You include it with the line:

```
#include "gpu/mxGPUArray.h"
```

See Also

gpuArray, mxArray

Introduced in R2013a

parallel.Cluster

Access cluster properties and behaviors

Constructors

parcluster

getCurrentCluster (in the workspace of the MATLAB worker)

Container Hierarchy

Parent	None
Children	parallel.Job, parallel.Pool

Description

A parallel.Cluster object provides access to a cluster, which controls the job queue, and distributes tasks to workers for execution.

Types

The two categories of clusters are the MATLAB Job Scheduler and common job scheduler (CJS). The MATLAB Job Scheduler is available in the MATLAB Parallel Server. The CJS clusters encompass all other types, including the local, generic, and third-party schedulers.

Use MJSComputeCloud objects to interact with MATLAB Parallel Server for Amazon EC2® clusters.

The following table describes the available types of cluster objects.

Cluster Type	Description
parallel.cluster.MJS	Interact with MATLAB Job Scheduler cluster on-premises
parallel.cluster.MJSComputeCloud	Interact with MATLAB Parallel Server for Amazon EC2 cluster
parallel.cluster.Local	Interact with CJS cluster running locally on client machine
parallel.cluster.HPCServer	Interact with CJS cluster running Windows Microsoft HPC Server
parallel.cluster.LSF	Interact with CJS cluster running LSF
parallel.cluster.PBSPro	Interact with CJS cluster running Altair PBS Pro
parallel.cluster.Torque	Interact with CJS cluster running TORQUE
parallel.cluster.Slurm	Interact with CJS cluster running Slurm
parallel.cluster.Generic	Interact with CJS cluster using the generic interface

Methods

Common to All Cluster Types

batch	Run MATLAB script or function on worker
createCommunicatingJob	Create communicating job on cluster
createJob	Create independent job on cluster
findJob	Find job objects stored in cluster
isequal	True if clusters have same property values
parpool	Create parallel pool on cluster
saveAsProfile	Save cluster properties to specified profile
saveProfile	Save modified cluster properties to its current profile

MATLAB Job Scheduler

changePassword	Prompt user to change MATLAB Job Scheduler password
demote	Demote job in cluster queue
logout	Log out of MATLAB Job Scheduler cluster
pause	Pause MATLAB Job Scheduler queue
promote	Promote job in MATLAB Job Scheduler cluster queue
resume	Resume processing queue in MATLAB Job Scheduler

MJSComputeCloud

shutdown	Shut down cloud cluster
start	Start cloud cluster
wait (cluster)	Wait for cloud cluster to change state

HPC Server, PBS Pro, LSF, TORQUE, Slurm, and Local Clusters

getDebugLog	Read output messages from job run in CJS cluster

Generic

getDebugLog	Read output messages from job run in CJS cluster
getJobClusterData	Get specific user data for job on generic cluster
getJobFolder	Folder on client where jobs are stored
getJobFolderOnCluster	Folder on cluster where jobs are stored
getLogLocation	Log location for job or task
setJobClusterData	Set specific user data for job on generic cluster

Properties

Common to All Cluster Types

The following properties are common to all cluster object types.

Property	Description
ClusterMatlabRoot	Specifies path to MATLAB for workers to use
Host	Host name of the cluster head node
JobStorageLocation	Location where cluster stores job and task information
Jobs	List of jobs contained in this cluster
LicenseNumber	License number to use when running jobs with this cluster
Modified	True if any properties in this cluster have been modified
NumThreads	Number of computational threads for workers
NumWorkers	Number of workers available for this cluster
OperatingSystem	Operating system of nodes used by cluster
Profile	Profile used to build this cluster
RequiresOnlineLicensing	True if the cluster is using online licensing
Type	Type of this cluster
UserData	Information associated with cluster object within client session

Specify the `JobStorageLocation` property as one of the following.

- If you use the generic scheduler interface in remote mode to interact with a third-party scheduler, and the client and workers use different operating systems, specify a structure. The structure must have the fields `'windows'` and `'unix'`. The fields are the Windows and Unix path corresponding to the folder where the cluster stores job and task information. The following structure specifies the same folder using a Windows UNC path (`'\\organization\some\path'`) and a Unix path (`'/organization/some/path'`):

  ```
  struct('windows', '\\organization\some\path', 'unix', '/organization/some/path')
  ```
- Otherwise, use a character vector or string to specify the folder where the cluster stores job and task information.

MATLAB Job Scheduler

MATLAB Job Scheduler cluster objects have the following properties in addition to the common properties:

Property	Description
AllHostAddresses	IP addresses of the cluster host
BusyWorkers	Workers currently running tasks
IdleWorkers	Workers currently available for running tasks
HasSecureCommunication	True if cluster is using secure communication
Name	Name of this cluster
NumBusyWorkers	Number of workers currently running tasks

Property	Description
NumIdleWorkers	Number of workers available for running tasks
PromptForPassword	True if system should prompt for password when authenticating user
SecurityLevel	Degree of security applied to cluster and its jobs. For descriptions of security levels, see "Set MATLAB Job Scheduler Cluster Security" (MATLAB Parallel Server).
State	Current state of cluster
Username	User accessing cluster

MJSComputeCloud

MJSComputeCloud cluster objects have the following properties in addition to the common properties:

Property	Description
BusyWorkers	Workers currently running tasks
Certificate	Cluster SSL certificate
HasSecureCommunication	True if cluster is using secure communication
Identifier	Unique cluster identifier
IdleWorkers	Workers currently available for running tasks
MatlabVersion	Version of MATLAB running on the workers
MaxNumWorkers	Maximum number of workers this cluster can use. • When you use a cluster with automatic resizing, workers are added automatically up to this maximum value as necessary. For more information on automatic resizing, see Resize Clusters Automatically. • When you use a cluster without automatic resizing, this value is the number of workers when you started the cluster.
Name	Name of this cluster
NumBusyWorkers	Number of workers currently running tasks
NumIdleWorkers	Number of workers available for running tasks
NumWorkersRequested	Number of workers requested for this cluster. The cluster adds or removes workers as soon as possible to reach this number.
SharedState	The shared state of the cluster, which can be: • Personal – Only you can use this cluster, so long as you created it. • Shareable – Anyone can use this cluster.

Property	Description
ShutdownAt	Shutdown time or event
State	Current state of cluster
Username	User accessing cluster

Local

Local cluster objects have no editable properties beyond the properties common to all clusters.

HPC Server

HPC Server cluster objects have the following properties in addition to the common properties:

Property	Description
ClusterVersion	Version of Microsoft Windows HPC Server running on the cluster
HasSharedFilesystem	Specify whether client and cluster nodes share JobStorageLocation
JobDescriptionFile	Name of XML job description file to use when creating jobs
JobTemplate	Name of job template to use for jobs submitted to HPC Server
Name	Name of this cluster
UseSOAJobSubmission	Allow service-oriented architecture (SOA) submission on HPC Server

PBS Pro and TORQUE

PBS Pro and TORQUE cluster objects have the following properties in addition to the common properties:

Property	Description
CommunicatingJobWrapper	Script that cluster runs to start workers
RcpCommand	Command to copy files to and from client
ResourceTemplate	Specify qsub options to request resources during job submission
RshCommand	Remote execution command used on worker nodes during communicating job
HasSharedFilesystem	Specify whether client and cluster nodes share JobStorageLocation
SubmitArguments	Specify additional arguments to use when submitting jobs

LSF

LSF cluster objects have the following properties in addition to the common properties:

Property	Description
ClusterName	Name of LSF cluster
CommunicatingJobWrapper	Script that the cluster runs to start workers
HasSharedFilesystem	Specify whether client and cluster nodes share JobStorageLocation
ResourceTemplate	Specify bsub options to request resources during job submission
SubmitArguments	Specify additional arguments to use when submitting jobs

Slurm

Slurm cluster objects have the following properties in addition to the common properties:

Property	Description
ClusterName	Name of the Slurm cluster
CommunicatingJobWrapper	Script that the cluster runs to start workers
ResourceTemplate	Specify sbatch options to request resources during job submission
SubmitArguments	Specify additional arguments to use when submitting jobs

Generic

If you create a generic cluster object from an R2017a or later profile, you have the following properties in addition to the common properties:

Property	Description
AdditionalProperties	Additional properties for plugin scripts
HasSharedFilesystem	Specify whether client and cluster nodes share JobStorageLocation
PluginScriptsLocation	Folder containing scheduler plugin scripts

If you create a generic cluster object from an R2016b or earlier profile, you have the following properties in addition to the common properties:

Property	Description
CancelJobFcn	Function to run when cancelling job
CancelTaskFcn	Function to run when cancelling task
CommunicatingSubmitFcn	Function to run when submitting communicating job
DeleteJobFcn	Function to run when deleting job
DeleteTaskFcn	Function to run when deleting task
GetJobStateFcn	Function to run when querying job state

Property	Description
HasSharedFilesystem	Specify whether client and cluster nodes share JobStorageLocation
IndependentSubmitFcn	Function to run when submitting independent job

Help

For further help on cluster objects, including links to help for specific cluster types and object properties, type:

```
help parallel.Cluster
```

See Also

parallel.Job, parallel.Task, parallel.Worker, parallel.Pool, parallel.cluster.Hadoop

Introduced in R2012a

parallel.cluster.Hadoop

Hadoop cluster for mapreducer, mapreduce and tall arrays

Constructors

parallel.cluster.Hadoop

Description

A parallel.cluster.Hadoop object provides access to a cluster for configuring mapreducer, mapreduce, and tall arrays.

Properties

A parallel.cluster.Hadoop object has the following properties.

Property	Description
AdditionalPaths	Paths to be added to MATLAB command search path on workers
AttachedFiles	Files transferred to the workers during a mapreduce call
AutoAttachFiles	Specifies whether automatically attach files
ClusterMatlabRoot	Specifies path to MATLAB for workers to use
HadoopConfigurationFile	Application configuration file to be given to Hadoop
HadoopInstallFolder	Installation location of Hadoop on the local machine
HadoopProperties	Map of name-value property pairs to be given to Hadoop
LicenseNumber	License number to use with online licensing
RequiresOnlineLicensing	Specify whether cluster uses online licensing
SparkInstallFolder	Installation location of Spark on the local machine
SparkProperties	Map of name-value property pairs to be given to Spark

HadoopProperties allows you to override configuration properties for Hadoop. See the list of properties in the Hadoop documentation.

The SparkInstallFolder is by default set to the SPARK_HOME environment variable. This is required for tall array evaluation on Hadoop (but not for mapreduce). For a correctly configured cluster, you only need to set the installation folder.

SparkProperties allows you to override configuration properties for Spark. See the list of properties in the Spark documentation.

Help

For further help, type:

```
help parallel.cluster.Hadoop
```

Specify Memory Properties

Spark enabled Hadoop clusters place limits on how much memory is available. You must adjust these limits to support your workflow.

Size of Data to Gather

The amount of data gathered to the client is limited by the Spark properties:

- `spark.driver.memory`
- `spark.executor.memory`

The amount of data to gather from a single Spark task must fit in these properties. A single Spark task processes one block of data from HDFS, which is 128 MB of data by default. If you gather a tall array containing most of the original data, you must ensure these properties are set to fit.

If these properties are set too small, you see an error like the following.

```
Error using tall/gather (line 50)
Out of memory; unable to gather a partition of size 300m from Spark.
Adjust the values of the Spark properties spark.driver.memory and
spark.executor.memory to fit this partition.
```

The error message also specifies the property settings you need.

Adjust the properties either in the default settings of the cluster or directly in MATLAB. To adjust the properties in MATLAB, add name-value pairs to the `SparkProperties` property of the cluster. For example:

```
cluster = parallel.cluster.Hadoop;
cluster.SparkProperties('spark.driver.memory') = '2048m';
cluster.SparkProperties('spark.executor.memory') = '2048m';
mapreducer(cluster);
```

Specify Working Memory Size for a MATLAB Worker

The amount of working memory for a MATLAB Worker is limited by the Spark property:

- `spark.yarn.executor.memoryOverhead`

By default, this is set to 2.5 GB. You typically need to increase this if you use `arrayfun`, `cellfun`, or custom datastores to generate large amounts of data in one go. It is advisable to increase this if you come across lost or crashed Spark Executor processes.

You can adjust these properties either in the default settings of the cluster or directly in MATLAB. To adjust the properties in MATLAB, add name-value pairs to the SparkProperties property of the cluster. For example:

```
cluster = parallel.cluster.Hadoop;
cluster.SparkProperties('spark.yarn.executor.memoryOverhead') = '4096m';
mapreducer(cluster);
```

See Also

`parallel.Cluster`, `parallel.Pool`

See Also

Topics
"Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52
"Run mapreduce on a Hadoop Cluster" on page 5-58

Introduced in R2014b

parallel.gpu.RandStream

Random number stream on a GPU

Description

Use `parallel.gpu.RandStream` to control the global GPU random number stream and create multiple independent streams on the GPU. When you generate random numbers on a GPU, the numbers are drawn from the GPU random number stream. This stream is different from the random stream of the client MATLAB session on the CPU.

To create random numbers on the GPU, use the random number generator functions `rand`, `randi`, and `randn` with `gpuArrays`. By default, these functions draw numbers from the global GPU random number stream. To use a different stream, follow the syntaxes described in "Object Functions" on page 10-49. If you use a GPU random number stream, the results are returned as a `gpuArray`.

Creation

Use the following syntaxes to create a single `parallel.gpu.RandStream` object. If you want to create multiple independent streams simultaneously, use the `parallel.gpu.RandStream.create` function.

Syntax

```
s = parallel.gpu.RandStream(gentype)
s = parallel.gpu.RandStream(gentype,Name,Value)
```

Description

`s = parallel.gpu.RandStream(gentype)` creates a random number stream that uses the uniform pseudorandom number generator algorithm specified by `'gentype'`.

`s = parallel.gpu.RandStream(gentype,Name,Value)` also specifies one or more optional `Name,Value` pairs to control properties of the stream.

Input Arguments

`'gentype'` — Random number generator algorithm
`'Threefry'` or `'Threefry4x64_20'` | `'Philox'` or `'Philox4x32_10'` | `'CombRecursive'` or `'mrg32k3a'`

Random number generator algorithm, specified as one of the following three random number generator algorithms supported on the GPU.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period in Full Precision
`'Threefry'` or `'Threefry4x64_20'`	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period in Full Precision
`'Philox'` or `'Philox4x32_10'`	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
`'CombRecursive'` or `'mrg32k3a'`	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})

For more information on the differences between generating random numbers on the GPU and CPU, see "Random Number Streams on a GPU" on page 8-6.

This argument sets the `Type` property.

Properties

Type — Random number generator algorithm
`'Threefry4x64_20'` | `'Philox4x32_10'` | `'MRG32K3A'`

This property is read-only.

Generator algorithm used by the stream specified as `'Threefry4x64_20'`, `'Philox4x32_10'`, or `'mrg32k3a'`.

Set this property using the `gentype` argument when you create the stream.

Data Types: `char`

Seed — Random number seed
0 (default) | nonnegative integer | `'shuffle'`

This property is read-only.

Random number seed, specified as the comma-separated pair consisting of `'Seed'` and a nonnegative integer or as the string or character vector `'shuffle'`. The seed specifies the starting point for the algorithm to generate random numbers. Specify `'Seed'` as an integer when you want reproducible results. Specifying `'Seed'` as `'shuffle'` seeds the generator based on the current time.

Set this property as a name-value pair when you create the stream.

NumStreams — Number of Streams
positive integer

This property is read-only.

Number of streams in the group in which the current stream was created, specified as a positive integer. Create multiple streams at once using the function `parallel.gpu.RandStream.create`.

StreamIndex — Stream index
positive integer

Stream index of the current stream, specified as a positive integer. The stream index identified individual streams when you create multiple streams at once using the function `parallel.gpu.RandStream.create`.

State — Current state
vector

Current state of the random number stream, specified as a vector. The internal state determines the sequence of random numbers produced by the random number stream s. The size of this state vector depends on the generator chosen.

Saving and restoring the internal state of the generator with the `State` property allows you to reproduce a sequence of random numbers. When you set this property, the value, you assign to `s.State` must be a value read from `s.State` previously. Use `reset` to return a stream to a predictable state without having previously read from the `State` property.

NormalTransform — Normal transformation algorithm
`'BoxMuller'` | `'Inversion'`

Normal transformation algorithm, specified as `'BoxMuller` or `'Inversion'`.

The normal transformation algorithm specifies the algorithm to use when generating normally distributed random numbers generated using `randn`. The `'BoxMuller'` algorithm is supported for the `'Threefry` and `'Philox'` generators. The `'Inversion'` algorithm is supported for the `'CombRecursive'` generator. No other transformation algorithms are supported on the GPU.

Data Types: `char`

Antithetic — Antithetic values
`false` or `0` (default)

This property is read-only.

Antithetic values, specified as `false` or `0`. This property indicates whether S generates antithetic pseudorandom values, that is, the usual values subtracted from `1` for uniform values.

This property is always `0`. The stream does not generate antithetic values. You cannot modify this property.

Data Types: `logical`

FullPrecision — Full precision generation
`true` or `1` (default)

This property is read-only.

Full precision generation, specified as `true` or `1`. This property indicates whether the random number stream generates values using full precision. Two random numbers are consumed to ensure all bits of a double are set.

This property is always `1`. You cannot modify this property.

Data Types: `logical`

Object Functions

parallel.gpu.RandStream.create	Create independent random number streams on a GPU
parallel.gpu.RandStream.list	Random number generator algorithms on the GPU
parallel.gpu.RandStream.getGlobalStream	Current global GPU random number stream
parallel.gpu.RandStream.setGlobalStream	Set GPU global random number stream

reset (RandStream)	Reset random number stream

By default, when you create random numbers on the GPU using random number generation functions, such as `rand`, the random numbers are drawn from the global random number stream on the GPU. To specify a different stream, create a `parallel.gpu.RandStream` object and pass it as the first input argument. For instance, create a 4-by-1 vector of random numbers using the Philox generator algorithm.

```
s = parallel.gpu.RandStream('Philox');
r = rand(s,4,1);
```

These functions accept a `parallel.gpu.RandStream` object and generate random numbers on the GPU:

`rand`	Uniformly distributed random numbers	Supported syntaxes, where `s` is a `parallel.gpu.RandStream` object:
`randi`	Uniformly distributed pseudorandom integers	`X = rand(s)`
`randn`	Normally distributed random numbers	`X = rand(s,n)` `X = rand(s,sz1,...,szN)` `X = rand(s,sz)` `X = rand(s,typename)` For details on other input arguments, see `rand`, `randi`, and `randn`.
`randperm`	Random permutation of integers	Supported syntaxes, where `s` is a `parallel.gpu.RandStream` object: `p = randperm(s,n)` `p = randperm(s,n,k)` For details on other input arguments, see `randperm`.

Examples

Change the Global GPU Stream

You can change the global random number stream on the GPU. First, define the random number stream that you want to set as the new global stream.

```
newStr = parallel.gpu.RandStream('Philox')

newStr =

Philox4x32_10 random stream on the GPU
            Seed: 0
  NormalTransform: BoxMuller
```

Next, set this new stream to be the global stream.

```
parallel.gpu.RandStream.setGlobalStream(newStr);
```

Check that `newStr` is now the current global stream.

```
newStr
```

```
newStr =

Philox4x32_10 random stream on the GPU (current global stream)
             Seed: 0
  NormalTransform: BoxMuller
```

On a GPU, the functions `rand`, `randi`, and `randn` now draw random numbers from the new global stream using the `'Philox'` generator algorithm.

Match the GPU and CPU Random Number Streams

If you have applications that require generating the same random numbers on the GPU and the CPU, you can set the streams to match. Create matching streams on both the GPU and CPU, and set them as the global stream in each case.

```
stCPU = RandStream('Threefry','Seed',0,'NormalTransform','Inversion');
stGPU = parallel.gpu.RandStream('Threefry','Seed',0,'NormalTransform','Inversion');
```

Only the `'Inversion'` normal transformation algorithm is available on both the GPU and CPU.

Set these streams to be the global streams on the CPU and GPU, respectively.

```
RandStream.setGlobalStream(stCPU);
parallel.gpu.RandStream.setGlobalStream(stGPU);
```

Calling `rand` and `randn` now produces the same sets of numbers on both the GPU and the client MATLAB session.

```
rC = rand(1,10)
rG = rand(1,10, 'gpuArray')


rC =
    0.1726    0.9207    0.8108    0.7169    0.8697    0.7920    0.4159    0.6503    0.1025    0.6

rG =
    0.1726    0.9207    0.8108    0.7169    0.8697    0.7920    0.4159    0.6503    0.1025    0.6

rnC = randn(1,10)
rnG = randn(1,10, 'gpuArray')

rnC =
   -0.9438    1.4095    0.8807    0.5736    1.1250    0.8133   -0.2124    0.3862   -1.2673    0

rnG =
   -0.9438    1.4095    0.8807    0.5736    1.1250    0.8133   -0.2124    0.3862   -1.2673    0
```

See Also
`RandStream` | `gpurng`

Topics
"Random Number Streams on a GPU" on page 8-6

Introduced in R2011b

parallel.Job

Access job properties and behaviors

Constructors

`createCommunicatingJob`, `createJob`, `findJob`, `recreate`

`getCurrentJob` (in the workspace of the MATLAB worker)

Container Hierarchy

Parent	parallel.Cluster
Children	parallel.Task

Description

A `parallel.Job` object provides access to a job, which you create, define, and submit for execution.

Types

The following table describes the available types of job objects. The job type is determined by the type of cluster, and whether the tasks must communicate with each other during execution.

Job Type	Description
parallel.job.MJSIndependentJob	Job of independent tasks on MATLAB Job Scheduler cluster
parallel.job.MJSCommunicatingJob	Job of communicating tasks on MATLAB Job Scheduler cluster
parallel.job.CJSIndependentJob	Job of independent tasks on CJS cluster
parallel.job.CJSCommunicatingJob	Job of communicating tasks on CJS cluster

Methods

Common to All Job Types

The following methods are common to all job object types.

cancel	Cancel job or task
createTask	Create new task in job
delete	Remove job or task object from cluster and memory
diary	Display or save Command Window text of batch job
fetchOutputs	Retrieve output arguments from all tasks in job
findTask	Task objects belonging to job object
listAutoAttachedFiles	List of files automatically attached to job, task, or parallel pool
load	Load workspace variables from batch job
recreate	Create new job from existing job
submit	Queue job in scheduler
wait	Wait for job to change state

CJS Jobs

CJS job objects have the following methods in addition to the common methods:
getTaskSchedulerIDs   Scheduler IDs of tasks in job

Properties

Common to All Job Types

The following properties are common to all job object types.

Property	Description
AdditionalPaths	Folders to add to MATLAB search path of workers
AttachedFiles	Files and folders that are sent to workers
AutoAddClientPath	Specifies whether user-added-entries on the client's path are automatically added to each worker's path
AutoAttachFiles	Specifies if dependent code files are automatically sent to workers
CreateDateTime	Date and time when the job is created
EnvironmentVariables	Names of environment variables that are sent to the workers
FinishDateTime	Date and time when the job finishes running
ID	Job's numeric identifier
JobData	Information made available to all workers for job's tasks
Name	Name of job
Parent	Cluster object containing this job
StartDateTime	Date and time when the job starts running
State	State of job: `'pending'`, `'queued'`, `'running'`, `'finished'`, or `'failed'`

Property	Description
SubmitDateTime	Date and time when the job is submitted to the queue
Tag	Label associated with job
Tasks	Array of task objects contained in job
Type	Job type: `'independent'`, `'pool'`, or `'spmd'`
UserData	Information associated with job object
Username	Name of user who owns job

MATLAB Job Scheduler Jobs

MATLAB Job Scheduler independent job objects and MATLAB Job Scheduler communicating job objects have the following properties in addition to the common properties:

Property	Description
AuthorizedUsers	Users authorized to access job
FinishedFcn	Callback function executed on client when this job finishes
NumWorkersRange	Minimum and maximum limits for number of workers to run job
QueuedFcn	Callback function executed on client when this job is submitted to queue
RestartWorker	True if workers are restarted before evaluating first task for this job
RunningFcn	Callback function executed on client when this job starts running
Timeout	Time limit, in seconds, to complete job

CJS Jobs

CJS independent job objects do not have any properties beyond the properties common to all job types.

CJS communicating job objects have the following properties in addition to the common properties:

Property	Description
NumWorkersRange	Minimum and maximum limits for number of workers to run job

Help

To get further help on a particular type of parallel.Job object, including a list of links to help for its properties, type `help parallel.job.<job-type>`. For example:

```
help parallel.job.MJSIndependentJob
```

See Also

parallel.Cluster, parallel.Task, parallel.Worker

Introduced in R2012a

parallel.Pool

Parallel pool of workers

Description

Use `parpool` to create a parallel pool. After you create the pool, parallel pool features, such as `parfor` or `parfeval`, run on the workers. With the `parallel.Pool` object, you can interact with the parallel pool.

`parallel.Pool` is the base class for the following types of pools:

- `ProcessPool`
- `ThreadPool`
- `ClusterPool`

Creation

Create a parallel pool of workers by using the `parpool` function.

See Also
`Future` | `parallel.Cluster`

Topics
"Run Code on Parallel Pools" on page 2-55

Introduced in R2013b

parallel.pool.DataQueue

Class that enables sending and listening for data between client and workers

Description

A data queue enables sending data or messages from workers back to the client in a parallel pool while a computation is carried out. For example, you can get intermediate values and an indication of the progress of the computation.

To send data from a parallel pool worker back to the client, first construct a `DataQueue` in the client. Pass this `DataQueue` into a `parfor`-loop or other parallel language construct, such as `spmd`. From the workers, call `send` to send data back to the client. At the client, register a function to be called each time data is received by using `afterEach`.

- You can call `send` from the process that calls the constructor, if required.
- You can construct the queue on the workers and send it back to the client to enable communication in the reverse direction. However, you cannot send a queue from one worker to another. Use `spmd`, `labSend`, or `labReceive` instead.
- Unlike all other handle objects, `DataQueue` instances do remain connected when they are sent to workers.

Creation

Syntax

```
q = parallel.pool.DataQueue
```

Description

`q = parallel.pool.DataQueue` takes no arguments and returns an object that can be used to send or listen for messages (or data) from different workers. You call the constructor only in the process where you want to receive the data. In the usual workflow, the workers should not be calling the constructor, but should be handed an existing `DataQueue` instance instead.

Properties

QueueLength — Number of items currently held on the queue
zero or positive integer

Read-only property that indicates how many items of data are waiting to be removed from the queue. The value is `0` or a positive integer on the process that created the `DataQueue` instance. The value is `0` on all other processes.

Methods

A `parallel.pool.DataQueue` object has the following methods.

afterEach   Define a function to call when new data is received on a DataQueue

send   Send data from worker to client using a data queue

Examples

Find Length of `DataQueue`

When you send a message to a `DataQueue` object, the message waits in the queue until it is processed by a listener. Each message adds `1` to the queue length. In this example, you use the `QueueLength` property to find the length of a `DataQueue` object.

When a MATLAB process creates a `DataQueue` object, any messages that are sent to the queue are held in the memory of that process. Therefore, the `QueueLength` property on all other processes is `0`. In this example, you create a `DataQueue` object on the client, and send data from a worker.

First, create a parallel pool with one worker.

```
parpool(1);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 1).
```

Then, create a `DataQueue`.

```
q = parallel.pool.DataQueue

q =
  DataQueue with properties:

    QueueLength: 0

```

A newly created `DataQueue` has an empty queue. You can use `parfor` to find `q.QueueLength` on the worker. Find the queue length on the client, and the queue length on the worker.

```
fprintf('On the client: %i\n', q.QueueLength)

On the client: 0

parfor i = 1
    fprintf('On the worker: %i\n', q.QueueLength)
end

On the worker: 0
```

As the queue is empty, the `QueueLength` is `0` for both the client and the worker. Next, send a message to the queue from the worker. Then, use the `QueueLength` property to find the length of the queue.

```
% Send a message first
parfor i = 1
    send(q, 'A message');
end

% Find the length
fprintf('On the client: %i\n', q.QueueLength)
```

```
On the client: 1
```

```
parfor i = 1
    fprintf('On the worker: %i\n', q.QueueLength)
end
```

```
On the worker: 0
```

The QueueLength property is 1 on the client, and 0 on the worker. Create a listener to process the queue by immediately displaying the data.

```
el = afterEach(q, @disp);
```

Wait until the queue is empty, then delete the listener.

```
while q.QueueLength > 0
    pause(0.1);
end
delete(el);
```

Use the QueueLength property to find the length of the queue.

```
fprintf('On the client: %i\n', q.QueueLength)
```

```
On the client: 0
```

QueueLength is 0 because the queue processing is complete.

Send a Message in a parfor-Loop, and Dispatch the Message on the Queue

Construct a DataQueue, and call afterEach.

```
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```

Start a parfor-loop, and send a message. The pending message is passed to the afterEach function, in this example @disp.

```
parfor i = 1:3
    send(q, i);
end;
```

```
     1

     2

     3
```

For more details on listening for data using a DataQueue, see afterEach.

Construct a Simple parfor Wait Bar Using a DataQueue

Create a DataQueue, and use afterEach to specify the function to execute each time the queue receives data. This example calls a subfunction that updates the wait bar.

Create a `parfor`-loop to carry out a computationally demanding task in MATLAB. Use `send` to send some dummy data on each iteration of the `parfor`-loop. When the queue receives the data, `afterEach` calls nUpdateWaitbar in the client MATLAB, and you can observe the wait bar progress.

```matlab
function a = parforWaitbar

D = parallel.pool.DataQueue;
h = waitbar(0, 'Please wait ...');
afterEach(D, @nUpdateWaitbar);

N = 200;
p = 1;

parfor i = 1:N
    a(i) = max(abs(eig(rand(400))));
    send(D, i);
end

    function nUpdateWaitbar(~)
        waitbar(p/N, h);
        p = p + 1;
    end
end
```


Call `afterEach` to Dispatch Data on a Queue

If you call `afterEach` and there are items on the queue waiting to be dispatched, these items are immediately dispatched to the function handle specified by `afterEach`. Call `afterEach` before sending data to the queue, to ensure that on `send`, the function handle `@disp` is called.

Construct a `DataQueue` and call `afterEach`.

```matlab
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```

If you then send messages to the queue, each message is passed to the function handle specified by `afterEach` immediately.

```matlab
parfor i = 1:3
    send(q, i);
end
send(q, 0);
    1

    3
```

```
      2

      0
```

If you send the data to the queue and then call `afterEach`, each of the pending messages are passed to the function handle specified by `afterEach`.

```
q = parallel.pool.DataQueue;
parfor i = 1:3
    send(q, i);
end
afterEach(q, @disp);
```

```
      3

      1

      2
```

See Also

afterEach | gcp | labReceive | labSend | parallel.pool.PollableDataQueue | parfor | poll | send | spmd

Topics
Class Attributes
Property Attributes
"Plot During Parameter Sweep with parfor" on page 9-97

Introduced in R2017a

parallel.pool.PollableDataQueue

Class that enables sending and polling for data between client and workers

Description

`PollableDataQueue` enables sending and polling for data or messages between workers and client in a parallel pool while a computation is being carried out. You can get intermediate values and progress of the computation.

To send data from a parallel pool worker back to the client, first construct a `PollableDataQueue` in the client. Pass this `PollableDataQueue` into a `parfor`-loop or other parallel language construct, such as `parfeval`. From the workers, call `send` to send data back to the client. At the client, use `poll` to retrieve the result of a message or data sent from a worker.

- You can call `send` from the process that calls the constructor, if required.
- You can construct the queue on the workers and send it back to the client to enable communication in the reverse direction. However, you cannot send a queue from one worker to another. Use `spmd`, `labSend`, or `labReceive` instead.
- Unlike all other handle objects, `PollableDataQueue` instances do remain connected when they are sent to workers.

Construction

`p = parallel.pool.PollableDataQueue`

The constructor for a `PollableDataQueue` takes no arguments and returns an object that can be used to send and poll for messages (or data) from different workers. You call the constructor only in the process where you want to receive the data. In the usual workflow, the workers should not be calling the constructor, but should be handed an existing `PollableDataQueue` instance instead.

Properties

`QueueLength` — Number of items currently held on the queue
zero or positive integer

Read-only property that indicates how many items of data are waiting to be removed from the queue. The value is `0` or a positive integer on the process that created the `DataQueue` instance. The value is `0` on all other processes.

Methods

A `parallel.pool.PollableDataQueue` object has the following methods.

poll	Retrieve data sent from a worker
send	Send data from worker to client using a data queue

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Find the Length of a PollableDataQueue

When you send a message to a `PollableDataQueue` object, the message waits in the queue. Each message adds `1` to the queue length. When you use `poll`, one message is collected from the queue. In this example, you use the `QueueLength` property to find the length of a `PollableDataQueue` object.

When a MATLAB process creates a `PollableDataQueue` object, any messages that are sent to the queue are held in the memory of that process. Therefore, the `QueueLength` property on all other processes is `0`. In this example, you create a `PollableDataQueue` object on the client, and send data from a worker.

First, create a parallel pool with one worker.

```
parpool(1);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 1).
```

Create a `PollableDataQueue`.

```
pdq = parallel.pool.PollableDataQueue
```

```
pdq =
  PollableDataQueue with properties:

    QueueLength: 0
```

A newly created `PollableDataQueue` has an empty queue. You can use `parfor` to find `pdq.QueueLength` on the worker. Find the queue length on the client, and the queue length on the worker.

```
fprintf('On the client: %i\n', pdq.QueueLength)
```

```
On the client: 0
```

```
parfor i = 1
    fprintf('On the worker: %i\n', pdq.QueueLength)
end
```

```
On the worker: 0
```

As the queue is empty, the `QueueLength` is `0` for both the client and the worker. Next, send a message to the queue. Then, use the `QueueLength` property to find the length of the queue.

```
% Send a message first
parfor i = 1
    send(pdq, 'A message');
end
```

```
% Find the length
fprintf('On the client: %i\n', pdq.QueueLength)
```

```
On the client: 1
```

```
parfor i = 1
    fprintf('On the worker: %i\n', pdq.QueueLength)
end
```

```
On the worker: 0
```

The QueueLength property is 1 on the client, and 0 on the worker. Use poll to retrieve the message from the queue.

```
msg = poll(pdq);
disp(msg)
```

```
A message
```

Use the QueueLength property to find the length of the queue.

```
fprintf('On the client: %i\n', pdq.QueueLength)
```

```
On the client: 0
```

QueueLength is 0 because the queue processing is complete.

Send a Message in a parfor-loop, and Poll for the Result

Construct a PollableDataQueue.

```
p = parallel.pool.PollableDataQueue;
```

Start a parfor-loop, and send a message, such as data with the value 1.

```
parfor i = 1
    send(p, i);
end
```

Poll for the result.

```
poll(p)
```

```
1
```

For more details on polling for data using a PollableDataQueue, see poll.

See Also

gcp | labReceive | labSend | parallel.pool.DataQueue | parfeval | parfevalOnAll | parfor | poll | send

Topics
Class Attributes
Property Attributes

Introduced in R2017a

parallel.Task

Access task properties and behaviors

Constructors

`createTask`, `findTask`

`getCurrentTask` (in the workspace of the MATLAB worker)

Container Hierarchy

Parent	parallel.Job
Children	none

Description

A parallel.Task object provides access to a task, which executes on a worker as part of a job.

Types

The following table describes the available types of task objects, determined by the type of cluster.

Task Type	Description
parallel.task.MJSTask	Task on MATLAB Job Scheduler cluster
parallel.task.CJSTask	Task on CJS cluster

Methods

All task type objects have the same methods, described in the following table.

cancel	Cancel job or task
delete	Remove job or task object from cluster and memory
listAutoAttachedFiles	List of files automatically attached to job, task, or parallel pool
wait	Wait for job to change state

Properties

Common to All Task Types

The following properties are common to all task object types.

Property	Description
CaptureDiary	Specify whether to return diary output

Property	Description
CreateDateTime	Date and time when the task is created
Diary	Text produced by execution of task object's function
Error	Task error information
ErrorIdentifier	Task error identifier
ErrorMessage	Message from task error
FinishDateTime	Date and time when the task is finished
Function	Function called when evaluating task
ID	Task's numeric identifier
InputArguments	Input arguments to task function
Name	Name of this task
NumOutputArguments	Number of arguments returned by task function
OutputArguments	Output arguments from running task function on worker
Parent	Job object containing this task
StartDateTime	Date and time when the task is started
State	Current state of task
UserData	Information associated with this task object
Warnings	Warning information issued during execution of the task, captured in a struct array with the fields `message`, `identifier`, and `stack`
Worker	Object representing worker that ran this task

MATLAB Job Scheduler Tasks

MATLAB Job Scheduler task objects have the following properties in addition to the common properties:

Property	Description
FailureInfo	Information returned from failed task
FinishedFcn	Callback executed in client when task finishes
MaximumRetries	Maximum number of times to rerun failed task
NumFailures	Number of times tasked failed
RunningFcn	Callback executed in client when task starts running
Timeout	Time limit, in seconds, to complete task

CJS Tasks

CJS task objects have the following properties in addition to the common properties:

Property	Description
SchedulerID	If you have submitted the task to a third-party scheduler, this is the ID that the scheduler gives to the task on submission. For example, this corresponds to the JOBID on a SLURM scheduler.

Help

To get further help on either type of parallel.Task object, including a list of links to help for its properties, type:

```
help parallel.task.MJSTask
help parallel.task.CJSTask
```

See Also

parallel.Cluster, parallel.Job, parallel.Worker

Introduced in R2012a

parallel.Worker

Access worker that ran task

Constructors

`getCurrentWorker` in the workspace of the MATLAB worker.

In the client workspace, a parallel.Worker object is available from the `Worker` property of a parallel.Task object.

Container Hierarchy

Parent	parallel.cluster.MJS
Children	none

Description

A parallel.Worker object provides access to the MATLAB worker session that executed a task as part of a job.

Types

Worker Type	Description
parallel.cluster.MJSWorker	MATLAB worker on MATLAB Job Scheduler cluster
parallel.cluster.CJSWorker	MATLAB worker on CJS cluster
parallel.ThreadWorker	MATLAB thread worker.

Methods

There are no methods for a parallel.Worker object other than generic methods for any objects in the workspace, such as `delete`, etc.

Properties

MATLAB Job Scheduler Worker

The following table describes the properties of a MATLAB Job Scheduler worker.

Property	Description
AllHostAddresses	IP addresses of worker host
Name	Name of worker, set when worker session started
Parent	MATLAB Job Scheduler cluster to which this worker belongs

CJS Worker

The following table describes the properties of an CJS worker.

Property	Description
ComputerType	Type of computer on which the worker ran; the value of the MATLAB function `computer` executed on the worker
Host	Host name where the worker executed the task
ProcessId	Process identifier for the worker

Thread Worker

The following table describes the properties of a thread worker.

Property	Description
ComputerType	Type of computer on which the worker ran; the value of the MATLAB function `computer` executed on the worker
Host	Host name where the worker executed the task

Help

To get further help on the types of parallel.Worker objects, including a list of links to help for its properties, type:

```
help parallel.cluster.MJSWorker
help parallel.cluster.CJSWorker
help parallel.ThreadWorker
```

See Also

parallel.Cluster, parallel.Job, parallel.Task

Introduced in R2012a

ProcessPool

Parallel pool of process workers on the local machine

Description

Use `parpool` to create a parallel pool of process workers on your local machine. After you create the pool, parallel pool features, such as `parfor` or `parfeval`, run on the workers. With the `ProcessPool` object, you can interact with the parallel pool.

Creation

Create a parallel pool of process workers on the local machine by using the `parpool` function.

```
pool = parpool("local")
```

Properties

`AttachedFiles` — Files and folders copied to workers
cell array of character vectors

Files and folders copied to workers, specified as a cell array of character vectors. To attach files and folders to the pool, use `addAttachedFiles`.

`AutoAddClientPath` — Indication whether user-added entries on client path are added to worker paths
`true` (default) | `false`

This property is read-only.

Indication whether user-added entries on client path are added to worker paths, specified as a logical value.

Data Types: `logical`

`Cluster` — Cluster on which the parallel pool is running
cluster object

This property is read-only.

Cluster on which the parallel pool is running, specified as a `parallel.Cluster` object.

`Connected` — Flag that indicates whether the parallel pool is running
`true` | `false`

This property is read-only.

Flag that indicates whether the parallel pool is running, specified as a logical value.

Data Types: `logical`

EnvironmentVariables — Environment variables copied to the workers
cell array of character vectors

This property is read-only.

Environment variables copied to the workers, specified as a cell array of character vectors.

FevalQueue — Queue of FevalFutures to run on the parallel pool
FevalQueue

This property is read-only.

Queue of FevalFutures to run on the parallel pool, specified as an `FevalQueue` object. You can use this property to check the pending and running future variables of the parallel pool. To create future variables, use `parfeval` and `parfevalOnAll`. For more information on future variables, see `Future`.

Data Types: `FevalQueue`

IdleTimeout — Time after which the pool shuts down if idle
nonnegative integer

Time in minutes after which the pool shuts down if idle, specified as an integer greater than zero. A pool is idle if it is not running code on the workers. By default `'IdleTimeout'` is the same as the value in your parallel preferences. For more information on parallel preferences, see "Specify Your Parallel Preferences" on page 5-9.

NumWorkers — Number of workers comprising the parallel pool
integer

This property is read-only.

Number of workers comprising the parallel pool, specified as an integer.

SpmdEnabled — Indication if pool can run `spmd` code
true (default) | false

This property is read-only.

Indication if pool can run `spmd` code, specified as a logical value.

Data Types: `logical`

Object Functions

addAttachedFiles	Attach files or folders to parallel pool
delete	Shut down parallel pool
listAutoAttachedFiles	List of files automatically attached to job, task, or parallel pool
parfeval	Execute function asynchronously on parallel pool worker
parfevalOnAll	Execute function asynchronously on all workers in parallel pool
ticBytes	Start counting bytes transferred within parallel pool
tocBytes	Read how many bytes have been transferred since calling ticBytes
updateAttachedFiles	Update attached files or folders on parallel pool

See Also
parpool

Topics
"Run Code on Parallel Pools" on page 2-55
"Choose Between Thread-Based and Process-Based Environments" on page 2-60

Introduced in R2020a

RemoteClusterAccess

Connect to schedulers when client utilities are not available locally

Constructor

`parallel.cluster.RemoteClusterAccess`

`r = parallel.cluster.RemoteClusterAccess(username)`

`r = parallel.cluster.RemoteClusterAccess(username,P1,V1,...,Pn,Vn)`

Description

`parallel.cluster.RemoteClusterAccess` allows you to establish a connection and run commands on a remote host. This class is intended for use with the generic scheduler interface when using remote submission of jobs or on nonshared file systems.

`r = parallel.cluster.RemoteClusterAccess(username)` uses the supplied username when connecting to the remote host, and returns a `RemoteClusterAccess` object r. You will be prompted for a password when establishing the connection.

`r = parallel.cluster.RemoteClusterAccess(username,P1,V1,...,Pn,Vn)` allows additional parameter-value pairs that modify the behavior of the connection. The accepted parameters are:

- `'IdentityFilename'` — A character vector containing the full path to the identity file to use when connecting to a remote host. If `'IdentityFilename'` is not specified, you are prompted for a password when establishing the connection.
- `'IdentityFileHasPassphrase'` — A logical indicating whether or not the identity file requires a passphrase. If true, you are prompted for a password when establishing a connection. If an identity file is not supplied, this property is ignored. This value is `false` by default.

For more information and detailed examples, see "Submitting from a Remote Host" on page 6-26 and "Submitting without a Shared File System" on page 6-27.

Methods

Method Name	Description
connect	`connect(r,clusterHost)` establishes a connection to the specified host using the user credential options supplied in the constructor. File mirroring is not supported.
	`connect(r,clusterHost,remoteDataLocation)` establishes a connection to the specified host using the user credential options supplied in the constructor. `remoteDataLocation` identifies a folder on the `clusterHost` that is used for file mirroring. The user credentials supplied in the constructor must have write access to this folder.

Method Name	Description
`disconnect`	`disconnect(r)` disconnects the existing remote connection. The `connect` method must have already been called.
`doLastMirrorForJob`	`doLastMirrorForJob(r,job)` performs a final copy of changed files from the remote `DataLocation` to the local `DataLocation` for the supplied job. Any running mirrors for the job also stop and the job files are removed from the remote `DataLocation`. The `startMirrorForJob` or `resumeMirrorForJob` method must have already been called.
`getRemoteJobLocation`	`getRemoteJobLocation(r,jobID,remoteOS)` returns the full path to the remote job location for the supplied `jobID`. Valid values for `remoteOS` are `'pc'` and `'unix'`.
`isJobUsingConnection`	`isJobUsingConnection(r,jobID)` returns `true` if the job is currently being mirrored.
`resumeMirrorForJob`	`resumeMirrorForJob(r,job)` resumes the mirroring of files from the remote `DataLocation` to the local `DataLocation` for the supplied job. This is similar to the `startMirrorForJob` method, but does not first copy the files from the local `DataLocation` to the remote `DataLocation`. The `connect` method must have already been called. This is useful if the original client MATLAB session has ended, and you are accessing the same files from a new client session.
`runCommand`	`[status,result] = runCommand(r,command)` runs the supplied command on the remote host and returns the resulting status and standard output. The `connect` method must have already been called.
`startMirrorForJob`	`startMirrorForJob(r,job)` copies all the job files from the local `DataLocation` to the remote `DataLocation`, and starts mirroring files so that any changes to the files in the remote `DataLocation` are copied back to the local `DataLocation`. The `connect` method must have already been called.
`stopMirrorForJob`	`stopMirrorForJob(r,job)` immediately stops the mirroring of files from the remote `DataLocation` to the local `DataLocation` for the specified job. The `startMirrorForJob` or `resumeMirrorForJob` method must have already been called. This cancels the running mirror and removes the files for the job from the remote location. This is similar to `doLastMirrorForJob`, except that `stopMirrorForJob` makes no attempt to ensure that the local job files are up to date. For normal mirror stoppage, use `doLastMirrorForJob`.
`getConnectedAccess`	`getConnectedAccess(host,username)` returns a `RemoteClusterAccess` object that is connected to the supplied host. This function may return a previously constructed `RemoteClusterAccess` object if one exists. `getConnectedAccess(...,P1,V1,...Pn,Vn)` passes the additional parameters to the `RemoteClusterAccess` constructor.

Method Name	Description
`getConnectedAccessWithMirror`	`getConnectedAccessWithMirror(host,location,username)` returns a `RemoteClusterAccess` object that is connected to the supplied host, using the location as the mirror location. This function may return a previously constructed `RemoteClusterAccess` object if one exists. `getConnectedAccessWithMirror(...,P1,V1,...Pn,Vn)` passes the additional parameters to the `RemoteClusterAccess` constructor.

Properties

A `RemoteClusterAccess` object has the following read-only properties. Their values are set when you construct the object or call its `connect` method.

Property Name	Description
`Hostname`	Name of the remote host to access.
`IdentityFileHasPassphrase`	Indicates if the identity file requires a passphrase.
`IdentityFilename`	Full path to the identity file used when connecting to the remote host.
`IsConnected`	Indicates if there is an active connection to the remote host.
`IsFileMirrorSupported`	Indicates if file mirroring is supported for this connection. This is `false` if no remote `DataLocation` is supplied to the `connect()` method.
`JobStorageLocation`	Location on the remote host for files that are being mirrored.
`UseIdentityFile`	Indicates if an identity file should be used when connecting to the remote host.
`Username`	User name for connecting to the remote host.

Examples

Mirror files from the remote data location. Assume the object `job` represents a job on your generic scheduler.

```
remoteConnection = parallel.cluster.RemoteClusterAccess('testname');
connect(remoteConnection,'headnode1','/tmp/filemirror');
startMirrorForJob(remoteConnection,job);
submit(job)
% Wait for the job to finish
wait(job);

% Ensure that all the local files are up to date, and remove the
% remote files
doLastMirrorForJob(remoteConnection,job);

% Get the output arguments for the job
results = fetchOutputs(job)
```

For more information and examples, see "Submitting from a Remote Host" on page 6-26 and "Submitting without a Shared File System" on page 6-27.

See Also

Topics
"Plugin Scripts for Generic Schedulers" on page 6-17

Introduced in R2011a

ThreadPool

Parallel pool of thread workers on the local machine

Description

Use `parpool` to create a parallel pool of thread workers on your local machine. After you create the pool, parallel pool features, such as `parfor` or `parfeval`, run on the workers. With the `ThreadPool` object, you can interact with the parallel pool.

Creation

Create a parallel pool of thread workers on the local machine by using the `parpool` function.

```
pool = parpool("threads")
```

Properties

`NumWorkers` — Number of thread workers comprising the parallel pool
integer

This property is read-only.

Number of thread workers comprising the parallel pool, specified as an integer.

Object Functions

delete   Shut down parallel pool
parfeval   Execute function asynchronously on parallel pool worker
parfevalOnAll   Execute function asynchronously on all workers in parallel pool

For compatibility with process workers, the following object functions are also supported. You do not need to use them because data and file transfer is not necessary for pools of thread workers.

addAttachedFiles   Attach files or folders to parallel pool
listAutoAttachedFiles   List of files automatically attached to job, task, or parallel pool
ticBytes   Start counting bytes transferred within parallel pool
tocBytes   Read how many bytes have been transferred since calling ticBytes
updateAttachedFiles   Update attached files or folders on parallel pool

See Also
parpool

Topics
"Run Code on Parallel Pools" on page 2-55
"Choose Between Thread-Based and Process-Based Environments" on page 2-60

Introduced in R2020a

Functions

addAttachedFiles

Attach files or folders to parallel pool

Syntax

```
addAttachedFiles(poolobj,files)
```

Description

`addAttachedFiles(poolobj,files)` adds extra attached files to the specified parallel pool. These files are transferred to each worker and are treated exactly the same as if they had been set at the time the pool was opened — specified by the parallel profile or the `'AttachedFiles'` argument of the `parpool` function.

Examples

Add Attached Files to Current Parallel Pool

Add two attached files to the current parallel pool.

```
poolobj = gcp;
addAttachedFiles(poolobj,{'myFun1.m','myFun2.m'})
```

Input Arguments

poolobj — Pool to which files attach
pool object

Pool to which files attach, specified as a pool object.

Example: `poolobj = gcp;`

files — Files or folders to attach
character vector | cell array

Files or folders to attach, specified as a character vector or cell array of character vectors. Each character vector can specify either an absolute or relative path to a file or folder.

Example: `{'myFun1.m','myFun2.m'}`

See Also
`gcp` | `getAttachedFilesFolder` | `listAutoAttachedFiles` | `parpool` | `updateAttachedFiles`

Topics
"Add and Modify Cluster Profiles" on page 5-14

Introduced in R2013b

afterAll

Specify a function to invoke after all parallel.Futures complete

Syntax

```
outputFuture = afterAll(futures,funtocall,nout)
outputFuture = afterAll(futures,funtocall,nout,'PassFuture',passFuture)
```

Description

`outputFuture = afterAll(futures,funtocall,nout)` automatically evaluates `funtocall` on the output arguments of all the futures in `futures` when they are all complete, and returns `outputFuture` to hold the result. `afterAll` evaluates `funtocall` on the vertical concatenation of the output arguments of all futures. If the number of output arguments of the elements in futures differ, `afterAll` uses the minimum and disregards the ending output arguments. `afterAll` invokes `funtocall` with `nout` output arguments.

A useful application for `afterAll` is to update user interfaces such as plots and apps during parallel computations using `parfeval`. For example, you can send several computations to workers using `parfeval` and update your user interface when all of them finish using `afterAll`.

`outputFuture = afterAll(futures,funtocall,nout,'PassFuture',passFuture)` behaves the same if `passFuture` is `false`. If `passFuture` is `true`, `afterAll` invokes `funtocall` on the array of futures `futures` and not on their output arguments. This happens even if elements of `futures` encountered errors.

Examples

Call `afterAll` on `parfeval` Computations

You can use `afterAll` to automatically invoke functions on all of the combined outputs of your `parfeval` computations.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx = 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

Display the maximum element among all of those vectors after they are created. `afterAll` executes the function handle on the combined output of all the futures when they all become ready.

```
afterAll(f, @(r) disp(max(r)), 0);

    0.9998
```

Combine `afterEach` and `afterAll`

You can combine `afterEach` and `afterAll` to automatically invoke more functions on the results of futures. Both `afterEach` and `afterAll` generate future variables that can be used again in `afterEach` and `afterAll`.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx= 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 8 workers.
```

Compute the largest element in each of those vectors when they become ready. `afterEach` executes the function handle on the output of each future when they become ready and creates another future to hold the results.

```
maxFuture = afterEach(f, @(r) max(r), 1);
```

To compute the minimum value among them, call `afterAll` on this new future. `afterAll` executes a function on the combined output arguments of all the futures after they all complete. In this case, `afterAll` executes the function `min` on the outputs of `maxFuture` after completing and creates another future to hold the result.

```
minFuture = afterAll(maxFuture, @(r) min(r), 1);
```

You can fetch the result using `fetchOutputs`. `fetchOutput` waits until the future completes to gather the results.

```
fetchOutputs(minFuture)
```

```
ans = 0.9973
```

You can check the result of `afterEach` by calling `fetchOutputs` on its future variable.

```
fetchOutputs(maxFuture)
```

```
ans = 10×1

    0.9996
    0.9989
    0.9994
    0.9973
    1.0000
    1.0000
    0.9989
    0.9994
    0.9998
    0.9999
```

Update a User Interface Asynchronously Using `afterEach` and `afterAll`

You can perform asynchronous computations on workers using `parfeval` and leave the user interface responsive. Use `afterEach` to update the user interface when intermediate computations are ready. Use `afterAll` to update the user interface when all the computations are ready.

Create a simple user interface using a `waitbar`.

```
h = waitbar(0, 'Waiting...');
```


Use `parfeval` to carry out time-consuming computations in the workers, for example, eigenvalues of random matrices. The computations happen asynchronously and the user interface updates during computation. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx = 1:100
    f(idx) = parfeval(@(n) real(eig(randn(n))), 1, 5e2);
end
```

Compute the largest value in each of the computations when they become ready using `afterEach`. Update the proportion of finished futures in the waitbar when each of them completes using `afterEach`.

```
maxFuture = afterEach(f, @max, 1);
updateWaitbarFuture = afterEach(f, @(~) waitbar(sum(strcmp('finished', {f.State}))/numel(f), h),
```

Close the waitbar when all the computations are done. Use `afterAll` on `updateWaitbarFuture` to continue automatically with a close operation. `afterAll` obtains the figure handle from `updateWaitbarFuture` and executes its function on it.

```
closeWaitbarFuture = afterAll(updateWaitbarFuture, @(h) delete(h), 0);
```

Show a histogram after all the maximum values are computed. Use `afterAll` on `maxFuture` to continue the operation automatically. `afterAll` obtains the maximum values from `maxFuture` and calls `histogram` on them.

```
showsHistogramFuture = afterAll(maxFuture, @histogram, 0);
```

Handle Errors in Future Variables

When computations for future variables result in an error, by default, `afterAll` does not evaluate its function. If you want to handle any errors, for example, you have a user interface that you want to update, you can use the name-value pair `PassFuture`. When set to `true`, the future variable is passed to the callback function. You can call `fetchOutputs` on it, process the outputs, and handle any possible errors.

Send computations to the workers using `parfeval`. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created. If your `parfeval` computations result in an error, the future variable errors, and its `Error` property reflects it.

```
errorFuture = parfeval( @(n) randn(n), 0, 0.5);
```

```
Starting parallel pool (parpool) using the 'Local' profile ...
connected to 4 workers.
```

```
wait(errorFuture);
errorFuture.Error
```

```
ans =
  ParallelException with properties:

     identifier: 'MATLAB:NonIntegerInput'
        message: 'Size inputs must be integers.'
          cause: {}
     remotecause: {[1×1 MException]}
          stack: [1×1 struct]
```

If you use `afterAll` on that future, the callback function is not evaluated. In the code below, the `msgbox` is not executed because the future errors.

```
afterAll(errorFuture, @() msgbox('Operation completed'), 0);
```

To handle futures that result in an error, use the name-value pair `PassFuture` when calling `afterAll`. The future variable is passed to the callback function instead of its outputs. Call `fetchOutputs` on it, and process its outputs. If the future results in an error, `fetchOutputs` throws an error that you can catch and handle. For example, the following code shows an error dialog box.

```
afterAll(errorFuture, @handleError, 0, 'PassFuture',true);
```
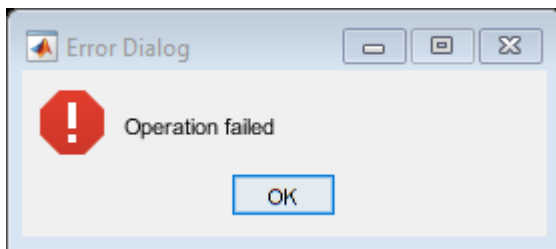


```
function handleError (f)
try
    output = fetchOutputs(f);
    % Do something with the output
catch
    errordlg('Operation failed');
end
end
```

Input Arguments

futures — Futures
parallel.Future

Futures, specified as an array of `parallel.Future`. `funtocall` is invoked on the combined outputs of all its elements when they all become ready. You can use `parfeval` to create futures.

If any element of `futures` encounters an error, `funtocall` is not invoked, and `outputFuture` completes with an error. To see if the future completes with an error, you can check the `Error` property of `outputFuture`. If you cancel an element of `futures`, this results in the same behavior as if the element encountered an error.

Example: `future = parfeval(@rand,1,1000,1); afterAll(future,@max,1);`

Data Types: `parallel.Future`

funtocall — Function to execute
function handle

Function to execute, specified as a function to call on the combined output arguments of all the futures in `futures` when they all become ready. `funtocall` is evaluated on the MATLAB client, not on the parallel pool workers.

Example: `funtocall = @max`

Data Types: `function handle`

nout — Number of outputs
integer

Number of outputs, specified as an integer, expected from `funtocall`.

Example: `afterAll(futures,@max,1)`

Data Types: `scalar`

passFuture — Type of input arguments to funtocall
logical scalar

Indicator, specified as a logical scalar that determines the type of input arguments to `funtocall`. If set to `true`, the future array `futures` is passed to `funtocall`. Otherwise, the output arguments of all futures in `futures` are passed to `funtocall`. This argument is optional and is `false` by default.

You can use this approach if you want to handle any errors. Set `passFuture` to `true` so that `afterAll` invokes `funtocall` on the outputs of `futures`, even if they encountered errors. You must call `fetchOutputs` on the input argument to `funtocall` to extract the results. If the future results in an error, `fetchOutputs` throws an error that you can catch and handle.

Example: `afterAll(futures,@(f) disp(fetchOutputs(f)),0,'PassFuture',true)`

Data Types: `logical scalar`

Output Arguments

outputFuture — Future
parallel.Future

Future, returned as a `parallel.Future` to hold the results of evaluating `funtocall` on the combined output arguments of all the futures in `futures` when they all become ready.

To extract the results, call `fetchOutputs` on `outputFuture`:

```
outputFutures = afterAll(futures,funtocall,nout);
[out1,out2,...,outM] = fetchOutputs(outputFutures);
```

Note that this is equivalent to calling `funtocall` after fetching the outputs in `futures`, except that `afterAll` calls automatically `funtocall` when all elements in `futures` complete:

```
[tmp1,tmp2,...,tmpN] = fetchOutputs(futures);
[out1,out2,...,outM] = funtocall(tmp1,tmp2,...tmpN)
```

`N` is the number of outputs from `futures` and `M` is the number of outputs specified in `afterAll` with the input argument `nout`.

Tips

- Use `afterAll` on any of the futures returned from `parfeval`, `parfevalOnAll`, `afterEach`, `afterAll`, or an array containing a combination of them. For example, use `afterAll` to automatically invoke more functions on the results of another `afterEach` or `afterAll`. You can invoke `afterAll` on futures before and after they finish.

- Use `cancel` on a future returned from `afterAll` to cancel its execution. If you invoke `afterAll` on a canceled future, this results in the same behavior as if the future encountered an error.

See Also
`afterEach` | `parallel.Future` | `parfeval`

Introduced in R2018a

afterEach

Define a function to call when new data is received on a DataQueue

Syntax

```
listener = afterEach(queue, funtocall)
```

Description

`listener = afterEach(queue, funtocall)` specifies a function `funtocall` to execute each time the `queue` receives new data. You can specify multiple different functions to call, because each call to `afterEach` creates a new listener on the queue. If you want to specify another function, call `afterEach` again. To remove the registration of the function with the `queue`, delete the returned `listener` object.

You must call `afterEach` in the same process where you created the data queue, otherwise an error occurs. After calling `afterEach`, any current data in the queue is dispatched immediately to the supplied function.

Examples

Call `afterEach` to Dispatch Data on a Queue

If you call `afterEach` and there are items on the queue waiting to be dispatched, these items are immediately dispatched to the `afterEach` function. Call `afterEach` before sending data to the queue, to ensure that on `send`, the function handle specified by `afterEach` is called.

Construct a `DataQueue` and call `afterEach`.

```
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```

If you then send messages to the queue, each message is passed to the function handle specified by `afterEach` immediately.

```
parfor i = 1
    send(q, 2);
end

    2

send(q, 3)

    3
```

You can also first send various messages to the queue. When you call `afterEach`, the pending messages are passed to the `afterEach` function, in this example to the function handle `@disp`.

```
q = parallel.pool.DataQueue;
parfor i = 1
    send(q, 2);
```

```
end
send(q, 3)

afterEach(q, @disp);

    2

    3
```

Remove a Callback by Deleting the Listener

Construct a `DataQueue` and create a listener.

```
D = parallel.pool.DataQueue;
listener = D.afterEach(@disp);
```

Send some data with the value 1.

```
D.send(1)

    1
```

Delete the listener.

```
delete(listener)
D.send(1)
```

No data is returned because you have removed the callback by deleting the listener.

Input Arguments

queue — Data queue
parallel.pool.DataQueue

Data queue, specified as a `parallel.pool.DataQueue` object.

Example: `q = parallel.pool.DataQueue;`

funtocall — Function handle
function handle

Function handle, specifying the function added to the list of functions to call when a piece of new data is received from `queue`.

Example: `listener = afterEach(queue, funtocall)`

All callback functions must accept `data` as single argument.

`afterEach(queue, @foo)` expects a function handle `@foo` to a function of the form

```
function foo(data)
end
```

When `send(queue, someData)` is called on the worker, `someData` is serialized and sent back to the client. `someData` is deserialized on the client and passed as the input to `foo(data)`.

Output Arguments

`listener` — listener
event.listener

Listener object created by `afterEach`, returned as the handle to an `event.listener` object.

See Also
event.listener | parallel.pool.DataQueue | parallel.pool.PollableDataQueue | parfor | poll | send

Introduced in R2017a

afterEach

Specify a function to invoke after each parallel.Future completes

Syntax

```
outputFuture = afterEach(futures,funtocall,nout)
outputFuture = afterEach(futures,funtocall,nout,'PassFuture',passFuture)
```

Description

`outputFuture = afterEach(futures,funtocall,nout)` automatically evaluates `funtocall` on the output arguments of each of the elements in `futures` as they become ready. `afterEach` calls `funtocall` with `nout` output arguments and produces `outputFuture` to hold the outputs.

A useful application for `afterEach` is to update user interfaces such as plots and apps during parallel computations using `parfeval`. For example, you can send several computations to workers using `parfeval` and update your user interface when each of those finishes using `afterEach`.

`outputFuture = afterEach(futures,funtocall,nout,'PassFuture',passFuture)` behaves the same if `passFuture` is `false`. If `passFuture` is `true`, `afterEach` invokes `funtocall` on each element in `futures`, and not on their output arguments. This happens even if elements of `futures` encountered errors.

Examples

Call `afterEach` on parfeval Computations

You can use `afterEach` to automatically invoke functions on each of the results of `parfeval` computations.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx = 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

Display the maximum element in each of those vectors after they are created. `afterEach` executes the function handle on the output of each future when they become ready.

```
afterEach(f, @(r) disp(max(r)), 0);
```

```
    0.9975

    0.9990

    0.9982

    0.9991
```

```
    0.9982

    0.9998

    0.9999

    0.9986

    0.9996

    0.9990
```

Combine `afterEach` and `afterAll`

You can combine `afterEach` and `afterAll` to automatically invoke more functions on the results of futures. Both `afterEach` and `afterAll` generate future variables that can be used again in `afterEach` and `afterAll`.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx= 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 8 workers.
```

Compute the largest element in each of those vectors when they become ready. `afterEach` executes the function handle on the output of each future when they become ready and creates another future to hold the results.

```
maxFuture = afterEach(f, @(r) max(r), 1);
```

To compute the minimum value among them, call `afterAll` on this new future. `afterAll` executes a function on the combined output arguments of all the futures after they all complete. In this case, `afterAll` executes the function `min` on the outputs of `maxFuture` after completing and creates another future to hold the result.

```
minFuture = afterAll(maxFuture, @(r) min(r), 1);
```

You can fetch the result using `fetchOutputs`. `fetchOutput` waits until the future completes to gather the results.

```
fetchOutputs(minFuture)
```

```
ans = 0.9973
```

You can check the result of `afterEach` by calling `fetchOutputs` on its future variable.

```
fetchOutputs(maxFuture)
```

```
ans = 10×1

    0.9996
```

```
0.9989
0.9994
0.9973
1.0000
1.0000
0.9989
0.9994
0.9998
0.9999
```

Update a User Interface Asynchronously Using `afterEach` and `afterAll`

You can perform asynchronous computations on workers using `parfeval` and leave the user interface responsive. Use `afterEach` to update the user interface when intermediate computations are ready. Use `afterAll` to update the user interface when all the computations are ready.

Create a simple user interface using a `waitbar`.

```
h = waitbar(0, 'Waiting...');
```


Use `parfeval` to carry out time-consuming computations in the workers, for example, eigenvalues of random matrices. The computations happen asynchronously and the user interface updates during computation. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx = 1:100
    f(idx) = parfeval(@(n) real(eig(randn(n))), 1, 5e2);
end
```

Compute the largest value in each of the computations when they become ready using `afterEach`. Update the proportion of finished futures in the waitbar when each of them completes using `afterEach`.

```
maxFuture = afterEach(f, @max, 1);
updateWaitbarFuture = afterEach(f, @(~) waitbar(sum(strcmp('finished', {f.State}))/numel(f), h),
```

Close the waitbar when all the computations are done. Use `afterAll` on `updateWaitbarFuture` to continue automatically with a close operation. `afterAll` obtains the figure handle from `updateWaitbarFuture` and executes its function on it.

```
closeWaitbarFuture = afterAll(updateWaitbarFuture, @(h) delete(h), 0);
```

Show a histogram after all the maximum values are computed. Use `afterAll` on `maxFuture` to continue the operation automatically. `afterAll` obtains the maximum values from `maxFuture` and calls `histogram` on them.

```
showsHistogramFuture = afterAll(maxFuture, @histogram, 0);
```

Handle Errors in Future Variables

When computations for future variables result in an error, by default, `afterEach` does not evaluate its function on the elements that failed. If you want to handle any errors, for example, you have a user interface that you want to update, you can use the name-value pair `PassFuture`. When set to `true`, the future variable is passed to the callback function. You can call `fetchOutputs` on it, process the outputs, and handle any possible errors.

Send computations to the workers using `parfeval`. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created. If your `parfeval` computations result in an error, the future variable errors, and its `Error` property reflects it.

```
errorFuture = parfeval(@(n) randn(n), 0, 0.5);
wait(errorFuture);
errorFuture.Error

ans =
  ParallelException with properties:

     identifier: 'MATLAB:NonIntegerInput'
        message: 'Size inputs must be integers.'
          cause: {}
    remotecause: {[1×1 MException]}
          stack: [1×1 struct]
```

If you use `afterEach` on that future, the callback function is not evaluated on those elements in the future that errored. In the code below, the `msgbox` is not executed because the future errors.

```
afterEach(errorFuture, @() msgbox('Operation completed'), 0);
```

To handle futures that result in errors, use the name-value pair `PassFuture` when calling `afterEach`. The future variable is passed to the callback function instead of its outputs. Call `fetchOutputs` on the future variable, and process its outputs. If the future results in an error, `fetchOutputs` throws an error that you can catch and handle. The following code shows an error dialog box.

```
afterEach(errorFuture, @handleError, 0, 'PassFuture', true);
```
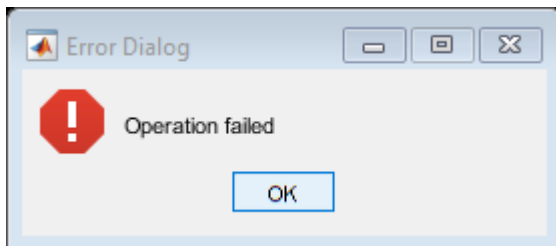


```
function handleError(f)
    try
        output = fetchOutputs(f);
        % Do something with the output
```

```
        catch
            errordlg('Operation failed');
        end
end
```

Input Arguments

futures — Futures
parallel.Future

Futures, specified as an array of `parallel.Future`. `funtocall` is invoked on each of its elements when they become ready. You can use `parfeval` to create a future.

If an element of `futures` encounters an error, `funtocall` is not evaluated for that element of `futures`, however, it is evaluated for other elements of `futures` that do not encounter errors. To see if there are any futures with errors, you can check the `Error` property of `outputFuture`. This property is an empty cell array if there are no errors. If there are errors, it is a cell array that contains as many cells as futures in `futures`. A cell contains an error if the corresponding element of `futures` encountered an error, and is otherwise empty. If you cancel an element of `futures`, this results in the same behavior as if the element encountered an error.

Example: `future = parfeval(@rand,1,1000,1); afterEach(future,@max,1);`

Data Types: `parallel.Future`

funtocall — Function to execute
function handle

Function to execute, specified as a function to call on the output arguments of each of the futures in `futures` when they become ready. `funtocall` is evaluated on the MATLAB client and not on the parallel pool workers.

Example: `funtocall = @max`

Data Types: `function handle`

nout — Number of outputs
integer

Number of outputs, specified as an integer, expected from `funtocall`.

Example: `afterEach(futures,@max,1)`

Data Types: `scalar`

passFuture — Type of input arguments to `funtocall`
logical scalar

Indicator, specified as a logical scalar that determines the type of input arguments to `funtocall`. If set to `true`, each future in `futures` is passed to `funtocall`. Otherwise, the output arguments of each future in `futures` are passed to `funtocall`. This argument is optional and is `false` by default.

You can use this approach if you want to handle any errors. Set `passFuture` to `true` so that `afterEach` invokes `funtocall` on each element in `futures`, even if they encountered errors. You must call `fetchOutputs` on the input argument to `funtocall` to extract the results. If there are any futures with errors, `fetchOutputs` throws an error that you can catch and handle.

Example: afterEach(futures,@(f) disp(fetchOutputs(f)),0,'PassFuture',true)

Data Types: logical scalar

Output Arguments

outputFuture — Future
parallel.Future

Future, returned as a `parallel.Future` to hold the results of evaluating `funtocall` on each of the futures in `futures` when they become ready. To extract the results, call `fetchOutputs` on `outputFuture`.

Tips

- Use `afterEach` on any of the futures returned from `parfeval`, `parfevalOnAll`, `afterAll`, `afterEach,` or an array containing a combination of them. For example, use `afterEach` to automatically invoke more functions on the results of another `afterAll` or `afterEach`. You can invoke `afterEach` on futures before and after they finish.

- Use `cancel` on a future returned from `afterEach` to cancel its execution. If you invoke `afterEach` on a canceled future, `afterEach` behaves the same way as if the future had an error.

See Also
afterAll | parallel.Future | parfeval

Introduced in R2018a

arrayfun

Apply function to each element of array on GPU

Syntax

```
B = arrayfun(FUN,A)
B = arrayfun(FUN,A1,...,An)
[B1,...,Bm] = arrayfun(FUN, ___ )
```

Description

Note This function behaves similarly to the MATLAB function `arrayfun`, except that the evaluation of the function happens on the GPU, not on the CPU. Any required data not already on the GPU is moved to GPU memory. The MATLAB function passed in for evaluation is compiled and then executed on the GPU. All output arguments are returned as gpuArray objects. You can retrieve gpuArray data using the `gather` function.

`B = arrayfun(FUN,A)` applies the function FUN to each element of the gpuArray A. `arrayfun` then concatenates the outputs from FUN into output gpuArray B. B is the same size as A and B(i,j,...) = FUN(A(i,j,...)). The input argument FUN is a function handle to a MATLAB function that takes one input argument and returns a scalar value. FUN is called as many times as there are elements of A.

You cannot specify the order in which `arrayfun` calculates the elements of B or rely on them being done in any particular order.

`B = arrayfun(FUN,A1,...,An)` applies FUN to the elements of the arrays A1,...,An, so that B(i,j,...) = FUN(A1(i,j,...),...,An(i,j,...)). The function FUN must take n input arguments and return a scalar. The nonsingleton dimensions of the inputs A1,...,An must all match, or the inputs must be scalar. Any singleton dimensions or scalar inputs are virtually replicated before being input to the function FUN.

`[B1,...,Bm] = arrayfun(FUN, ___ )` returns multiple output arrays B1,...,Bm when the function FUN returns m output values. `arrayfun` calls FUN each time with as many outputs as there are in the call to `arrayfun`, that is, m times. If you call `arrayfun` with more output arguments than supported by FUN, MATLAB generates an error. FUN can return output arguments having different data types, but the data type of each output must be the same each time FUN is called.

Examples

Run a Function on the GPU

In this example, a small function applies correction data to an array of measurement data. The function defined in the file `myCal.m` is shown here.

```
function c = myCal(rawdata, gain, offset)
    c = (rawdata .* gain) + offset;
end
```

The function performs only element-wise operations when applying a gain factor and offset to each element of the `rawdata` array.

Create a nominal measurement.

```
meas = ones(1000)*3; % 1000-by-1000 matrix
```

The function allows the gain and offset to be arrays of the same size as `rawdata`, so that unique corrections can be applied to individual measurements. In a typical situation, you can keep the correction data on the GPU so that you do not have to transfer it for each application:

```
gn   = rand(1000,'gpuArray')/100 + 0.995;
offs = rand(1000,'gpuArray')/50  - 0.01;
```

Run your calibration function on the GPU.

```
corrected = arrayfun(@myCal,meas,gn,offs);
```

The function runs on the GPU because the input arguments `gn` and `offs` are already in GPU memory. The input array `meas` is converted to a gpuArray before the function runs.

Retrieve the corrected results from the GPU to the MATLAB workspace.

```
results = gather(corrected);
```

Use a Function with Multiple Outputs

You can define a MATLAB function as follows.

```
function [o1,o2] = aGpuFunction(a,b,c)
    o1 = a + b;
    o2 = o1 .* c + 2;
end
```

Evaluate this function on the GPU.

```
s1 = rand(400,'gpuArray');
s2 = rand(400,'gpuArray');
s3 = rand(400,'gpuArray');
[o1,o2] = arrayfun(@aGpuFunction,s1,s2,s3);
whos

 Name        Size          Bytes  Class

  o1        400x400           108  gpuArray
  o2        400x400           108  gpuArray
  s1        400x400           108  gpuArray
  s2        400x400           108  gpuArray
  s3        400x400           108  gpuArray
Use gather to retrieve the data from the GPU to the MATLAB
```

Use `gather` to retrieve the data from the GPU to the MATLAB workspace.

```
d = gather(o2);
```

Use Random Number Functions with `arrayfun`

The function `myfun.m` generates and uses a random number R.

```
function Y = myfun(X)
    R = rand();
    Y = R.*X;
end
```

If you use `arrayfun` to run this function with an input variable that is a gpuArray, the function runs on the GPU. The size of X determines the number of random elements to generate. The following code passes the gpuArray matrix G to `myfun` on the GPU.

```
G = 2*ones(4,4,'gpuArray')
H = arrayfun(@myfun, G)
```

Because G is a 4-by-4 gpuArray, `myfun` generates 16 random value scalar elements for R, one for each calculation with an element of G.

Input Arguments

FUN — Function to apply
function handle

Function to apply to the elements of the input arrays, specified as a function handle. FUN must return scalar values. For each output argument, FUN must return values of the same class each time it is called. FUN must accept numerical or logical input data.

FUN must be a handle to a function that is written in the MATLAB language. You cannot specify FUN as a handle to a MEX-function.

FUN can contain the following built-in MATLAB functions and operators.

abs	csch	log2	sin	Scalar expansion versions of the following:
and	double	log10	single	
acos	eps	log1p	sinh	
acosh	eq	logical	sqrt	*
acot	erf	lt	tan	/
acoth	erfc	max	tanh	\
acsc	erfcinv	min	times	^
acsch	erfcx	minus	true	
asec	erfinv	mod	uint8	
asech	exp	NaN	uint16	Branching instructions:
asin	expm1	ne	uint32	
asinh	false	not	uint64	break
atan	fix	ones	xor	continue
atan2	floor	or	zeros	else
atanh	gamma	pi		elseif
beta	gammaln	plus	+	for
betaln	ge	pow2	-	if
bitand	gt	power	.*	return
bitcmp	hypot	rand	./	while
bitget	imag	randi	.\	
bitor	Inf	randn	.^	
bitset	int8	rdivide	==	
bitshift	int16	real	~=	
bitxor	int32	reallog	<	
cast	int64	realmax	<=	
ceil	intmax	realmin	>	
complex	intmin	realpow	>=	
conj	isfinite	realsqrt	&	
cos	isinf	rem	|	
cosh	isnan	round	~	
cot	ldivide	sec	&&	
coth	le	sech	||	
csc	log	sign		

Functions that create arrays (such as Inf, NaN, ones, rand, randi, randn, and zeros) do not support size specifications as input arguments. Instead, the size of the generated array is determined by the size of the input variables to your functions. Enough array elements are generated to satisfy the needs of your input or output variables. You can specify the data type using both class and "like" syntaxes. The following examples show supported syntaxes for array-creation functions:

```
a = rand;
b = ones();
c = zeros("like", x);
d = Inf("single");
e = randi([0 9], "uint32");
```

When you use rand, randi, and randn to generate random numbers within FUN, each element is generated from a different substream. For more information about generating random numbers on the GPU, see "Random Number Streams on a GPU" on page 8-6.

A — Input array
scalars | vectors | matrices | multidimensional arrays

Input array, specified as scalars, vectors, matrices, or multidimensional arrays. At least one input array argument must be a gpuArray for arrayfun to run on the GPU. Each array that is stored in CPU memory is converted to a gpuArray before the function is evaluated. If you plan to make several calls to arrayfun with the same array, it is more efficient to convert that array to a gpuArray.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

B — Output array
gpuArray

Output array, returned as a gpuArray.

Tips

- The first time you call `arrayfun` to run a particular function on the GPU, there is some overhead time to set up the function for GPU execution. Subsequent calls of `arrayfun` with the same function can run faster.

- Nonsingleton dimensions of input arrays must match each other. In other words, the corresponding dimensions of arguments `A1,...,An`, must be equal to each other, or equal to one. Whenever a dimension of an input array is singleton (equal to `1`), `arrayfun` uses singleton expansion. The array is virtually replicated along the singleton dimension to match the largest of the other arrays in that dimension. When a dimension of an input array is singleton and the corresponding dimension in another argument array is zero, `arrayfun` virtually diminishes the singleton dimension to `0`.

  Each dimension of the output array `B` is the same size as the largest of the input arrays in that dimension for nonzero size, or zero otherwise. The following code shows how dimensions of size `1` are scaled up or down to match the size of the corresponding dimension in other arguments.

  ```
  R1 = rand(2,5,4,'gpuArray');
  R2 = rand(2,1,4,3,'gpuArray');
  R3 = rand(1,5,4,3,'gpuArray');
  R = arrayfun(@(x,y,z)(x+y.*z),R1,R2,R3);
  size(R)

     2     5     4     3

  R1 = rand(2,2,0,4,'gpuArray');
  R2 = rand(2,1,1,4,'gpuArray');
  R = arrayfun(@plus,R1,R2);
  size(R)

     2     2     0     4
  ```

- Because the operations supported by `arrayfun` are strictly element-wise, and each computation of each element is performed independently of the others, certain restrictions are imposed:

  - Input and output arrays cannot change shape or size.
  - Array-creation functions such as `rand` do not support size specifications. Arrays of random numbers have independent streams for each element.

- Like `arrayfun` in MATLAB, matrix exponential power, multiplication, and division (^, *, /, \) perform element-wise calculations only.

- Operations that change the size or shape of the input or output arrays (`cat`, `reshape`, and so on) are not supported.

- Read-only indexing (`subsref`) and access to variables of the parent (outer) function workspace from within nested functions is supported. You can index variables that exist in the function before

the evaluation on the GPU. Assignment or `subsasgn` indexing of these variables from within the nested function is not supported. For an example of the supported usage, see "Stencil Operations on a GPU" on page 9-198.

- Anonymous functions do not have access to their parent function workspace.
- Overloading the supported functions is not allowed.
- The code cannot call scripts.
- There is no `ans` variable to hold unassigned computation results. Make sure to explicitly assign to variables the results of all calculations.
- The following language features are not supported: persistent or global variables, `parfor`, `spmd`, `switch`, and `try/catch`.
- P-code files cannot contain a call to `arrayfun` with gpuArray data.

See Also
`gather` | `gpuArray` | `pagefun`

Topics
"Improve Performance of Element-wise MATLAB® Functions on the GPU using ARRAYFUN" on page 9-169
"Using GPU ARRAYFUN for Monte-Carlo Simulations" on page 9-192

Introduced in R2010b

batch

Run MATLAB script or function on worker

Syntax

```
j = batch(script)
j = batch(myCluster,script)
j = batch(fcn,N,{x1,...,xn})
j = batch(myCluster,fcn,N,{x1,...,xn})
j = batch( ___ ,Name,Value)
```

Description

`j = batch(script)` runs the script file `script` on a worker in the cluster specified by the default cluster profile. (Note: Do not include the `.m` file extension with the script name.) The function returns `j`, a handle to the job object that runs the script. The script file `script` is copied to the worker.

`j = batch(myCluster,script)` is identical to `batch(script)` except that the script runs on a worker in the cluster specified by the cluster object `myCluster`.

`j = batch(fcn,N,{x1,...,xn})` runs the function `fcn` on a worker in the cluster specified by the default cluster profile. The function returns `j`, a handle to the job object that runs the function. The function is evaluated with the given arguments, `x1,...,xn`, and returns `N` output arguments. The function file for `fcn` is copied to the worker. (Note: Do not include the `.m` file extension with the function name argument.)

`j = batch(myCluster,fcn,N,{x1,...,xn})` is identical to `batch(fcn,N,{x1,...,xn})` except that the function runs on a worker in the cluster specified by the cluster object `myCluster`.

`j = batch( ___ ,Name,Value)` specifies options that modify the behavior of a job using one or more name-value pair arguments. These options support batch for functions and scripts, unless otherwise indicated. Use this syntax in addition to any of the input argument combinations in previous syntaxes.

Examples

Run Script as Batch Job

Use `batch` to offload work to a MATLAB worker session that runs in the background. You can continue using MATLAB while computations take place.

Run a script as a batch job by using the `batch` function. By default, `batch` uses your default cluster profile. Check your default cluster profile on the MATLAB **Home** tab, in the **Environment** section, in **Parallel > Select a Default Cluster**. Alternatively, you can specify a cluster profile with the `'Profile'` name-value pair argument.

```
job = batch('myScript');
```

`batch` does not block MATLAB and you can continue working while computations take place.

If you want to block MATLAB until the job finishes, use the `wait` function on the job object.

```
wait(job);
```

By default, MATLAB saves the Command Window output from the batch job to the diary of the job. To retrieve it, use the `diary` function.

```
diary(job)
```

```
--- Start Diary ---
n = 100

--- End Diary ---
```

After the job finishes, fetch the results by using the `load` function.

```
load(job,'x');
plot(x)
```


If you want to load all the variables in the batch job, use the `load` function without arguments.

When you have fetched all the required variables, delete the job object to clean up its data and avoid consuming resources unnecessarily.

```
delete(job);
clear job
```

Note that if you send a script file using `batch`, MATLAB transfers all the workspace variables to the cluster, even if your script does not use them. The data transfer time for a large workspace can be

substantial. As a best practice, convert your script to a function file to avoid this communication overhead. For an example that uses a function, see "Run Batch Job and Access Files from Workers" on page 11-26.

For more advanced options with `batch`, see "Run Batch Job and Access Files from Workers" on page 11-26.

Run Batch Job and Access Files from Workers

You can offload your computations to run in the background by using `batch`. If your code needs access to files, you can use additional options, such as `'AttachedFiles'` or `'AdditionalPaths'`, to make the data accessible. You can close or continue working in MATLAB while computations take place and recover the results later.

Prepare Example

Prepare and copy the supporting files for this example by using the following command.

```
prepareSupportingFiles;
```

Run Batch Job

Create a cluster object using `parcluster`. By default, `parcluster` uses your default cluster profile. Check your default cluster profile on the MATLAB **Home** tab, in the **Environment** section, in **Parallel** > **Select a Default Cluster**.

```
c = parcluster();
```

Place your code inside a function and submit it as a batch job by using `batch`. For an example of a custom function, see the supporting function `myFunction`. Specify the expected number of output arguments and a cell array with inputs to the function.

If your code uses a parallel pool, use the `'Pool'` name-value pair argument to create a parallel pool with the number of workers that you specify. `batch` uses an additional worker to run the function itself.

By default, `batch` changes the initial working directory of the workers to the current folder of the MATLAB client. It can be useful to control the initial working directory in the workers. For example, you might want to control it if your cluster uses a different filesystem, and therefore the paths are different, such as when you submit from a Windows client machine to a Linux cluster.

- To keep the initial working directory of the workers and use their default, set `'CurrentFolder'` to `'.'`.
- To change the initial working directory, set `'CurrentFolder'` to a folder of your choice.

This example uses a parallel pool with three workers and chooses a temporary location for the initial working directory.

```
job = batch(c,@myFunction,1,{}, ...
    'Pool',3, ...
    'CurrentFolder',tempdir);
```

`batch` offloads the computations in your function to a parallel worker, so you can continue working in MATLAB while computations take place.

If you want to block MATLAB until the job completes, use the `wait` function on the job object.

```
wait(job);
```

To retrieve the results, use `fetchOutputs` on the job object.

If your code has an error, then `fetchOutputs` throws an error. You can access error information by checking the `Error` property of `Task` objects in the job. In this example, the code depends on a file that the workers cannot find.

```
getReport(job.Tasks(1).Error)

ans =
    'Error using myFunction (line 4)
     Unable to read file 'mydata2.dat'. No such file or directory.'
```

Access Files from Workers

By default, `batch` automatically analyzes your code and transfers required files to the workers. In some cases, you must explicitly transfer those files--for example, when you determine the name of a file at runtime.

In this example, `myFunction` accesses the supporting file `mydata.dat`, which `batch` automatically detects and transfers. The function also accesses `mydata1.dat`, but it resolves the name of the file at runtime, so the automatic dependency analysis does not detect it.

```
type myFunction.m

function X = myFunction()
    A = load("mydata.dat");
    X = zeros(flip(size(A)));
    parfor i = 1:3
        B = load("mydata"+i+".dat");
        X = X + A\B;
    end
end
```

If the data is in a location that the workers can access, you can use the name-value pair argument `'AdditionalPaths'` to specify the location. `'AdditionalPaths'` adds this path to the MATLAB search path of the workers and makes the data visible to them.

```
pathToData = pwd;
job(2) = batch(c,@myFunction,1,{}, ...
    'Pool',3, ...
    'CurrentFolder',tempdir, ...
    'AdditionalPaths',pathToData);
wait(job(2));
```

If the data is in a location that the workers cannot access, you can transfer files to the workers by using the `'AttachedFiles'` name-value pair argument.

```
job(3) = batch(c,@myFunction,1,{}, ...
    'Pool',3, ...
    'CurrentFolder',tempdir, ...
    'AttachedFiles',"mydata"+string(1:3)+".dat");
```

Find Existing Job

You can close MATLAB after job submission and retrieve the results later. Before you close MATLAB, make a note of the job ID.

```
job3ID = job(3).ID
```

```
job3ID = 19
```

When you open MATLAB again, you can find the job by using the `findJob` function.

```
job(3) = findJob(c,'ID',job3ID);
wait(job(3));
```

Alternatively, you can use the Job Monitor to track your job. You can open it from the MATLAB **Home** tab, in the **Environment** section, in **Parallel > Monitor Jobs**.

Retrieve Results and Clean Up Data

To retrieve the results of a batch job, use the `fetchOutputs` function. `fetchOutputs` returns a cell array with the outputs of the function run with `batch`.

```
X = fetchOutputs(job(3))
```

```
X = 1×1 cell array
    {40×207 double}
```

When you have retrieved all the required outputs and do not need the job object anymore, delete it to clean up its data and avoid consuming resources unnecessarily.

```
delete(job)
clear job
```

Input Arguments

`script` — MATLAB script
character vector | string

MATLAB script to be evaluated by the worker, specified as a character vector or string.

Example: `batch('aScript');`

Data Types: `char` | `string`

`myCluster` — Cluster
`parallel.Cluster` object

Cluster, specified as a `parallel.Cluster` object that represents cluster compute resources. To create the object, use the `parcluster` function.

Example: `cluster = parcluster; batch(cluster,'aScript');`

Data Types: `parallel.Cluster`

`fcn` — Function to be evaluated by the worker
function handle | character vector

Function to be evaluated by the worker, specified as a function handle or function name.

Example: batch(@myFunction,1,{x,y});

Data Types: char | string | function_handle

N — Number of outputs
nonnegative integer

Number of outputs expected from the evaluated function fcn, specified as a nonnegative integer.

Example: batch(@myFunction,1,{x,y});

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

{x1,...,xn} — Input arguments
cell array

Input arguments to the function fcn, specified as a cell array.

Example: batch(@myFunction,1,{x,y});

Data Types: cell

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: j = batch(@myFunction,1,{x,y},'Pool',3);

Workspace — Workspace on the worker
1-by-1 struct

Workspace on the worker just before the script or function is called, specified as the comma-separated pair consisting of 'Workspace' and a 1-by-1 struct. The field names of the struct define the names of the variables, and the field values are assigned to the workspace variables. By default, this parameter has a field for every variable in the current workspace where batch is executed. This parameter supports only the running of scripts.

Example: workspace.myVar = 5; j = batch('aScript','Workspace',workspace);

Data Types: struct

Profile — Cluster profile
character vector | string

Cluster profile used to identify the cluster, specified as the comma-separated pair consisting of 'Profile' and a character vector or string. If this option is omitted, the default profile is used to identify the cluster and is applied to the job and task properties.

Example: j = batch('aScript','Profile','local');

Data Types: char | string

AdditionalPaths — Paths to add to workers
character vector | string array | cell array of character vectors

Paths to add to the MATLAB search path of the workers before the script or function executes, specified as the comma-separated pair consisting of 'AdditionalPaths' and a character vector, string array, or cell array of character vectors.

The default search path might not be the same on the workers as it is on the client; the path difference could be the result of different current working folders (`cwd`), platforms, or network file system access. Specifying the `'AdditionalPaths'` name-value pair argument helps ensure that workers look for files, such as code files, data files, or model files, in the correct locations.

You can use `'AdditionalPaths'` to access files in a shared file system. Note that path representations can vary depending on the target machines. `'AdditionalPaths'` must be the paths as seen by the machines in the cluster. For example, if `Z:\data` on your local Windows machine is `/network/data` to your Linux cluster, then add the latter to `'AdditionalPaths'`. If you use a datastore, use `'AlternateFileSystemRoots'` instead to deal with other representations. For more information, see "Set Up Datastore for Processing on Different Machines or Clusters".

Note that `AdditionalPaths` only helps to find files when you refer to them using a relative path or file name, and not an absolute path.

Example: `j = batch(@myFunction,1,{x,y},'AdditionalPaths','/network/data/');`

Data Types: `char` | `string` | `cell`

AttachedFiles — Files or folders to transfer
character vector | string array | cell array of character vectors

Files or folders to transfer to the workers, specified as the comma-separated pair consisting of `'AttachedFiles'` and a character vector, string array, or cell array of character vectors.

Example: `j = batch(@myFunction,1,{x,y},'AttachedFiles','myData.dat');`

Data Types: `char` | `string` | `cell`

AutoAddClientPath — Flag to add user-added entries on client path to worker path
true (default) | false

Flag to add user-added entries on the client path to worker paths, specified as the comma-separated pair consisting of `'AutoAddClientPath'` and a logical value.

Example: `j = batch(@myFunction,1,{x,y},'AutoAddClientPath',false);`

Data Types: `logical`

AutoAttachFiles — Flag to enable dependency analysis
true (default) | false

Flag to enable dependency analysis and automatically attach code files to the job, specified as the comma-separated pair consisting of `'AutoAttachFiles'` and a logical value. If you set the value to `true`, the batch script or function is analyzed and the code files that it depends on are automatically transferred to the workers.

Example: `j = batch(@myFunction,1,{x,y},'AutoAttachFiles',true);`

Data Types: `logical`

CurrentFolder — Folder in which the script or function executes
character vector | string

Folder in which the script or function executes, specified as the comma-separated pair consisting of `'CurrentFolder'` and a character vector or string. There is no guarantee that this folder exists on the worker. The default value for this property is the current directory of MATLAB when the `batch` command is executed. If the argument is `'.'`, there is no change in folder before batch execution.

Example: j = batch(@myFunction,1,{x,y},'CurrentFolder','.');

Data Types: `char` | `string`

CaptureDiary — Flag to collect the diary
`true` (default) | `false`

Flag to collect the diary from the function call, specified as the comma-separated pair consisting of `'CaptureDiary'` and a logical value. For information on the collected data, see `diary`.

Example: j = batch('aScript','CaptureDiary',false);

Data Types: `logical`

EnvironmentVariables — Environment variables to copy
character vector | string array | cell array of character vectors

Environment variables to copy from the client session to the workers, specified as the comma-separated pair consisting of `'EnvironmentVariables'` and a character vector, string array, or cell array of character vectors. The names specified here are appended to the `EnvironmentVariables` property specified in the applicable parallel profile to form the complete list of environment variables. Listed variables that are not set are not copied to the workers. These environment variables are set on the workers for the duration of the batch job.

Example: j = batch('aScript','EnvironmentVariables',"MY_ENV_VAR");

Data Types: `char` | `string` | `cell`

Pool — Number of workers to make into a parallel pool
`0` (default) | nonnegative integer | 2-element vector of nonnegative integers

Number of workers to make into a parallel pool, specified as the comma-separated pair consisting of `'Pool'` and either:

- A nonnegative integer.
- A 2-element vector of nonnegative integers, which is interpreted as a range. The size of the resulting parallel pool is as large as possible in the range requested.

In addition, note that `batch` uses another worker to run the batch job itself.

The script or function uses this pool to execution statements such as `parfor` and `spmd` that are inside the batch code. Because the pool requires N workers in addition to the worker running the batch, the cluster must have at least N+1 workers available. You do not need a parallel pool already running to execute `batch`, and the new pool that batch creates is not related to a pool you might already have open. For more information, see "Run a Batch Job with a Parallel Pool" on page 1-9.

If you use the default value, `0`, the script or function runs on only a single worker and not on a parallel pool.

Example: j = batch(@myFunction,1,{x,y},'Pool',4);

Example: j = batch(@myFunction,1,{x,y},'Pool',[2,6]);

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

j — Job
parallel.Job

Job that runs the script or function, returned as a `parallel.Job` object.

Example: j = batch('aScript');

Data Types: parallel.Job

Tips

To view the status or track the progress of a batch job, use the Job Monitor, as described in "Job Monitor" on page 5-24. You can also use the Job Monitor to retrieve a job object for a batch job that was created in a different session, or for a batch job that was created without returning a job object from the `batch` call.

Delete any batch jobs you no longer need to avoid consuming cluster storage resources unnecessarily.

See Also
delete | diary | fetchOutputs (Job) | findJob | load | wait

Introduced in R2008a

bsxfun

Binary singleton expansion function for gpuArray

Syntax

```
C = bsxfun(FUN,A,B)
```

Description

Note

- The function `arrayfun` offers improved functionality compared to `bsxfun`. `arrayfun` is recommended.

- This function behaves similarly to the MATLAB function `bsxfun`, except that the evaluation of the function happens on the GPU, not on the CPU. Any required data not already on the GPU is moved to GPU memory. The MATLAB function passed in for evaluation is compiled and then executed on the GPU. All output arguments are returned as gpuArray objects. You can retrieve gpuArray data using the `gather` function.

`C = bsxfun(FUN,A,B)` applies the element-by-element binary operation specified by `FUN` to arrays A and B, with singleton expansion enabled.

Examples

Deviation of Matrix Elements from Column Mean

Use `bsxfun` with a matrix to subtract the mean of each column from all elements in that column. Then normalize by the standard deviation of each column.

```
A = rand(4,'gpuArray');
B = bsxfun(@minus,A,mean(A));
C = bsxfun(@rdivide,B,std(B))
```

Evaluate Combinations of Inputs

You can use `bsxfun` to evaluate a function for different combinations of inputs.

```
A = rand(4,'gpuArray');
B = bsxfun(@minus,A,mean(A));
C = bsxfun(@rdivide,B,std(B))

C =

   -1.2957   -1.1587   -0.8727    0.2132
   -0.2071    0.9960    0.3272   -1.2763
```

```
    0.4786    0.6523   -0.7228    1.1482
    1.0243   -0.4896    1.2684   -0.0851
```

Create a function handle that represents the function $f(a,b) = 1 - ae^{-b}$. Use `bsxfun` to apply the function to vectors `a` and `b`. `bsxfun` uses singleton expansion to expand the vectors into matrices and evaluates the function with all permutations of the input variables.

```
a = gpuArray(1:7);
b = gpuArray(pi*[0 1/4 1/2 3/4 1 5/4 6/4 7/4 2]).';
fun = @(a,b) 1 - a.*exp(-b);
c = bsxfun(fun,a,b)

c =

         0   -1.0000   -2.0000   -3.0000   -4.0000   -5.0000   -6.0000
    0.5441    0.0881   -0.3678   -0.8238   -1.2797   -1.7356   -2.1916
    0.7921    0.5842    0.3764    0.1685   -0.0394   -0.2473   -0.4552
    0.9052    0.8104    0.7157    0.6209    0.5261    0.4313    0.3365
    0.9568    0.9136    0.8704    0.8271    0.7839    0.7407    0.6975
    0.9803    0.9606    0.9409    0.9212    0.9015    0.8818    0.8621
    0.9910    0.9820    0.9731    0.9641    0.9551    0.9461    0.9371
    0.9959    0.9918    0.9877    0.9836    0.9795    0.9754    0.9713
    0.9981    0.9963    0.9944    0.9925    0.9907    0.9888    0.9869
```

Input Arguments

FUN — Binary function to apply
function handle

Function to apply to the elements of the input arrays, specified as a function handle. FUN must be a handle to a supported element-wise function, or an element-wise function written in the MATLAB language that uses supported functions and syntax. Fun must return scalar values. For each output argument, FUN must return values of the same class each time it is called.

FUN must be a handle to a function that is written in the MATLAB language. You cannot specify FUN as a handle to a MEX-function.

FUN can contain the following built-in MATLAB functions and operators.

abs	csch	log2	sin	Scalar expansion versions of the following:
and	double	log10	single	
acos	eps	log1p	sinh	
acosh	eq	logical	sqrt	*
acot	erf	lt	tan	/
acoth	erfc	max	tanh	\
acsc	erfcinv	min	times	^
acsch	erfcx	minus	true	
asec	erfinv	mod	uint8	Branching instructions:
asech	exp	NaN	uint16	
asin	expm1	ne	uint32	break
asinh	false	not	uint64	continue
atan	fix	ones	xor	else
atan2	floor	or	zeros	elseif
atanh	gamma	pi		for
beta	gammaln	plus	+	if
betaln	ge	pow2	-	return
bitand	gt	power	.*	while
bitcmp	hypot	rand	./	
bitget	imag	randi	.\	
bitor	Inf	randn	.^	
bitset	int8	rdivide	==	
bitshift	int16	real	~=	
bitxor	int32	reallog	<	
cast	int64	realmax	<=	
ceil	intmax	realmin	>	
complex	intmin	realpow	>=	
conj	isfinite	realsqrt	&	
cos	isinf	rem	|	
cosh	isnan	round	~	
cot	ldivide	sec	&&	
coth	le	sech	||	
csc	log	sign		

Functions that create arrays (such as `Inf`, `NaN`, `ones`, `rand`, `randi`, `randn`, and `zeros`) do not support size specifications as input arguments. Instead, the size of the generated array is determined by the size of the input variables to your functions. Enough array elements are generated to satisfy the needs of your input or output variables. You can specify the data type using both class and `"like"` syntaxes. The following examples show supported syntaxes for array-creation functions:

```
a = rand;
b = ones();
c = zeros("like", x);
d = Inf("single");
e = randi([0 9], "uint32");
```

When you use `rand`, `randi`, and `randn` to generate random numbers within FUN, each element is generated from a different substream. For more information about generating random numbers on the GPU, see "Random Number Streams on a GPU" on page 8-6.

A,B — Input arrays
scalars | vectors | matrices | multidimensional arrays

Input arrays, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must have compatible sizes. For more information, see "Compatible Array Sizes for Basic Operations". Whenever a dimension of A or B is singleton (equal to one), `bsxfun` virtually replicates the array along that dimension to match the other array. In the case where a dimension of A or B is singleton,

and the corresponding dimension in the other array is zero, `bsxfun` virtually diminishes the singleton dimension to zero.

At least one of the inputs must be a gpuArray. Each array that is stored on CPU memory is converted to a gpuArray before the function is evaluated. If you plan to make several calls to `bsxfun` with the same array, it is more efficient to convert that array to a gpuArray.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Output Arguments

C — Output array
scalar | vector | matrix | multidimensional array

Output array, returned as a scalar, vector, matrix, or multidimensional array, depending on the sizes of A and B. C is returned as a gpuArray.

Tips

- The first time you call `bsxfun` to run a particular function on the GPU, there is some overhead time to set up the function for GPU execution. Subsequent calls of `bsxfun` with the same function can run faster.

- Nonsingleton dimensions of input arrays must match each other. In other words, the corresponding dimensions of arguments A, B, etc., must be equal to each other, or equal to one. Whenever a dimension of an input array is singleton (equal to 1), `bsxfun` uses singleton expansion. The array is replicated along the singleton dimension to match the largest of the other arrays in that dimension. When a dimension of an input array is singleton and the corresponding dimension in another argument array is zero, `bsxfun` virtually diminishes the singleton dimension to 0.

  Each dimension of the output array C is the same size as the largest of the input arrays in that dimension for nonzero size, or zero otherwise. The following code shows how dimensions of size 1 are scaled up or down to match the size of the corresponding dimension in other arguments.

  ```
  R1 = rand(2,5,4,'gpuArray');
  R2 = rand(2,1,4,3,'gpuArray');
  R = bsxfun(@plus,R1,R2);
  size(R)

     2     5     4     3

  R1 = rand(2,2,0,4,'gpuArray');
  R2 = rand(2,1,1,4,'gpuArray');
  R = bsxfun(@plus,R1,R2);
  size(R)

     2     2     0     4
  ```

- Because the operations supported by `bsxfun` are strictly element-wise, and each computation of each element is performed independently of the others, certain restrictions are imposed:

  - Input and output arrays cannot change shape or size.

- Functions such as `rand` do not support size specifications. Arrays of random numbers have independent streams for each element.
- Like `bsxfun` in MATLAB, matrix exponential power, multiplication, and division (`^`, `*`, `/`, `\`) perform element-wise calculations only.
- Operations that change the size or shape of the input or output arrays (`cat`, `reshape`, and so on), are not supported.
- Read-only indexing (`subsref`) and access to variables of the parent (outer) function workspace from within nested functions is supported. You can index variables that exist in the function before the evaluation on the GPU. Assignment or `subsasgn` indexing of these variables from within the nested function is not supported. For an example of the supported usage, see "Stencil Operations on a GPU" on page 9-198
- Anonymous functions do not have access to their parent function workspace.
- Overloading the supported functions is not allowed.
- The code cannot call scripts.
- There is no `ans` variable to hold unassigned computation results. Make sure to explicitly assign to variables the results of all calculations.
- The following language features are not supported: persistent or global variables, `parfor`, `spmd`, `switch`, and `try/catch`.
- P-code files cannot contain a call to `bsxfun` with gpuArray data.

See Also
`arrayfun` | `gather` | `gpuArray` | `pagefun`

Introduced in R2012a

cancel

Cancel job or task

Syntax

```
cancel(t)
cancel(j)
```

Arguments

t	Pending or running task to cancel.
j	Pending, running, or queued job to cancel.

Description

`cancel(t)` stops the task object, `t`, that is currently in the pending or running state. The task's `State` property is set to `finished`, and no output arguments are returned. An error message stating that the task was canceled is placed in the task object's `ErrorMessage` property, and the worker session running the task is restarted.

`cancel(j)` stops the job object, `j`, that is pending, queued, or running. The job's `State` property is set to `finished`, and a cancel is executed on all tasks in the job that are not in the `finished` state. A job object that has been canceled cannot be started again.

If the job is running from a MATLAB Job Scheduler, any worker sessions that are evaluating tasks belonging to the job object are restarted.

If the specified job or task is already in the `finished` state, no action is taken.

Examples

Cancel a task. Note afterward the task's `State`, `ErrorIdentifier`, and `ErrorMessage` properties.

```
c = parcluster();
job1 = createJob(c);
t = createTask(job1, @rand, 1, {3,3});
cancel(t)
t

 Task with properties:

               ID: 1
            State: finished
         Function: @rand
           Parent: Job 1
        StartTime:
 Running Duration: 0 days 0h 0m 0s

  ErrorIdentifier: parallel:task:UserCancellation
     ErrorMessage: The task was cancelled by user "mylogin" on machine
                   "myhost.mydomain.com".
```

See Also
delete | submit

Introduced before R2006a

cancel

Cancel queued or running future

Syntax

```
cancel(F)
```

Description

`cancel(F)` stops the queued and running futures contained in `F`. No action is taken for finished futures. For each element of `F` that is not already in state `'finished'`, the function sets its `State` property to `'finished'`, and its `Error` property to contain an `MException` indicating that the execution was cancelled.

Some functions and operations are unable to be interrupted by **Ctrl+C**, such as `save`. When you use `parfeval` or `parfevalOnAll` to offload these functions and operations to a worker, `cancel` is unable to stop these futures. To stop the worker executing these functions and operations, interactively use the `delete` function to shut down the worker's pool.

Examples

Query and Cancel `parfeval` Futures

When you use `parfeval` or `parfevalOnAll` to run computations in the background, you create objects called futures. You can use the `State` property of a future to find out whether it is running, queued or finished. You can also use the `FevalQueue` property of a parallel pool to access running and queued futures. To cancel futures, you can use the `cancel` function. In this example, you:

- Use `cancel` to cancel futures directly.
- Check completion errors on completed futures.
- Use the `FevalQueue` property to access futures.

Add Work to Queue

Create a parallel pool `p` with two workers.

```
p = parpool(2);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

When you use `parfeval` to run computations in the background, the function creates and adds a future for each computation to the pool queue. Tasks remain in the queue until a worker becomes idle. When a worker becomes idle, it starts to compute a task if the queue is not empty. When a worker completes a task, the task is removed from the queue and the worker becomes idle.

Use `parfeval` to create an array of futures `f` by instructing workers to execute the function `pause`. Use an argument of `1` for the third future, and an argument of `Inf` for all other futures.

```
for n = 1:5
    if n == 3
        f(n) = parfeval(@pause,0,1);
    else
        f(n) = parfeval(@pause,0,Inf);
    end
end
```

Each use of `parfeval` returns a future object that represents the execution of a function on a worker. Except for the third future, every future will take an infinite amount of time to compute. The future created by `parfeval(@pause,0,Inf)` is an extreme case of a future which can slow down a queue.

Cancel Futures Directly

You can use the `State` property to obtain the status of futures. Construct a cell array of the state of each future in `f`.

```
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'running'}    {'queued'}    {'queued'}    {'queued'}
```

Every task except for the third pauses forever.

Cancel the second future directly with `cancel`.

```
cancel(f(2));
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'running'}    {'queued'}    {'queued'}
```

After you cancel the second future, the third future runs. Wait until the third future completes, then examine the states again.

```
wait(f(3));
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'finished'}    {'running'}    {'queued'}
```

The third future now has the state `'finished'`.

Check Completion Errors

When a future completes, its `State` property becomes `'finished'`. To distinguish between futures which are cancelled and complete normally, use the `Error` property.

```
fprintf("f(2): %s\n", f(2).Error.message)
```

```
f(2): Execution of the future was cancelled.
```

```
fprintf("f(3): %s\n", f(3).Error.message)
```

```
f(3):
```

The code cancels the second future, as the message property indicates. The second future was cancelled, as stated in the `message` property. The third future completes without error, and therefore does not have an error message.

Cancel Futures in Pool Queue

You can use the `FevalQueue` property to access the futures in the pool queue.

```
p.FevalQueue
```

```
ans =
 FevalQueue with properties:

       Number Queued: 1
      Number Running: 2
```

The queue has two properties: `RunningFutures` and `QueuedFutures`. The `RunningFutures` property is an array of futures corresponding to tasks that are currently running.

```
disp(p.FevalQueue.RunningFutures)
```

```
 1x2 FevalFuture array:

        ID              State  FinishDateTime  Function  Error
      --------------------------------------------------------
   1    12              running                @pause
   2    15              running                @pause
```

The `QueuedFutures` property is an array of futures corresponding to tasks that are currently queued and not running.

```
disp(p.FevalQueue.QueuedFutures)
```

```
 FevalFuture with properties:

                  ID: 16
            Function: @pause
      CreateDateTime: 15-Jul-2020 17:29:37
       StartDateTime:
    Running Duration: 0 days 0h 0m 0s
               State: queued
               Error: none
```

You can cancel a single future or an array of futures. Cancel all the futures in `QueuedFutures`.

```
cancel(p.FevalQueue.QueuedFutures);
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'finished'}    {'running'}    {'finished'}
```

`RunningFutures` and `QueuedFutures` are sorted from newest to oldest, regardless of whether `f` is in order from newest to oldest. Each future has a unique `ID` property for the lifetime of the client. Check the `ID` property of each of the futures in `f`.

```
disp(f)
```

```
1x5 FevalFuture array:

        ID              State        FinishDateTime  Function  Error
    ----------------------------------------------------------------
  1   12              running                        @pause
  2   13  finished (unread)  15-Jul-2020 17:29:37    @pause    Error
  3   14  finished (unread)  15-Jul-2020 17:29:39    @pause
  4   15              running                        @pause
  5   16  finished (unread)  15-Jul-2020 17:29:39    @pause    Error
```

Compare the result against the `ID` property of each of the `RunningFutures`.

```
for j = 1:length(p.FevalQueue.RunningFutures)
    rf = p.FevalQueue.RunningFutures(j);
    fprintf("p.FevalQueue.RunningFutures(%i): ID = %i\n", j, rf.ID)
end
```

```
p.FevalQueue.RunningFutures(1): ID = 12
p.FevalQueue.RunningFutures(2): ID = 15
```

Here, `RunningFutures` is an array containing `f(1)` and `f(4)`. If you cancel `RunningFutures(2)`, you cancel the fourth future `f(4)`.

Sometimes, futures are not available in the workspace, for example, if you execute the same piece of code twice before it finishes, or if you use `parfeval` in a function. You can cancel futures that are not available in the workspace.

Clear `f` from the workspace.

```
clear f
```

You can use `RunningFutures` and `QueuedFutures` to access futures that have not yet completed. Use `RunningFutures` to cancel `f(4)`.

```
rf2 = p.FevalQueue.RunningFutures(2);
cancel(rf2)
rf2.State
```

```
ans =
'finished'
```

To cancel all the futures still in the queue, use the following code.

```
cancel([p.FevalQueue.RunningFutures p.FevalQueue.QueuedFutures])
```

Cancel Array of Futures

Run a function several times until a satisfactory result is found. In this case, the array of futures F is cancelled when a result is greater than 0.95.

```
N = 100;
for idx = N:-1:1
    F(idx) = parfeval(@rand,1); % Create a random scalar
end
result = NaN; % No result yet.
for idx = 1:N
```

```
    [~, thisResult] = fetchNext(F);
    if thisResult > 0.95
        result = thisResult;
        % Have all the results needed, so break
        break;
    end
end
% With required result, cancel any remaining futures
cancel(F)
result
```

Input Arguments

F — Future
parallel.FevalFuture

Future object, returned as a `parallel.FevalFuture`, that represents the execution of `fcn` on a parallel worker and holds its results. Use `fetchOutputs` or `fetchNext` to collect the results.

See Also

afterAll | afterEach | fetchNext | fetchOutputs | isequal | parfeval | parfevalOnAll

Introduced in R2013b

changePassword

Prompt user to change MATLAB Job Scheduler password

Syntax

```
changePassword(mjs)
changePassword(mjs,username)
```

Arguments

`mjs`	MATLAB Job Scheduler cluster object on which password is changing
`username`	Character vector identifying the user whose password is changing

Description

`changePassword(mjs)` prompts you to change your password as the current user on the MATLAB Job Scheduler cluster represented by cluster object `mjs`. (Use the `parcluster` function to create a cluster object.) In the dialog box that opens, you must enter your current password as well as the new password.

`changePassword(mjs,username)` prompts you as the MATLAB Job Scheduler cluster `admin` user to change the password for another specified user. In the dialog box that opens, you must enter the cluster `admin` password in addition to the user's new password. This allows the cluster `admin` to reset a password for a user who is not available or has forgotten the password. (Note: The cluster `admin` account was created when the MATLAB Job Scheduler cluster was started with a security level of 1, 2, or 3.)

Examples

Change Password with a Cluster Profile

Change your password for the MATLAB Job Scheduler cluster identified by a MATLAB Job Scheduler cluster profile called `MyMjsProfile`.

```
mjs = parcluster('MyMjsProfile');
changePassword(mjs)
```

Change Password on a Parallel Pool

Change your password for the MATLAB Job Scheduler cluster on which the parallel pool is running.

```
p = gcp;
mjs = p.Cluster;
changePassword(mjs)
```

Change Password of Another User

Change the password for another user named `john`. The `admin` user can access the MATLAB Job Scheduler from a different session of MATLAB to do this, or change the `Username` property of the cluster object within `john`'s MATLAB client session.

```
mjs = parcluster('MyMjsProfile');
mjs.Username = 'admin'      % Generates prompt for admin user password.
changePassword(mjs,'john') % Generates prompt for both passwords.
```
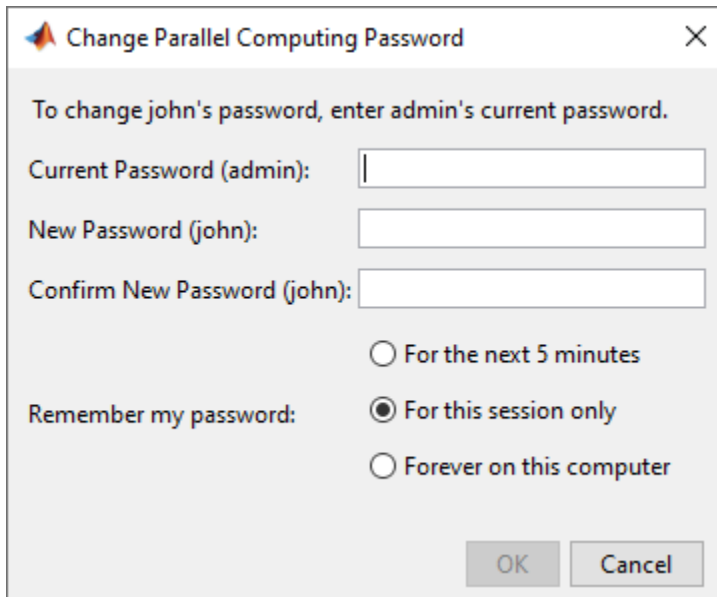

At this point, the `admin` user might want to set the session user back to `john`.

```
mjs.Username = 'john'  % Prompted again for password.
```

See Also
`logout` | `parcluster` | `startjobmanager`

Topics
"Set MATLAB Job Scheduler Cluster Security" (MATLAB Parallel Server)

Introduced in R2010b

classUnderlying

(Not recommended) Class of elements within gpuArray or distributed array

---

Note `classUnderlying` is not recommended. Use `underlyingType` instead. For more information, see "Compatibility Considerations".

Syntax

```
C = classUnderlying(D)
```

Description

`C = classUnderlying(D)` returns the name of the class of the elements contained within the gpuArray or distributed array `D`. Similar to the MATLAB `class` function, this returns a character vector indicating the class of the data.

Examples

Examine the class of the elements of a gpuArray.

```
N  = 1000;
G8 = ones(1,N,'uint8','gpuArray');
G1 = NaN(1,N,'single','gpuArray');
c8 = classUnderlying(G8)
c1 = classUnderlying(G1)

c8 =

uint8

c1 =

single
```

Examine the class of the elements of a distributed array.

```
N  = 1000;
D8 = ones(1,N,'uint8','distributed');
D1 = NaN(1,N,'single','distributed');
c8 = classUnderlying(D8)
c1 = classUnderlying(D1)

c8 =

uint8

c1 =

single
```

Compatibility Considerations

classUnderlying and isaUnderlying are not recommended
Not recommended starting in R2020b

classUnderlying and isaUnderlying are not recommended. Use underlyingType and isUnderlyingType instead.

See Also

codistributed | distributed | gpuArray | isUnderlyingType | mustBeUnderlyingType | underlyingType

Introduced in R2013b

clear

Remove objects from MATLAB workspace

Syntax

```
clear obj
```

Arguments

obj   An object or an array of objects.

Description

`clear obj` removes `obj` from the MATLAB workspace.

Note Use `parfevalOnAll` instead of `parfor` or `spmd` if you want to use `clear`. This preserves workspace "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49.

Examples

This example creates two job objects on the MATLAB Job Scheduler jm. The variables for these job objects in the MATLAB workspace are `job1` and `job2`. `job1` is copied to a new variable, `job1copy`; then `job1` and `job2` are cleared from the MATLAB workspace. The job objects are then restored to the workspace from the job object's `Jobs` property as `j1` and `j2`, and the first job in the MATLAB Job Scheduler is shown to be identical to `job1copy`, while the second job is not.

```
c = parcluster();
delete(c.Jobs) % Assure there are no jobs
job1 = createJob(c);
job2 = createJob(c);
job1copy = job1;
clear job1 job2;
j1 = c.Jobs(1);
j2 = c.Jobs(2);
isequal (job1copy,j1)

ans =
     1

isequal (job1copy,j2)

ans =
     0
```

Tips

If `obj` references an object in the cluster, it is cleared from the workspace, but it remains in the cluster. You can restore `obj` to the workspace with the `parcluster`, `findJob`, or `findTask` function; or with the `Jobs` or `Tasks` property.

See Also
createJob | createTask | findJob | findTask | parcluster

Introduced before R2006a

codistributed

Create codistributed array from replicated local data

Syntax

```
C = codistributed(X)
C = codistributed(X,codist)
C = codistributed(X,lab,codist)
C = codistributed(C1,codist)
```

Description

`C = codistributed(X)` distributes a replicated array `X` using the default codistributor, creating a codistributed array `C` as a result. `X` must be a replicated array, that is, it must have the same value on all workers. `size(C)` is the same as `size(X)`.

`C = codistributed(X,codist)` distributes a replicated array `X` using the distribution scheme defined by codistributor `codist`. `X` must be a replicated array, namely it must have the same value on all workers. `size(C)` is the same as `size(X)`. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = codistributed(X,lab,codist)` distributes a local array `X` that resides on the worker identified by `lab`, using the codistributor `codist`. Local array `X` must be defined on all workers, but only the value from `lab` is used to construct `C`. `size(C)` is the same as `size(X)`.

`C = codistributed(C1,codist)` accepts an array `C1` that is already codistributed, and redistributes it into `C` according to the distribution scheme defined by the codistributor `codist`. This is the same as calling `C = redistribute(C1,codist)`. If the existing distribution scheme for `C1` is the same as that specified in `codist`, then the result `C` is the same as the input `C1`.

Examples

Create a 1000-by-1000 codistributed array `C1` using the default distribution scheme.

```
spmd
    N = 1000;
    X = magic(N);            % Replicated on every worker
    C1 = codistributed(X); % Partitioned among the workers
end
```

Create a 1000-by-1000 codistributed array `C2`, distributed by rows (over its first dimension).

```
spmd
    N = 1000;
    X = magic(N);
    C2 = codistributed(X,codistributor1d(1));
end
```

Tips

`gather` essentially performs the inverse of `codistributed`.

See Also

"What Is a Datastore?" | codistributor1d | codistributor2dbc | distributed | gather | getLocalPart | globalIndices | redistribute | subsasgn | subsref

codistributed.build

Create codistributed array from distributed data

Syntax

```
D = codistributed.build(L,codist)
D = codistributed.build(L,codist,'noCommunication')
```

Description

`D = codistributed.build(L,codist)` forms a codistributed array with `getLocalPart(D) = L`. The codistributed array `D` is created as if you had combined all copies of the local array `L`. The distribution scheme is specified by `codist`. Global error checking ensures that the local parts conform with the specified distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`D = codistributed.build(L,codist,'noCommunication')` builds a codistributed array, without performing any interworker communications for error checking.

`codist` must be complete, which you can check by calling `codist.isComplete()`. The requirements on the size and structure of the local part `L` depend on the class of `codist`. For the 1-D and 2-D block-cyclic codistributors, `L` must have the same class and sparsity on all workers. Furthermore, the local part `L` must represent the region described by the `globalIndices` method on `codist`.

Examples

Create a codistributed array of size 1001-by-1001 such that column `ii` contains the value `ii`.

```
spmd
    N = 1001;
    globalSize = [N,N];
    % Distribute the matrix over the second dimension (columns),
    % and let the codistributor derive the partition from the
    % global size.
    codistr = codistributor1d(2, ...
                codistributor1d.unsetPartition,globalSize)

    % On 4 workers, codistr.Partition equals [251,250,250,250].
    % Allocate storage for the local part.
    localSize = [N, codistr.Partition(labindex)];
    L = zeros(localSize);

    % Use globalIndices to map the indices of the columns
    % of the local part into the global column indices.
    globalInd = codistr.globalIndices(2);
    % On 4 workers, globalInd has the values:
    % 1:251    on worker 1
    % 252:501  on worker 2
    % 502:751  on worker 3
    % 752:1001 on worker 4
```

```
% Initialize the columns of the local part to
% the correct value.
for localCol = 1:length(globalInd)
    globalCol = globalInd(localCol);
    L(:,localCol) = globalCol;
end
D = codistributed.build(L,codistr)
end
```

See Also
codistributor1d | codistributor2dbc | gather | getLocalPart | globalIndices | redistribute | subsasgn | subsref

Introduced in R2009b

codistributed.cell

Create codistributed cell array

Syntax

```
C = codistributed.cell(n)
C = codistributed.cell(m,n,p,...)
C = codistributed.cell([m,n,p,...])
C = cell(n,codist)
C = cell(m,n,p,...,codist)
C = cell([m,n,p,...],codist)
```

Description

`C = codistributed.cell(n)` creates an n-by-n codistributed array of underlying class cell, distributing along columns.

`C = codistributed.cell(m,n,p,...)` or `C = codistributed.cell([m,n,p,...])` creates an m-by-n-by-p-by-... codistributed array of underlying class cell, using a default scheme of distributing along the last nonsingleton dimension.

Optional arguments to `codistributed.cell` must be specified after the required arguments, and in the following order:

- `codist` — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.
- `'noCommunication'` — Specifies that no communication is to be performed when constructing the array, skipping some error checking steps.

`C = cell(n,codist)` is the same as `C = codistributed.cell(n, codist)`. You can also use the `'noCommunication'` object with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:

```
spmd
    C = cell(8,codistributor1d());
end
```

`C = cell(m,n,p,...,codist)` and `C = cell([m,n,p,...],codist)` are the same as `C = codistributed.cell(m,n,p,...)` and `C = codistributed.cell([m,n,p,...])`, respectively. You can also use the optional `'noCommunication'` argument with this syntax.

Examples

With four workers,

```
spmd(4)
    C = codistributed.cell(1000);
end
```

creates a 1000-by-1000 distributed cell array C, distributed by its second dimension (columns). Each worker contains a 1000-by-250 local piece of C.

```
spmd(4)
    codist = codistributor1d(2, 1:numlabs);
    C = cell(10, 10, codist);
end
```

creates a 10-by-10 codistributed cell array C, distributed by its columns. Each worker contains a 10-by-labindex local piece of C.

See Also
cell | distributed.cell

Introduced in R2009b

codistributed.colon

Distributed colon operation

Syntax

```
codistributed.colon(a,d,b)
codistributed.colon(a,b)
codistributed.colon( ___ ,codist)
codistributed.colon( ___ ,'noCommunication')
codistributed.colon( ___ ,codist,'noCommunication')
```

Description

`codistributed.colon(a,d,b)` partitions the vector `a:d:b` into `numlabs` contiguous subvectors of equal, or nearly equal length, and creates a codistributed array whose local portion on each worker is the `labindex`-th subvector.

`codistributed.colon(a,b)` uses `d = 1`.

Optional arguments to `codistributed.colon` must be specified after the required arguments, and in the following order:

`codistributed.colon( ___ ,codist)` uses the codistributor object `codist` to specify the distribution scheme of the resulting vector. If omitted, the result is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`codistributed.colon( ___ ,'noCommunication')` or `codistributed.colon( ___ ,codist,'noCommunication')` specifies that no communication is to be performed when constructing the vector, skipping some error checking steps.

Examples

Partition the vector `1:10` into four subvectors among four workers.

```
spmd(4); C = codistributed.colon(1,10), end

Lab 1:
  This worker stores C(1:3).
        LocalPart: [1 2 3]
     Codistributor: [1x1 codistributor1d]
Lab 2:
  This worker stores C(4:6).
        LocalPart: [4 5 6]
     Codistributor: [1x1 codistributor1d]
Lab 3:
  This worker stores C(7:8).
        LocalPart: [7 8]
     Codistributor: [1x1 codistributor1d]
Lab 4:
  This worker stores C(9:10).
```

```
        LocalPart: [9 10]
     Codistributor: [1x1 codistributor1d]
```

See Also
codistributor1d | codistributor2dbc | colon | for

Introduced in R2009b

codistributed.spalloc

Allocate space for sparse codistributed matrix

Syntax

```
SD = codistributed.spalloc(M,N,nzmax)
SD = spalloc(M,N,nzmax,codist)
```

Description

`SD = codistributed.spalloc(M,N,nzmax)` creates an M-by-N all-zero sparse codistributed matrix with room to hold `nzmax` nonzeros.

Optional arguments to `codistributed.spalloc` must be specified after the required arguments, and in the following order:

- `codist` — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. The allocated space for nonzero elements is consistent with the distribution of the matrix among the workers according to the `Partition` of the codistributor.

- `'noCommunication'` — Specifies that no communication is to be performed when constructing the array, skipping some error checking steps. You can also use this argument with `SD = spalloc(M, N, nzmax, codistr)`.

`SD = spalloc(M,N,nzmax,codist)` is the same as `SD = codistributed.spalloc(M,N,nzmax,codist)`. You can also use the optional arguments with this syntax.

Examples

Allocate space for a 1000-by-1000 sparse codistributed matrix with room for up to 2000 nonzero elements. Use the default codistributor. Define several elements of the matrix.

```
spmd  % codistributed array created inside spmd statement
    N = 1000;
    SD = codistributed.spalloc(N, N, 2*N);
    for ii=1:N-1
      SD(ii,ii:ii+1) = [ii ii];
    end
end
```

See Also
distributed.spalloc | spalloc | sparse

Introduced in R2009b

codistributed.speye

Create codistributed sparse identity matrix

Syntax

```
CS = codistributed.speye(n)
CS = codistributed.speye(m,n)
CS = codistributed.speye([m,n])
CS = speye(n,codist)
CS = speye(m,n,codist)
CS = speye([m,n],codist)
```

Description

`CS = codistributed.speye(n)` creates an n-by-n sparse codistributed array of underlying class double.

`CS = codistributed.speye(m,n)` or `CS = codistributed.speye([m,n])` creates an m-by-n sparse codistributed array of underlying class double.

Optional arguments to `codistributed.speye` must be specified after the required arguments, and in the following order:

- `codist` — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

- `'noCommunication'` — Specifies that no interworker communication is to be performed when constructing the array, skipping some error checking steps.

`CS = speye(n,codist)` is the same as `CS = codistributed.speye(n,codist)`. You can also use the optional arguments with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:

```
spmd
    CS = codistributed.speye(8,codistributor1d());
end
```

`CS = speye(m,n,codist)` and `CS = speye([m,n],codist)` are the same as `CS = codistributed.speye(m,n)` and `CS = codistributed.speye([m,n])`, respectively. You can also use the optional arguments with this syntax.

Note To create a sparse codistributed array of underlying class logical, first create an array of underlying class double and then cast it using the `logical` function:

```
CLS = logical(speye(m,n,codistributor1d()))
```

Examples

With four workers,

```
spmd(4)
    CS = speye(1000,codistributor())
end
```

creates a 1000-by-1000 sparse codistributed double array `CS`, distributed by its second dimension (columns). Each worker contains a 1000-by-250 local piece of `CS`.

```
spmd(4)
    codist = codistributor1d(2,1:numlabs);
    CS = speye(10,10,codist);
end
```

creates a 10-by-10 sparse codistributed double array `CS`, distributed by its columns. Each worker contains a 10-by-`labindex` local piece of `CS`.

See Also

`distributed.speye` | `sparse` | `speye`

Introduced in R2009b

codistributed.sprand

Create codistributed sparse array of uniformly distributed pseudo-random values

Syntax

```
CS = codistributed.sprand(m,n,density)
CS = sprand(n,codist)
```

Description

`CS = codistributed.sprand(m,n,density)` creates an m-by-n sparse codistributed array with approximately `density*m*n` uniformly distributed nonzero double entries.

Optional arguments to `codistributed.sprand` must be specified after the required arguments, and in the following order:

- `codist` — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

- `'noCommunication'` — Specifies that no interworker communication is to be performed when constructing the array, skipping some error checking steps.

`CS = sprand(n,codist)` is the same as `CS = codistributed.sprand(n, codist)`. You can also use the optional arguments with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:

```
spmd
    CS = codistributed.sprand(8,8,0.2,codistributor1d());
end
```

Examples

With four workers,

```
spmd(4)
    CS = codistributed.sprand(1000,1000,0.001);
end
```

creates a 1000-by-1000 sparse codistributed double array `CS` with approximately 1000 nonzeros. `CS` is distributed by its second dimension (columns), and each worker contains a 1000-by-250 local piece of `CS`.

```
spmd(4)
    codist = codistributor1d(2,1:numlabs);
    CS = sprand(10,10,0.1,codist);
end
```

creates a 10-by-10 codistributed double array `CS` with approximately 10 nonzeros. `CS` is distributed by its columns, and each worker contains a 10-by-`labindex` local piece of `CS`.

Tips

When you use `sprand` on the workers in the parallel pool, or in an independent or communicating job, each worker sets its random generator seed to a value that depends only on the `labindex` or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

See Also

`distributed.sprandn` | `rand` | `sprand`

Introduced in R2009b

codistributed.sprandn

Create codistributed sparse array of uniformly distributed pseudo-random values

Syntax

```
CS = codistributed.sprandn(m,n,density)
CS = sprandn(n,codist)
```

Description

`CS = codistributed.sprandn(m,n,density)` creates an m-by-n sparse codistributed array with approximately `density*m*n` normally distributed nonzero double entries.

Optional arguments to `codistributed.sprandn` must be specified after the required arguments, and in the following order:

- `codist` — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.
- `'noCommunication'` — Specifies that no interworker communication is to be performed when constructing the array, skipping some error checking steps.

`CS = sprandn(n,codist)` is the same as `CS = codistributed.sprandn(n, codist)`. You can also use the optional arguments with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:

```
spmd
    CS = codistributed.sprandn(8,8,0.2,codistributor1d());
end
```

Examples

With four workers,

```
spmd(4)
    CS = codistributed.sprandn(1000,1000,0.001);
end
```

creates a 1000-by-1000 sparse codistributed double array CS with approximately 1000 nonzeros. CS is distributed by its second dimension (columns), and each worker contains a 1000-by-250 local piece of CS.

```
spmd(4)
    codist = codistributor1d(2,1:numlabs);
    CS = sprandn(10,10,0.1,codist);
end
```

creates a 10-by-10 codistributed double array CS with approximately 10 nonzeros. CS is distributed by its columns, and each worker contains a 10-by-labindex local piece of CS.

Tips

When you use `sprandn` on the workers in the parallel pool, or in an independent or communicating job, each worker sets its random generator seed to a value that depends only on the `labindex` or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

See Also

`codistributed.speye` | `codistributed.sprand` | `distributed.sprandn` | `rand` | `randn` | `sparse` | `sprandn`

Introduced in R2009b

codistributor

Create codistributor object for codistributed arrays

Syntax

```
codist = codistributor()
codist = codistributor('1d')
codist = codistributor('1d',dim)
codist = codistributor('1d',dim,part)
codist = codistributor('2dbc')
codist = codistributor('2dbc',lbgrid)
codist = codistributor('2dbc',lbgrid,blksize)
```

Description

There are two schemes for distributing arrays. The scheme denoted by the character vector `'1d'` distributes an array along a single specified subscript, the distribution dimension, in a noncyclic, partitioned manner. The scheme denoted by `'2dbc'`, employed by the parallel matrix computation software ScaLAPACK, applies only to two-dimensional arrays, and varies both subscripts over a rectangular computational grid of labs (workers) in a blocked, cyclic manner.

`codist = codistributor()`, with no arguments, returns a default codistributor object with zero-valued or empty parameters, which can then be used as an argument to other functions to indicate that the function is to create a codistributed array if possible with default distribution. For example,

```
Z = zeros(..., codistributor())
R = randn(..., codistributor())
```

`codist = codistributor('1d')` is the same as `codist = codistributor()`.

`codist = codistributor('1d',dim)` also forms a codistributor object with `codist.Dimension = dim` and default partition.

`codist = codistributor('1d',dim,part)` also forms a codistributor object with `codist.Dimension = dim` and `codist.Partition = part`.

`codist = codistributor('2dbc')` forms a 2-D block-cyclic codistributor object. For more information about `'2dbc'` distribution, see "2-Dimensional Distribution" on page 4-12.

`codist = codistributor('2dbc',lbgrid)` forms a 2-D block-cyclic codistributor object with the lab grid defined by `lbgrid` and with default block size.

`codist = codistributor('2dbc',lbgrid,blksize)` forms a 2-D block-cyclic codistributor object with the lab grid defined by `lbgrid` and with a block size defined by `blksize`.

`codist = getCodistributor(D)` returns the codistributor object of codistributed array D.

Examples

On four workers, create a 3-dimensional, 2-by-6-by-4 array with distribution along the second dimension, and partition scheme [1 2 1 2]. In other words, worker 1 contains a 2-by-1-by-4 segment, worker 2 a 2-by-2-by-4 segment, etc.

```
spmd
    dim = 2; % distribution dimension
    codist = codistributor('1d',dim,[1 2 1 2],[2 6 4]);
    if mod(labindex,2)
        L = rand(2,1,4);
    else
        L = rand(2,2,4);
    end
    A = codistributed.build(L,codist)
end
A
```

On four workers, create a 20-by-5 codistributed array A, distributed by rows (over its first dimension) with a uniform partition scheme.

```
spmd
    dim = 1; % distribution dimension
    partn = codistributor1d.defaultPartition(20);
    codist = codistributor('1d',dim,partn,[20 5]);
    L = magic(5) + labindex;
    A = codistributed.build(L,codist)
end
A
```

See Also

codistributed | codistributor1d | codistributor2dbc | getCodistributor | getLocalPart | redistribute

Introduced in R2008b

codistributor1d

Create 1-D codistributor object for codistributed arrays

Syntax

```
codist = codistributor1d()
codist = codistributor1d(dim)
codist = codistributor1d(dim,part)
codist = codistributor1d(dim,part,gsize)
```

Description

The 1-D codistributor distributes arrays along a single, specified distribution dimension, in a noncyclic, partitioned manner.

`codist = codistributor1d()` forms a codistributor1d object using default dimension and partition. The default dimension is the last nonsingleton dimension of the codistributed array. The default partition distributes the array along the default dimension as evenly as possible.

`codist = codistributor1d(dim)` forms a 1-D codistributor object for distribution along the specified dimension: `1` distributes along rows, `2` along columns, etc.

`codist = codistributor1d(dim,part)` forms a 1-D codistributor object for distribution according to the partition vector `part`. For example `C1 = codistributor1d(1,[1,2,3,4])` describes the distribution scheme for an array of ten rows to be codistributed by its first dimension (rows), to four workers, with 1 row to the first, 2 rows to the second, etc.

The resulting codistributor of any of the above syntax is incomplete because its global size is not specified. A codistributor constructed in this manner can be used as an argument to other functions as a template codistributor when creating codistributed arrays.

`codist = codistributor1d(dim,part,gsize)` forms a codistributor object with distribution dimension `dim`, distribution partition `part`, and global size of its codistributed arrays `gsize`. The resulting codistributor object is complete and can be used to build a codistributed array from its local parts with `codistributed.build`. To use a default dimension, specify `codistributor1d.unsetDimension` for that argument; the distribution dimension is derived from `gsize` and is set to the last non-singleton dimension. Similarly, to use a default partition, specify `codistributor1d.unsetPartition` for that argument; the partition is then derived from the default for that global size and distribution dimension.

The local part on worker `labidx` of a codistributed array using such a codistributor is of size `gsize` in all dimensions except `dim`, where the size is `part(labidx)`. The local part has the same class and attributes as the overall codistributed array. Conceptually, the overall global array could be reconstructed by concatenating the various local parts along dimension `dim`.

Examples

Use a codistributor1d object to create an N-by-N matrix of ones, distributed by rows.

```
N = 1000;
spmd
```

```
    codistr = codistributor1d(1); % 1st dimension (rows)
    C = ones(N,codistr);
end
```

Use a fully specified codistributor1d object to create a trivial N-by-N codistributed matrix from its local parts. Then visualize which elements are stored on worker 2.

```
N = 1000;
spmd
    codistr = codistributor1d( ...
                    codistributor1d.unsetDimension, ...
                    codistributor1d.unsetPartition, ...
                    [N,N]);
    myLocalSize = [N,N]; % start with full size on each lab
    % then set myLocalSize to default part of whole array:
    myLocalSize(codistr.Dimension) = codistr.Partition(labindex);
    myLocalPart = labindex*ones(myLocalSize); % arbitrary values
    D = codistributed.build(myLocalPart,codistr);
end
spy(D==2);
```

See Also
codistributed | codistributor2dbc | redistribute

codistributor1d.defaultPartition

Default partition for codistributed array

Syntax

```
P = codistributor1d.defaultPartition(n)
```

Description

`P = codistributor1d.defaultPartition(n)` is a vector with `sum(P) = n` and `length(P) = numlabs`. The first `rem(n,numlabs)` elements of `P` are equal to `ceil(n/numlabs)` and the remaining elements are equal to `floor(n/numlabs)`. This function is the basis for the default distribution of codistributed arrays.

Examples

If `numlabs = 4`, the following code returns the vector `[3 3 2 2]` on all workers:

```
spmd
    P = codistributor1d.defaultPartition(10)
end
```

See Also

codistributed | codistributed.colon | codistributor1d

Introduced in R2009b

codistributor2dbc

Create 2-D block-cyclic codistributor object for codistributed arrays

Syntax

```
codist = codistributor2dbc()
codist = codistributor2dbc(lbgrid)
codist = codistributor2dbc(lbgrid,blksize)
codist = codistributor2dbc(lbgrid,blksize,orient)
codist = codistributor2dbc(lbgrid,blksize,orient,gsize)
```

Description

The 2-D block-cyclic codistributor can be used only for two-dimensional arrays. It distributes arrays along two subscripts over a rectangular computational grid of labs (workers) in a block-cyclic manner. For a complete description of 2-D block-cyclic distribution, default parameters, and the relationship between block size and lab grid, see "2-Dimensional Distribution" on page 4-12. The 2-D block-cyclic codistributor is used by the ScaLAPACK parallel matrix computation software library.

`codist = codistributor2dbc()` forms a 2-D block-cyclic codistributor2dbc codistributor object using default lab grid and block size.

`codist = codistributor2dbc(lbgrid)` forms a 2-D block-cyclic codistributor object using the specified lab grid and default block size. `lbgrid` must be a two-element vector defining the rows and columns of the lab grid, and the rows times columns must equal the number of workers for the codistributed array.

`codist = codistributor2dbc(lbgrid,blksize)` forms a 2-D block-cyclic codistributor object using the specified lab grid and block size.

`codist = codistributor2dbc(lbgrid,blksize,orient)` allows an orientation argument. Valid values for the orientation argument are `'row'` for row orientation, and `'col'` for column orientation of the lab grid. The default is row orientation.

The resulting codistributor of any of the above syntax is incomplete because its global size is not specified. A codistributor constructed this way can be used as an argument to other functions as a template codistributor when creating codistributed arrays.

`codist = codistributor2dbc(lbgrid,blksize,orient,gsize)` forms a codistributor object that distributes arrays with the global size `gsize`. The resulting codistributor object is complete and can therefore be used to build a codistributed array from its local parts with `codistributed.build`. To use the default values for lab grid, block size, and orientation, specify them using `codistributor2dbc.defaultLabGrid`, `codistributor2dbc.defaultBlockSize`, and `codistributor2dbc.defaultOrientation`, respectively.

Examples

Use a codistributor2dbc object to create an N-by-N matrix of ones.

```
N = 1000;
spmd
```

```
    codistr = codistributor2dbc();
    D = ones(N,codistr);
end
```

Use a fully specified codistributor2dbc object to create a trivial N-by-N codistributed matrix from its local parts. Then visualize which elements are stored on worker 2.

```
N = 1000;
spmd
    codistr = codistributor2dbc(...
                codistributor2dbc.defaultLabGrid, ...
                codistributor2dbc.defaultBlockSize, ...
                'row',[N,N]);
    myLocalSize = [length(codistr.globalIndices(1)), ...
                    length(codistr.globalIndices(2))];
    myLocalPart = labindex*ones(myLocalSize);
    D = codistributed.build(myLocalPart,codistr);
end
spy(D==2);
```

See Also
codistributed | codistributor1d | getLocalPart | redistribute

codistributor2dbc.defaultLabGrid

Default computational grid for 2-D block-cyclic distributed arrays

Syntax

```
grid = codistributor2dbc.defaultLabGrid()
```

Description

`grid = codistributor2dbc.defaultLabGrid()` returns a vector, `grid = [nrow ncol]`, defining a computational grid of `nrow`-by-`ncol` workers in the open parallel pool, such that `numlabs = nrow x ncol`.

The grid defined by `codistributor2dbc.defaultLabGrid` is as close to a square as possible. The following rules define `nrow` and `ncol`:

- If `numlabs` is a perfect square, `nrow = ncol = sqrt(numlabs)`.
- If `numlabs` is an odd power of 2, then `nrow = ncol/2 = sqrt(numlabs/2)`.
- `nrow <= ncol`.
- If `numlabs` is a prime, `nrow = 1`, `ncol = numlabs`.
- `nrow` is the greatest integer less than or equal to `sqrt(numlabs)` for which `ncol = numlabs/nrow` is also an integer.

Examples

View the computational grid layout of the default distribution scheme for the open parallel pool.

```
spmd
    grid = codistributor2dbc.defaultLabGrid
end
```

See Also
codistributed | codistributor2dbc | numlabs

Composite

Create Composite object

Syntax

```
C = Composite()
C = Composite(nlabs)
```

Description

`C = Composite()` creates a Composite object on the client using workers from the parallel pool. The actual number of workers referenced by this Composite object depends on the size of the pool and any existing Composite objects. Generally, you should construct Composite objects outside any `spmd` statement.

`C = Composite(nlabs)` creates a Composite object on the parallel pool set that matches the specified constraint. `nlabs` must be a vector of length 1 or 2, containing integers or `Inf`. If `nlabs` is of length 1, it specifies the exact number of workers to use. If `nlabs` is of size 2, it specifies the minimum and maximum number of workers to use. The actual number of workers used is the maximum number of workers compatible with the size of the parallel pool, and with other existing Composite objects. An error is thrown if the constraints on the number of workers cannot be met.

A Composite object has one entry for each lab; initially each entry contains no data. Use either indexing or an `spmd` block to define values for the entries.

Examples

The following examples all use a local parallel pool of four workers, opened with the statement:

```
p = parpool('local',4);
```

This example shows how to create a Composite object with no defined elements, then assign values using a `for`-loop in the client.

```
c = Composite();  % One element per worker in the pool
for w = 1:length(c)
    c{w} = 0;    % Value stored on each worker
end
```

This example shows how to assign Composite elements in an `spmd` block.

```
c = Composite();
spmd
    c = 0;    % Value stored on each worker
end
```

This example shows how to assign the elements of a Composite with a value from each worker.

```
c = Composite();
spmd
    c = labindex;
```

```
end
c{:}

    1


    2


    3


    4
```

This example shows how to use a distributed array vector to set the values of a Composite.

```
d = distributed([3 1 4 2]); % One integer per worker
spmd
    c = getLocalPart(d);    % Unique value on each worker
end
c{:}

    3


    1


    4


    2
```

Tips

* A Composite is created on the workers of the existing parallel pool. If no pool exists, `Composite` starts a new parallel pool, unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `Composite` cannot start one, the result is a 1-by-1 Composite in the client workspace.

See Also

`parallel.pool.Constant` | `parpool` | `spmd`

createCommunicatingJob

Create communicating job on cluster

Syntax

```
job = createCommunicatingJob(cluster)
job = createCommunicatingJob(...,'p1',v1,'p2',v2,...)
job = createCommunicatingJob(...,'Type','pool',...)
job = createCommunicatingJob(...,'Type','spmd',...)
job = createCommunicatingJob(...,'Profile','profileName',...)
```

Description

`job = createCommunicatingJob(cluster)` creates a communicating job object for the identified cluster.

`job = createCommunicatingJob(...,'p1',v1,'p2',v2,...)` creates a communicating job object with the specified property values. For a listing of the valid properties of the created object, see the `parallel.Job` object reference page. The property name must be a character vector, with the value being the appropriate type for that property. In most cases, the values specified in these property-value pairs override the values in the profile. But when you specify `AttachedFiles` or `AdditionalPaths` at the time of creating a job, the settings are combined with those specified in the applicable profile. If an invalid property name or property value is specified, the object will not be created.

`job = createCommunicatingJob(...,'Type','pool',...)` creates a communicating job of type `'pool'`. This is the default if `'Type'` is not specified. A `'pool'` job runs the specified task function with a parallel pool available to run the body of `parfor` loops or `spmd` blocks. Note that only one worker runs the task function, and the rest of the workers in the cluster form the parallel pool. So on a cluster of N workers for a `'pool'` type job, only N-1 workers form the actual pool that performs the `spmd` and `parfor` code found within the task function.

`job = createCommunicatingJob(...,'Type','spmd',...)` creates a communicating job of type `'spmd'`, where the specified task function runs simultaneously on all workers, and `lab*` functions can be used for communication between workers.

`job = createCommunicatingJob(...,'Profile','profileName',...)` creates a communicating job object with the property values specified in the profile `'profileName'`. If no profile is specified and the cluster object has a value specified in its `'Profile'` property, the cluster's profile is automatically applied.

Examples

Example 11.1. Pool Type Communicating Job

Consider the function `'myFunction'` which uses a `parfor` loop:

```
function result = myFunction(N)
    result = 0;
    parfor ii=1:N
```

```
        result = result + max(eig(rand(ii)));
    end
end
```

Create a communicating job object to evaluate `myFunction` on the default cluster:

```
myCluster = parcluster;
j = createCommunicatingJob(myCluster,'Type','pool');
```

Add the task to the job, supplying an input argument:

```
createTask(j, @myFunction, 1, {100});
```

Set the number of workers required for parallel execution:

```
j.NumWorkersRange = [5 10];
```

Run the job.

```
submit(j);
```

Wait for the job to finish and retrieve its results:

```
wait(j)
out = fetchOutputs(j)
```

Delete the job from the cluster.

```
delete(j);
```

See Also
createJob | createTask | findJob | parcluster | recreate | submit

Introduced in R2012a

createJob

Create independent job on cluster

Syntax

```
obj = createJob(cluster)
obj = createJob(...,'p1',v1,'p2',v2,...)
job = createJob(...,'Profile','profileName',...)
```

Arguments

obj	The job object.
cluster	The cluster object created by `parcluster`.
p1, *p2*	Object properties configured at object creation.
v1, v2	Initial values for corresponding object properties.

Description

`obj = createJob(cluster)` creates an independent job object for the identified cluster.

The job's data is stored in the location specified by the cluster's `JobStorageLocation` property.

`obj = createJob(...,'p1',v1,'p2',v2,...)` creates a job object with the specified property values. For a listing of the valid properties of the created object, see the `parallel.Job` object reference page. The property name must be a character vector, with the value being the appropriate type for that property. In most cases, the values specified in these property-value pairs override the values in the profile; but when you specify `AttachedFiles` or `AdditionalPaths` at the time of creating a job, the settings are combined with those specified in the applicable profile. If an invalid property name or property value is specified, the object will not be created.

`job = createJob(...,'Profile','profileName',...)` creates an independent job object with the property values specified in the profile `'profileName'`. If a profile is not specified and the cluster has a value specified in its `'Profile'` property, the cluster's profile is automatically applied. For details about defining and applying profiles, see "Discover Clusters and Use Cluster Profiles" on page 5-11.

Examples

Example 11.2. Create and Run a Basic Job

Construct an independent job object using the default profile.

```
c = parcluster
j = createJob(c);
```

Add tasks to the job.

```
for i = 1:10
    createTask(j,@rand,1,{10});
end
```

Run the job.

```
submit(j);
```

Wait for the job to finish running, and retrieve the job results.

```
wait(j);
out = fetchOutputs(j);
```

Display the random matrix returned from the third task.

```
disp(out{3});
```

Delete the job.

```
delete(j);
```

Example 11.3. Create a Job with Attached Files

Construct an independent job with attached files in addition to those specified in the default profile.

```
c = parcluster
j = createJob(c,'AttachedFiles',...
        {'myapp/folderA','myapp/folderB','myapp/file1.m'});
```

See Also
createCommunicatingJob | createTask | findJob | parcluster | recreate | submit

Introduced before R2006a

createTask

Create new task in job

Syntax

```
t = createTask(j, F, N, {inputargs})
t = createTask(j, F, N, {C1,...,Cm})
t = createTask(..., 'p1',v1,'p2',v2,...)
t = createTask(...,'Profile', 'ProfileName',...)
```

Arguments

t	Task object or vector of task objects.
j	The job that the task object is created in.
F	A handle to the function that is called when the task is evaluated, or an array of function handles.
N	The number of output arguments to be returned from execution of the task function. This is a double or array of doubles.
{inputargs}	A row cell array specifying the input arguments to be passed to the function F. Each element in the cell array will be passed as a separate input argument. If this is a cell array of cell arrays, a task is created for each cell array.
{C1,...,Cm}	Cell array of cell arrays defining input arguments to each of m tasks.
p1, *p2*	Task object properties configured at object creation.
v1, v2	Initial values for corresponding task object properties.

Description

t = createTask(j, F, N, {inputargs}) creates a new task object in job j, and returns a reference, t, to the added task object. This task evaluates the function specified by a function handle or function name F, with the given input arguments {inputargs}, returning N output arguments.

t = createTask(j, F, N, {C1,...,Cm}) uses a cell array of m cell arrays to create m task objects in job j, and returns a vector, t, of references to the new task objects. Each task evaluates the function specified by a function handle or function name F. The cell array C1 provides the input arguments to the first task, C2 to the second task, and so on, so that there is one task per cell array. Each task returns N output arguments. If F is a cell array, each element of F specifies a function for each task in the vector; it must have m elements. If N is an array of doubles, each element specifies the number of output arguments for each task in the vector. Multidimensional matrices of inputs F, N and {C1,...,Cm} are supported; if a cell array is used for F, or a double array for N, its dimensions must match those of the input arguments cell array of cell arrays. The output t will be a vector with the same number of elements as {C1,...,Cm}. Note that because a communicating job has only one task, this form of vectorized task creation is not appropriate for such jobs.

t = createTask(..., 'p1',v1,'p2',v2,...) adds a task object with the specified property values. For a listing of the valid properties of the created object, see the `parallel.Task` object

reference page. The property name must be a character vector, with the value being the appropriate type for that property. The values specified in these property-value pairs override the values in the profile. If an invalid property name or property value is specified, the object will not be created.

`t = createTask(...,'Profile', 'ProfileName',...)` creates a task object with the property values specified in the cluster profile `ProfileName`. For details about defining and applying cluster profiles, see "Discover Clusters and Use Cluster Profiles" on page 5-11.

Examples

Create a Job with One Task

Create a job object.

```
c = parcluster(); % Use default profile
j = createJob(c);
```

Add a task object which generates a 10-by-10 random matrix.

```
t = createTask(j, @rand, 1, {10,10});
```

Run the job.

```
submit(j);
```

Wait for the job to finish running, and get the output from the task evaluation.

```
wait(j);
taskoutput = fetchOutputs(j);
```

Show the 10-by-10 random matrix.

```
disp(taskoutput{1});
```

Create a Job with Three Tasks

This example creates a job with three tasks, each of which generates a 10-by-10 random matrix.

```
c = parcluster(); % Use default profile
j = createJob(c);
t = createTask(j, @rand, 1, {{10,10} {10,10} {10,10}});
```

Create a Task with Different Property Values

This example creates a task that captures the worker diary, regardless of the setting in the profile.

```
c = parcluster(); % Use default profile
j = createJob(c);
t = createTask(j,@rand,1,{10,10},'CaptureDiary',true);
```

Create Single Task with All Cell Array Inputs

Create a job object on the default cluster.

```
c = parcluster;
job = createJob(c);
```

To create a single task with all cell arrays as its input arguments, use the {C1} syntax of
createTask. For example, to create a task that runs strjoin({'1','1','2'},{'+','='}), use
the following code.

```
task = createTask(job,@strjoin,1,{{{'1','1','2'},{'+','='}}});
task.InputArguments{:}
```

```
ans = 1×3 cell
    {'1'}    {'1'}    {'2'}
```

```
ans = 1×2 cell
    {'+'}    {'='}
```

Submit and wait for the job.

```
submit(job);
wait(job);
```

Retrieve the outputs and display them.

```
outputs = fetchOutputs(job);
disp(outputs{1});
```

```
1+1=2
```

If you attempt to use the {inputargs} syntax with {inputargs} = {{'1','1','2'},
{'+','='}}, then createTask uses the {C1,...,Cm} syntax and creates multiple tasks. For
example, the following code incorrectly creates two tasks, one for strjoin('1','1','2') and one
for strjoin('+','=').

```
task = createTask(job,@strjoin,1,{{'1','1','2'},{'+','='}});
```

See Also
createCommunicatingJob | createJob | findTask | recreate

Introduced before R2006a

delete

Package: parallel

Remove job or task object from cluster and memory

Syntax

```
delete(obj)
```

Description

delete(obj) removes the job or task object, obj, from the local MATLAB session, and removes it from the cluster's JobStorageLocation. When the object is deleted, references to it become invalid. Invalid objects should be removed from the workspace with the clear command. If multiple references to an object exist in the workspace, deleting one reference to that object invalidates the remaining references to it. These remaining references should be cleared from the workspace with the clear command.

When you delete a job object, this also deletes all the task objects contained in that job. Any references to those task objects will also be invalid, and you should clear them from the workspace.

If obj is an array of objects and one of the objects cannot be deleted, the other objects in the array are deleted and a warning is returned.

Because its data is lost when you delete an object, delete should be used only after you have retrieved all required output data from the effected object.

Examples

Delete Job

Create a job object using the default profile. Then delete the job.

```
myCluster = parcluster;
j = createJob(myCluster,'Name','myjob');
t = createTask(j,@rand,1,{10});
delete(j);
clear j t
```

Delete All Jobs on Cluster

Delete all jobs on the cluster identified by the profile myProfile.

```
myCluster = parcluster('myProfile');
delete(myCluster.Jobs)
```

Delete All Jobs Depending on State

Use the syntax with multiple outputs of `findJob` to obtain the jobs by state. In this example, delete the pending jobs.

```
myCluster = parcluster;
[pending queued running completed] = findJob(myCluster);
delete(pending);
```

Alternatively, use `findJob` to retrieve the jobs that match a specific state. In this example, find the jobs in state failed and delete them.

```
myCluster = parcluster;
failed = findJob(myCluster,'State','failed');
delete(failed);
```

Input Arguments

obj — Job or task object to delete
`parallel.Job` object | `parallel.Task` object

Job or task object to delete, specified as a `parallel.Job` or `parallel.Task` object. You can create jobs and tasks with `createJob` and `createTask`.

Data Types: `parallel.Job` | `parallel.Task`

See Also
batch | createJob | createTask | findJob | findTask | wait

Introduced in R2012a

delete

Package: `parallel`

Shut down parallel pool

Syntax

`delete(poolobj)`

Description

`delete(poolobj)` shuts down the parallel pool associated with the object `poolobj`, and destroys the communicating job that comprises the pool. Subsequent parallel language features will automatically start a new parallel pool, unless your parallel preferences disable this behavior.

References to the deleted pool object become invalid. Invalid objects should be removed from the workspace with the `clear` command. If multiple references to an object exist in the workspace, deleting one reference to that object invalidates the remaining references to it. These remaining references should be cleared from the workspace with the `clear` command.

Examples

Shut Down Current Parallel Pool

To get the current parallel pool, use the `gcp` function.

`poolobj = gcp('nocreate');`

Shut down the current pool by using the `delete` function.

`delete(poolobj);`

Input Arguments

poolobj — Parallel pool
`parallel.Pool`

Parallel pool to shut down, specified as a `parallel.Pool` object. You can get the current parallel pool with the `gcp` function.

Example: `delete(gcp('nocreate'))`

Data Types: `parallel.Pool`

See Also
`gcp` | `parpool`

Introduced in R2013b

demote

Demote job in cluster queue

Syntax

```
demote(c,job)
```

Arguments

c	Cluster object that contains the job.
job	Job object demoted in the job queue.

Description

demote(c,job) demotes the job object job that is queued in the cluster c.

If job is not the last job in the queue, demote exchanges the position of job and the job that follows it in the queue.

Examples

Create and submit multiple jobs to the MATLAB Job Scheduler identified by the default parallel configuration:

```
c = parcluster();
pause(c) % Prevent submissions from running.

j1 = createJob(c,'Name','Job A');
j2 = createJob(c,'Name','Job B');
j3 = createJob(c,'Name','Job C');
submit(j1);submit(j2);submit(j3);
```

Demote one of the jobs by one position in the queue:

```
demote(c,j2)
```

Examine the new queue sequence:

```
[pjobs,qjobs,rjobs,fjobs] = findJob(c);
get(qjobs,'Name')

    'Job A'
    'Job C'
    'Job B'
```

Tips

After a call to demote or promote, there is no change in the order of job objects contained in the Jobs property of the cluster object. To see the scheduled order of execution for jobs in the queue, use the findJob function in the form [pending queued running finished] = findJob(c).

See Also

createJob | findJob | promote | submit

Introduced before R2006a

diary

Package: parallel

Display or save Command Window text of batch job

Syntax

```
diary(job)
diary(job, 'filename')
```

Arguments

job	Job from which to view Command Window output text.
'filename'	File to append with Command Window output text from batch job

Description

`diary(job)` displays the Command Window output from the batch job in the MATLAB Command Window. The Command Window output will be captured only if the `batch` command included the `'CaptureDiary'` argument with a value of `true`.

`diary(job, 'filename')` causes the Command Window output from the batch job to be appended to the specified file.

The captured Command Window output includes only the output generated by execution of the task function. Output is not captured from code that runs asynchronously from the task.

See Also
batch | diary | load

Introduced in R2008a

distributed

Create distributed array from data in the client workspace or a datastore

Syntax

```
D = distributed(ds)
D = distributed(X)
D = distributed(C,dim)
```

Description

`D = distributed(ds)` creates a distributed array from a `datastore ds`. `D` is a distributed array stored in parts on the workers of the open parallel pool. You operate on the entire array as a single entity, however, workers operate only on their part of the array, and automatically transfer data between themselves when necessary.

To retrieve the distributed array elements from the pool back to an array in the MATLAB workspace, use `gather`.

`D = distributed(X)` creates a distributed array from an array `X`.

Constructing a distributed array from local data this way is appropriate only if the MATLAB client can store the entirety of `X` in its memory. To construct large distributed arrays, use one of the constructor methods such as `ones(___,'distributed')`, `zeros(___,'distributed')`, etc. For a list, see "Constructor" on page 10-13.

If the input argument is already a distributed array, the result is the same as the input.

`D = distributed(C,dim)` creates a distributed array from a Composite array `C`, with the entries of `C` concatenated and distributed along the dimension `dim`. If you omit `dim`, then the first dimension is the distribution dimension.

All entries of the Composite array must have the same class. Dimensions other than the distribution dimension must match.

Examples

Create Distributed Arrays

Create a small array and distribute it.

```
Nsmall = 50;
D1 = distributed(magic(Nsmall));
```

Create a large distributed array directly, using a build method.

```
Nlarge = 1000;
D2 = rand(Nlarge,'distributed');
```

Retrieve elements of a distributed array, and note where the arrays are located by their `Class`.

```
D3 = gather(D2);
whos
```

```
  Name              Size              Bytes  Class

  D1               50x50                733  distributed
  D2             1000x1000              733  distributed
  D3             1000x1000          8000000  double
  Nlarge           1x1                   8  double
  Nsmall           1x1                   8  double
```

Create a Distributed Array from a Datastore

This example shows how to create and load distributed arrays using `datastore`. You first create a datastore using an example data set. This data set is too small to show equal partitioning of the data over the workers. To simulate a large data set, artificially increase the size of the datastore using `repmat`.

```
files = repmat({'airlinesmall.csv'}, 10, 1);
ds = tabularTextDatastore(files);
```

Select the example variables.

```
ds.SelectedVariableNames = {'DepTime','DepDelay'};
ds.TreatAsMissing = 'NA';
```

Create a distributed table by reading the datastore in parallel. Partition the datastore with one partition per worker. Each worker then reads all data from the corresponding partition. The files must be in a shared location accessible from the workers.

```
dt = distributed(ds);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

Finally, display summary information about the distributed table.

```
summary(dt)
```

```
Variables:

    DepTime: 1,235,230×1 double
        Values:

            min             1
            max          2505
            NaNs       23,510

    DepDelay: 1,235,230×1 double
        Values:

            min         -1036
            max          1438
            NaNs       23,510
```

Create a Distributed Array from a Composite Array

Start a parallel pool of workers and create a `Composite` array by using `spmd`.

```
p = parpool("local",4);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).

spmd
C = rand(3,labindex-1);
end
C


C =

   Lab 1: class = double, size = [3  0]
   Lab 2: class = double, size = [3  1]
   Lab 3: class = double, size = [3  2]
   Lab 4: class = double, size = [3  3]
```

To create a distributed array out of the `Composite` array, use the `distributed` function. For this example, distribute the entries along the second dimension.

```
d = distributed(C,2)

d =

    0.6383    0.9730    0.2934    0.3241    0.9401    0.1897
    0.5195    0.7104    0.1558    0.0078    0.3231    0.3685
    0.1398    0.3614    0.3421    0.9383    0.3569    0.5250

spmd
    d
end
Lab 1:

  This worker does not store any elements of d.

Lab 2:

  This worker stores d(:,1).

        LocalPart: [3x1 double]
    Codistributor: [1x1 codistributor1d]

Lab 3:

  This worker stores d(:,2:3).

        LocalPart: [3x2 double]
    Codistributor: [1x1 codistributor1d]

Lab 4:

  This worker stores d(:,4:6).

        LocalPart: [3x3 double]
    Codistributor: [1x1 codistributor1d]
```

When you are finished with the computations, delete the parallel pool.

```
delete(p);
```

Tips

- A distributed array is created on the workers of the existing parallel pool. If no pool exists, `distributed` starts a new parallel pool unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `distributed` cannot start one, the result is the full array in the client workspace.

See Also

codistributed | datastore | gather | ones | parpool | spmd | tall | zeros

distributed.cell

Create distributed cell array

Syntax

```
D = distributed.cell(n)
D = distributed.cell(m, n, p, ...)
D = distributed.cell([m, n, p, ...])
```

Description

`D = distributed.cell(n)` creates an n-by-n distributed array of underlying class cell.

`D = distributed.cell(m, n, p, ...)` or `D = distributed.cell([m, n, p, ...])` create an m-by-n-by-p-by-... distributed array of underlying class cell.

Examples

Create a distributed 1000-by-1000 cell array:

```
D = distributed.cell(1000)
```

See Also

`cell` | `codistributed.cell`

Introduced in R2009b

distributed.spalloc

Allocate space for sparse distributed matrix

Syntax

```
SD = distributed.spalloc(M,N,nzmax)
```

Description

`SD = distributed.spalloc(M,N,nzmax)` creates an M-by-N all-zero sparse distributed matrix with room to hold `nzmax` nonzeros.

Examples

Allocate space for a 1000-by-1000 sparse distributed matrix with room for up to 2000 nonzero elements, then define several elements:

```
N = 1000;
SD = distributed.spalloc(N,N,2*N);
for ii=1:N-1
    SD(ii,ii:ii+1) = [ii ii];
end
```

See Also

`codistributed.spalloc` | `spalloc` | `sparse`

Introduced in R2009b

distributed.speye

Create distributed sparse identity matrix

Syntax

```
DS = distributed.speye(n)
DS = distributed.speye(m,n)
DS = distributed.speye([m,n])
```

Description

`DS = distributed.speye(n)` creates an n-by-n sparse distributed array of underlying class double.

`DS = distributed.speye(m,n)` or `DS = distributed.speye([m,n])` creates an m-by-n sparse distributed array of underlying class double.

Examples

Create a distributed 1000-by-1000 sparse identity matrix:

```
 N = 1000;
 DS = distributed.speye(N);
```

See Also
`codistributed.speye` | `eye` | `speye`

Introduced in R2009b

distributed.sprand

Create distributed sparse array of uniformly distributed pseudo-random values

Syntax

```
DS = distributed.sprand(m,n,density)
```

Description

`DS = distributed.sprand(m,n,density)` creates an m-by-n sparse distributed array with approximately `density*m*n` uniformly distributed nonzero double entries.

Examples

Create a 1000-by-1000 sparse distributed double array `DS` with approximately 1000 nonzeros.

```
DS = distributed.sprand(1000,1000,0.001);
```

Tips

When you use `sprand` on the workers in the parallel pool, or in an independent or communicating job, each worker sets its random generator seed to a value that depends only on the `labindex` or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

See Also
`codistributed.sprand` | `distributed.speye` | `distributed.sprandn` | `rand` | `randn` | `sparse` | `sprand`

Introduced in R2009b

distributed.sprandn

Create distributed sparse array of normally distributed pseudo-random values

Syntax

```
DS = distributed.sprandn(m,n,density)
```

Description

`DS = distributed.sprandn(m,n,density)` creates an m-by-n sparse distributed array with approximately `density*m*n` normally distributed nonzero double entries.

Examples

Create a 1000-by-1000 sparse distributed double array `DS` with approximately 1000 nonzeros.

```
DS = distributed.sprandn(1000,1000,0.001);
```

Tips

When you use `sprandn` on the workers in the parallel pool, or in an independent or communicating job, each worker sets its random generator seed to a value that depends only on the `labindex` or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

See Also
`codistributed.sprandn` | `distributed.speye` | `distributed.sprand` | `rand` | `randn` | `sparse` | `sprandn`

Introduced in R2009b

dload

Load distributed arrays and Composite objects from disk

Syntax

```
dload
dload filename
dload filename X
dload filename X Y Z ...
dload -scatter ...
[X,Y,Z,...] = dload('filename','X','Y','Z',...)
```

Description

`dload` without any arguments retrieves all variables from the binary file named `matlab.mat`. If `matlab.mat` is not available, the command generates an error.

`dload filename` retrieves all variables from a file given a full pathname or a relative partial pathname. If `filename` has no extension, `dload` looks for `filename.mat`. `dload` loads the contents of distributed arrays and Composite objects onto parallel pool workers, other data types are loaded directly into the workspace of the MATLAB client.

`dload filename X` loads only variable X from the file. `dload filename X Y Z ...` loads only the specified variables. `dload` does not support wildcards, nor the `-regexp` option. If any requested variable is not present in the file, a warning is issued.

`dload -scatter ...` distributes nondistributed data if possible. If the data cannot be distributed, a warning is issued.

`[X,Y,Z,...] = dload('filename','X','Y','Z',...)` returns the specified variables as separate output arguments (rather than a structure, which the `load` function returns). If any requested variable is not present in the file, an error occurs.

When loading distributed arrays, the data is distributed over the available parallel pool workers using the default distribution scheme. It is not necessary to have the same size pool open when loading as when saving using `dsave`.

When loading Composite objects, the data is sent to the available parallel pool workers. If the Composite is too large to fit on the current parallel pool, the data is not loaded. If the Composite is smaller than the current parallel pool, a warning is issued.

Examples

Load variables X, Y, and Z from the file `fname.mat`:

```
dload fname X Y Z
```

Use the function form of `dload` to load distributed arrays P and Q from file `fname.mat`:

```
[P,Q] = dload('fname.mat','P','Q');
```

See Also
Composite | distributed | dsave | load | parpool

Introduced in R2010a

dsave

Save workspace distributed arrays and Composite objects to disk

Syntax

```
dsave
dsave filename
dsave filename X
dsave filename X Y Z
```

Description

dsave without any arguments creates the binary file named `matlab.mat` and writes to the file all workspace variables, including distributed arrays and Composite objects. You can retrieve the variable data using `dload`.

`dsave filename` saves all workspace variables to the binary file named `filename.mat`. If you do not specify an extension for `filename`, it assumes the extension `.mat`.

`dsave filename X` saves only variable X to the file.

`dsave filename X Y Z` saves X, Y, and Z. `dsave` does not support wildcards, nor the `-regexp` option.

`dsave` does not support saving sparse distributed arrays.

Examples

With a parallel pool open, create and save several variables to `mydatafile.mat`:

```
D = rand(1000,'distributed'); % Distributed array
C = Composite();              %
C{1} = magic(20);             % Data on worker 1 only
X = rand(40);                 % Client workspace only
dsave mydatafile D C X        % Save all three variables
```

See Also
Composite | distributed | dload | parpool | save

Introduced in R2010a

exist

Check whether Composite is defined on workers

Syntax

```
h = exist(C,labidx)
h = exist(C)
```

Description

`h = exist(C,labidx)` returns `true` if the entry in Composite `C` has a defined value on the worker with labindex `labidx`, `false` otherwise. In the general case where `labidx` is an array, the output `h` is an array of the same size as `labidx`, and `h(i)` indicates whether the Composite entry `labidx(i)` has a defined value.

`h = exist(C)` is equivalent to `h = exist(C, 1:length(C))`.

If `exist(C,labidx)` returns `true`, `C(labidx)` does not throw an error, provided that the values of `C` on those workers are serializable. The function throws an error if any `labidx` is invalid.

Examples

Define a variable on a random number of workers. Check on which workers the Composite entries are defined, and get all those values:

```
spmd
  if rand() > 0.5
      c = labindex;
  end
end
ind = exist(c);
cvals = c(ind);
```

See Also
Composite

Introduced in R2008b

existsOnGPU

Determine if gpuArray or CUDAKernel is available on GPU

Syntax

```
TF = existsOnGPU(DATA)
```

Description

TF = existsOnGPU(DATA) returns a logical value indicating whether the gpuArray or CUDAKernel object represented by DATA is still present on the GPU and available from your MATLAB session. The result is false if DATA is no longer valid and cannot be used. Such arrays and kernels are invalidated when the GPU device has been reset with any of the following:

```
reset(dev)    % Where dev is the current gpuDevice
gpuDevice(ix) % Where ix is valid index of current or different device
gpuDevice([]) % With an empty argument (as opposed to no argument)
```

Examples

Query Existence of gpuArray

Create a gpuArray on the selected GPU device, then reset the device. Query array's existence and content before and after resetting.

```
g = gpuDevice(1);
M = gpuArray(magic(4));
M_exists = existsOnGPU(M)

    1

M  % Display gpuArray

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

reset(g);
M_exists = existsOnGPU(M)

    0

M  % Try to display gpuArray

Data no longer exists on the GPU.

clear M
```

See Also

gpuArray | gpuDevice | parallel.gpu.CUDAKernel | reset

Introduced in R2012a

eye

Identity matrix

Syntax

```
E = eye(sz,arraytype)
E = eye(sz,datatype,arraytype)

E = eye(sz,'like',P)
E = eye(sz,datatype,'like',P)

C = eye(sz,codist)
C = eye(sz,datatype,codist)
C = eye(sz, ___ ,codist,'noCommunication')
C = eye(sz, ___ ,codist,'like',P)
```

Description

`E = eye(sz,arraytype)` creates an *arraytype* identity matrix with underlying class of double, with ones on the main diagonal and zeros elsewhere.

`E = eye(sz,datatype,arraytype)` creates an *arraytype* identity matrix with underlying class of *datatype*, with ones on the main diagonal and zeros elsewhere.

The size and type of array are specified by the argument options according to the following table.

Argument	Values	Descriptions
sz	n	Specifies array size as an n-by-n matrix.
	m,n	Specifies array size as an m-by-n matrix.
	[m n]	
arraytype	'distributed'	Specifies distributed array.
	'codistributed'	Specifies codistributed array, using the default distribution scheme.
	'gpuArray'	Specifies gpuArray.
datatype	'double' (default), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'	Specifies underlying class of the array, i.e., the data type of its elements.

`E = eye(sz,'like',P)` creates an identity matrix of the same type and underlying class (data type) as array P.

`E = eye(sz,datatype,'like',P)` creates an identity matrix of the specified underlying class (*datatype*), and the same type as array P.

C = eye(sz,codist) or C = eye(sz,*datatype*,codist) creates a codistributed identity matrix of the specified size and underlying class (the default *datatype* is 'double'). The codistributor object, codist, specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = eye(8,codistributor1d());
end
```

C = eye(sz,___,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = eye(sz,___,codist,'like',P) creates a codistributed identity matrix with the specified size, underlying class (*datatype*), and distribution scheme. If either the *datatype* or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

Examples

Create Distributed Identity Matrix

Create a 1000-by-1000 distributed identity matrix of underlying class double:

```
D = eye(1000,'distributed');
```

Create Codistributed Identity Matrix

Create a 1000-by-1000 codistributed double identity matrix, distributed by its second dimension (columns).

```
spmd(4)
    C = eye(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed uint16 identity matrix , distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = eye(1000,1000,'uint16',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray Identity Matrix

Create a 1000-by-1000 gpuArray identity matrix of underlying class uint32:

```
G = eye(1000,'uint32','gpuArray');
```

See Also
Inf | NaN | codistributed.speye | distributed.speye | eye | false | ones | true | zeros

Introduced in R2006b

false

Array of logical 0 (false)

Syntax

```
F = false(sz,arraytype)

F = false(sz,'like',P)

C = false(sz,codist)
C = false(sz, ___ ,codist,'noCommunication')
C = false(sz, ___ ,codist,'like',P)
```

Description

`F = false(sz,arraytype)` creates a matrix with `false` values in all elements.

The size and type of array are specified by the argument options according to the following table.

Argument	Values	Descriptions
sz	n	Specifies size as an n-by-n matrix.
	`m,n` or `[m n]`	Specifies size as an m-by-n matrix.
	`m,n,...,k` or `[m n ... k]`	Specifies size as an m-by-n-by-...-by-k array.
arraytype	`'distributed'`	Specifies distributed array.
	`'codistributed'`	Specifies codistributed array, using the default distribution scheme.
	`'gpuArray'`	Specifies gpuArray.

`F = false(sz,'like',P)` creates an array of `false` values with the same type as array P.

`C = false(sz,codist)` creates a codistributed array of `false` values with the specified size. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = false(8,codistributor1d());
end
```

`C = false(sz, ___ ,codist,'noCommunication')` specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

`C = false(sz, ___ ,codist,'like',P)` creates a codistributed array of `false` values with the specified size and distribution scheme. If the codistributor argument is omitted, the distribution scheme is taken from the codistributed array P.

Examples

Create Distributed False Matrix

Create a 1000-by-1000 distributed array of `false`s with underlying class double:

```
D = false(1000,'distributed');
```

Create Codistributed False Matrix

Create a 1000-by-1000 codistributed matrix of `false`s, distributed by its second dimension (columns).

```
spmd(4)
    C = false(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of `C`.

Create a 1000-by-1000 codistributed matrix of `false`s, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = false(1000,1000,codist);
end
```

Each worker contains a 100-by-`labindex` local piece of `C`.

Create gpuArray False Matrix

Create a 1000-by-1000 gpuArray of `false`s:

```
G = false(1000,'gpuArray');
```

See Also
Inf | NaN | eye | false | ones | true | zeros

Introduced in R2006b

fetchNext

Package: `parallel`

Retrieve next available unread FevalFuture outputs

Syntax

```
[idx,B1,B2,...,Bn] = fetchNext(F)
[idx,B1,B2,...,Bn] = fetchNext(F,TIMEOUT)
```

Description

`[idx,B1,B2,...,Bn] = fetchNext(F)` waits for an unread FevalFuture in the array of futures F to finish, and then returns the linear index of that future in array F as `idx`, along with the future's results in `B1,B2,...,Bn`. Before this call, the `'Read'` property of the particular future is `false`; afterward it is `true`.

`[idx,B1,B2,...,Bn] = fetchNext(F,TIMEOUT)` waits no longer than `TIMEOUT` seconds for a result to become available. If the timeout expires before any result becomes available, all output arguments are empty.

If there are no futures in F whose `'Read'` property is `false`, then an error is reported. You can check whether there are any unread futures using `anyUnread = ~all([F.Read])`.

If the element of F which has become finished encountered an error during execution, that error will be thrown by `fetchNext`. However, that future's `'Read'` property is set `true`, so that any subsequent calls to `fetchNext` can proceed.

Examples

Request several function evaluations, and update a progress bar while waiting for completion.

```
N = 100;
for idx = N:-1:1
    % Compute the rank of N magic squares
    F(idx) = parfeval(@rank,1,magic(idx));
end
% Build a waitbar to track progress
h = waitbar(0,'Waiting for FevalFutures to complete...');
results = zeros(1,N);
for idx = 1:N
    [completedIdx,thisResult] = fetchNext(F);
    % store the result
    results(completedIdx) = thisResult;
    % update waitbar
    waitbar(idx/N,h,sprintf('Latest result: %d',thisResult));
end
delete(h)
```

Tips

The `fetchNext` function returns the linear index of the future from its array. If instead, you need the subscript values of a multidimensional array, you can use the `ind2sub` function to convert the values.

See Also

`fetchOutputs` | `isequal` | `parfeval` | `parfevalOnAll` | `parpool`

Introduced in R2013b

fetchOutputs

Package: parallel

Retrieve output arguments from all tasks in job

Syntax

```
data = fetchOutputs(job)
```

Description

`data = fetchOutputs(job)` retrieves the output arguments contained in the tasks of a finished job. If the job has M tasks, each row of the M-by-N cell array `data` contains the output arguments for the corresponding task in the job. Each row has N elements, where N is the greatest number of output arguments from any one task in the job. The N elements of a row are arrays containing the output arguments from that task. If a task has less than N output arguments, the excess arrays in the row for that task are empty. The order of the rows in `data` is the same as the order of the tasks contained in the job's `Tasks` property.

Calling `fetchOutputs` does not remove the output data from the location where it is stored. To remove the output data, use the `delete` function to remove individual tasks or entire jobs.

`fetchOutputs` reports an error if the job is not in the `'finished'` state, or if one of its tasks encountered an error during execution. If some tasks completed successfully, you can access their output arguments directly from the `OutputArguments` property of the tasks.

Examples

Create a job to generate a random matrix:

```
myCluster = parcluster; % Use default profile
j = createJob(myCluster,'Name','myjob');
t = createTask(j,@rand,1,{10});
submit(j);
```

Wait for the job to finish and retrieve the random matrix:

```
wait(j)
data = fetchOutputs(j);
data{1}
```

Introduced in R2012a

fetchOutputs

Package: `parallel`

Retrieve all output arguments from Future

Syntax

```
[B1,B2,...,Bn] = fetchOutputs(F)
[B1,B2,...,Bn] = fetchOutputs(F,'UniformOutput',false)
```

Description

`[B1,B2,...,Bn] = fetchOutputs(F)` fetches all outputs of the `Future` object `F` after first waiting for each element of `F` to reach the state `'finished'`. An error results if any element of `F` has `NumOutputArguments` less than the requested number of outputs.

When `F` is a vector of FevalFutures, each output argument is formed by concatenating the corresponding output arguments from each future in `F`. An error results if these outputs cannot be concatenated. To avoid this error, set the `'UniformOutput'` option to `false`.

`[B1,B2,...,Bn] = fetchOutputs(F,'UniformOutput',false)` requests that `fetchOutputs` combine the future outputs into cell arrays `B1,B2,...,Bn`. The outputs of `F` can be of any size or type.

After the call to `fetchOutputs`, all futures in `F` have their `'Read'` property set to `true`. `fetchOutputs` returns outputs for all futures in `F` regardless of the value of each future's `'Read'` property.

Examples

Create an FevalFuture, and fetch its outputs.

```
f = parfeval(@rand,1,3);
R = fetchOutputs(f)
```

```
0.5562    0.6218    0.3897
0.0084    0.4399    0.2700
0.0048    0.9658    0.8488
```

Create an FevalFuture vector, and fetch all its outputs. For efficiency, preallocate an array of future objects before.

```
F(1:10) = parallel.FevalFuture;
for idx = 1:10
    F(idx) = parfeval(@rand,1,1,10); % One row each future
end
R = fetchOutputs(F); % 10-by-10 concatenated output
```

See Also
`fetchNext` | `parfeval` | `parpool`

Introduced in R2013b

feval

Evaluate kernel on GPU

Syntax

```
feval(KERN, x1, ..., xn)
[y1, ..., ym] = feval(KERN, x1, ..., xn)
```

Description

`feval(KERN, x1, ..., xn)` evaluates the CUDA kernel `KERN` with the given arguments `x1, ..., xn`. The number of input arguments, n, must equal the value of the `NumRHSArguments` property of `KERN`, and their types must match the description in the `ArgumentTypes` property of `KERN`. The input data can be regular MATLAB data, GPU arrays, or a mixture of the two.

`[y1, ..., ym] = feval(KERN, x1, ..., xn)` returns multiple output arguments from the evaluation of the kernel. Each output argument corresponds to the value of the non-const pointer inputs to the CUDA kernel after it has executed. The output from `feval` running a kernel on the GPU is always `gpuArray` type, even if all the inputs are data from the MATLAB workspace. The number of output arguments, m, must not exceed the value of the `MaxNumLHSArguments` property of `KERN`.

Examples

If the CUDA kernel within a CU file has the following signature:

```
void myKernel(const float * pIn, float * pInOut1, float * pInOut2)
```

The corresponding kernel object in MATLAB then has the properties:

```
MaxNumLHSArguments: 2
   NumRHSArguments: 3
     ArgumentTypes: {'in single vector'  ...
                     'inout single vector' 'inout single vector'}
```

You can use `feval` on this code's kernel (`KERN`) with the syntax:

```
[y1, y2] = feval(KERN, x1, x2, x3)
```

The three input arguments, `x1`, `x2`, and `x3`, correspond to the three arguments that are passed into the CUDA function. The output arguments, `y1` and `y2`, are gpuArray types, and correspond to the values of `pInOut1` and `pInOut2` after the CUDA kernel has executed.

See Also
`arrayfun` | `gather` | `gpuArray` | `parallel.gpu.CUDAKernel`

Introduced in R2010b

findJob

Find job objects stored in cluster

Syntax

```
out = findJob(c)
[pending queued running completed] = findJob(c)
out = findJob(c,'p1',v1,'p2',v2,...)
```

Arguments

c	Cluster object in which to find the job.
pending	Array of jobs whose `State` is `pending` in cluster `c`.
queued	Array of jobs whose `State` is `queued` in cluster `c`.
running	Array of jobs whose `State` is `running` in cluster `c`.
completed	Array of jobs that have completed running, i.e., whose `State` is `finished` or `failed` in cluster `c`.
out	Array of jobs found in cluster `c`.
p1, *p2*	Job object properties to match.
v1, v2	Values for corresponding object properties.

Description

`out = findJob(c)` returns an array, `out`, of all job objects stored in the cluster `c`. Jobs in the array are ordered by the `ID` property of the jobs, indicating the sequence in which they were created.

`[pending queued running completed] = findJob(c)` returns arrays of all job objects stored in the cluster `c`, by state. Within `pending`, `running`, and `completed`, the jobs are returned in sequence of creation. Jobs in the array `queued` are in the order in which they are queued, with the job at `queued(1)` being the next to execute. The `completed` jobs include those that failed. Jobs that are deleted or whose status is unavailable are not returned by this function.

`out = findJob(c,'p1',v1,'p2',v2,...)` returns an array, `out`, of job objects whose property values match those passed as property-value pairs, *p1*, v1, *p2*, v2, etc. The property name must be a character vector, with the value being the appropriate type for that property. For a match, the object property value must be exactly the same as specified, including letter case. For example, if a job's `Name` property value is `MyJob`, then `findJob` will not find that object while searching for a `Name` property value of `myjob`.

See Also
createJob | findTask | parcluster | recreate | submit

Introduced before R2006a

findTask

Task objects belonging to job object

Syntax

```
tasks = findTask(j)
tasks = findTask(j, taskFcn)
[pending running completed] = findTask(j)
tasks = findTask(j,'p1',v1,'p2',v2,...)
```

Arguments

j	Job object.
tasks	Returned task objects.
pending	Array of tasks in job `obj` whose `State` is `pending`.
running	Array of tasks in job `obj` whose `State` is `running`.
completed	Array of completed tasks in job `obj`, i.e., those whose `State` is `finished` or `failed`.
p1, *p2*	Task object properties to match.
v1, v2	Values for corresponding object properties.

Description

`tasks = findTask(j)` gets a 1-by-N array of task objects belonging to a job object j. Tasks in the array are ordered by the `ID` property of the tasks, indicating the sequence in which they were created.

`tasks = findTask(j, taskFcn)` returns an array of task objects that belong to the job j, using `taskFcn` to select them. `taskFcn` is a function handle that accepts `j.Tasks` as an input argument, and returns a logical array indicating the tasks to return.

`[pending running completed] = findTask(j)` returns arrays of all task objects stored in the job object j, sorted by `state`. Within each array (`pending`, `running`, and `completed`), the tasks are returned in sequence of creation.

`tasks = findTask(j,'p1',v1,'p2',v2,...)` returns an array of task objects belonging to a job object j. The returned task objects will be only those matching the specified property-value pairs, *p1*, v1, *p2*, v2, etc. The property name must be a character vector, with the value being the appropriate type for that property. For a match, the object property value must be exactly the same as specified, including letter case. For example, if a task's `Name` property value is `MyTask`, then `findTask` will not find that object while searching for a `Name` property value of `mytask`.

Examples

Create a job object.

```
c = parcluster();
j = createJob(c);
```

Add a task to the job object.

```
createTask(j,@rand,1,{10})
```

Find all task objects now part of job j.

```
t = findTask(j)
```

Tips

If job j is contained in a remote service, `findTask` will result in a call to the remote service. This could result in `findTask` taking a long time to complete, depending on the number of tasks retrieved and the network speed. Also, if the remote service is no longer available, an error will be thrown.

See Also
createJob | createTask | findJob

Introduced before R2006a

for (drange)

`for`-loop over distributed range

Syntax

```
for variable = drange(colonop)
    statement
    ...
    statement
end
```

Description

The general format is

```
for variable = drange(colonop)
    statement
    ...
    statement
end
```

The `colonop` is an expression of the form `start:increment:finish` or `start:finish`. The default value of increment is 1. The `colonop` is partitioned by `codistributed.colon` into `numlabs` contiguous segments of nearly equal length. Each segment becomes the iterator for a conventional for-loop on an individual worker.

The most important property of the loop body is that each iteration must be independent of the other iterations. Logically, the iterations can be done in any order. No communication with other workers is allowed within the loop body. The functions that perform communication are `gop`, `gcat`, `gplus`, `codistributor`, `codistributed`, `gather`, and `redistribute`.

It is possible to access portions of codistributed arrays that are local to each worker, but it is not possible to access other portions of codistributed arrays.

The `break` statement can be used to terminate the loop prematurely.

Examples

Find the rank of magic squares. Access only the local portion of a codistributed array.

```
r = zeros(1, 40, codistributor());
for n = drange(1:40)
   r(n) = rank(magic(n));
end
r = gather(r);
```

Perform Monte Carlo approximation of pi. Each worker is initialized to a different random number state.

```
m = 10000;
for p = drange(1:numlabs)
   z = rand(m,1) + i*rand(m,1);
```

```
    c = sum(abs(z) < 1)
end
k = gplus(c)
p = 4*k/(m*numlabs);
```

Attempt to compute Fibonacci numbers. This will *not* work, because the loop bodies are dependent.

```
f = zeros(1, 50, codistributor());
f(1) = 1;
f(2) = 2;
for n = drange(3:50)
    f(n) = f(n-1) + f(n-2)
end
```

See Also
for | numlabs | parfor

Introduced in R2007b

gather

Transfer distributed array or gpuArray to local workspace

Syntax

```
X = gather(A)
[X1,X2,...,Xn] = gather(A1,A2,...,,Xn)
X = gather(C,lab)
[X1,X2,...,Xn] = gather(C1,C2,...,Cn,lab)
```

Description

`X = gather(A)` can operate on the following array data:

- On a gpuArray: Transfers the elements of `A` from the GPU to the local workspace and assigns them to `X`.
- On a distributed array, outside an `spmd` statement: Gathers together the elements of `A` from the multiple workers to the local workspace and assigns them to `X`.
- On a codistributed array, inside an `spmd` statement or communicating job: Gathers together the elements of `A` and replicates them into `X` on every worker.

You can call `gather` on other data types, such as tall arrays (See `gather (tall)`). If the data type does not support gathering, then `gather` has no effect.

Gathering GPU arrays or distributed arrays can be costly and is generally not necessary unless you need to use your result with functions that do not support these types of arrays. For more information on function support, see "Run MATLAB Functions on a GPU" on page 8-9 or "Run MATLAB Functions with Distributed Arrays" on page 4-18.

`X = gather(gpuArray(X))`, `X = gather(distributed(X))`, or `X = gather(codistributed(X))` return the original array `X`.

`[X1,X2,...,Xn] = gather(A1,A2,...,,Xn)` gathers multiple arrays `A1,A2,...,An` into the corresponding outputs `X1,X2,...,Xn`. The number of input arguments and output arguments must match.

`X = gather(C,lab)` converts a codistributed array `C` to a variant array `X`, such that all of the elements are contained on worker `lab`, and `X` is a 0-by-0 empty double on all other workers.

`[X1,X2,...,Xn] = gather(C1,C2,...,Cn,lab)` gathers codistributed arrays `C1,C2,...,Cn` into corresponding outputs `X1,X2,...,Xn`, with all elements on worker `lab`. The number of input arguments and output arguments must match.

Examples

Gather gpuArrays

Gather the results of a GPU operation to the MATLAB workspace.

```
G = gpuArray(rand(1024,1));
F = sqrt(G);    % Input and output are both gpuArray
W = gather(G); % Return array to workspace
whos
```

```
Name          Size         Bytes  Class

F           1024x1           108  gpuArray
G           1024x1           108  gpuArray
W           1024x1          8192  double
```

Gather Distributed Arrays

Gather all of the elements from a distributed array D onto the client.

```
n = 10;
D = distributed(magic(n)); % Distribute array to workers
M = gather(D)              % Return array to client
```

Gather Codistributed Arrays

Distribute a magic square across your workers, then gather the whole matrix onto every worker and then onto the client. This code results in the equivalent of M = magic(n) on all workers and the client.

```
n = 10;
spmd
  C = codistributed(magic(n));
  M = gather(C) % Gather all elements to all workers
end
S = gather(C)    % Gather elements to client
```

Gather all of the elements of C onto worker 1, for operations that cannot be performed across distributed arrays.

```
n = 10;
spmd
  C = codistributed(magic(n));
  out = gather(C,1);
  if labindex == 1
    % Characteristic sum for this magic square:
    characteristicSum = sum(1:n^2)/n;
    % Ensure that the diagonal sums are equal to the
    % characteristic sum:
    areDiagonalsEqual = isequal ...
      (trace(out),trace(flipud(out)),characteristicSum)
  end
end
```

```
Lab 1:
  areDiagonalsEqual =
     1
```

Input Arguments

A — Array to gather
gpuArray | distributed array | codistributed array

Array to gather, specified as a gpuArray, distributed array, or codistributed array.

Tips

Note that `gather` assembles the codistributed or distributed array in the workspaces of all the workers on which it executes, or on the MATLAB client, respectively, but not both. If you are using `gather` within an `spmd` statement, the gathered array is accessible on the client via its corresponding `Composite` object; see "Access Worker Variables with Composites" on page 3-7. If you are running `gather` in a communicating job, you can return the gathered array to the client as an output argument from the task.

As the `gather` function requires communication between all the workers, you cannot gather data from all the workers onto a single worker by placing the function inside a conditional statement such as `if labindex == 1`.

See Also
arrayfun | bsxfun | codistributed | distributed | gpuArray | pagefun

Introduced in R2006b

gcat

Global concatenation

Syntax

```
Xs = gcat(X)
Xs = gcat(X,dim)
Xs = gcat(X,dim,targetlab)
```

Description

`Xs = gcat(X)` concatenates the variant array X from each worker in the second dimension. The result is replicated on all workers.

`Xs = gcat(X,dim)` concatenates the variant array X from each worker in the dimension indicated by `dim`.

`Xs = gcat(X,dim,targetlab)` performs the reduction, and places the result into `res` only on the worker indicated by `targetlab`. `res` is set to `[]` on all other workers.

Examples

With four workers,

```
Xs = gcat(labindex)
```

returns `Xs = [1 2 3 4]` on all four workers.

See Also

cat | gop | labindex | numlabs

Introduced in R2006b

gcp

Get current parallel pool

Syntax

```
p = gcp
p = gcp('nocreate')
```

Description

`p = gcp` returns a parallel.Pool object representing the current parallel pool. The current pool is where parallel language features execute, such as `parfor`, `spmd`, `distributed`, `Composite`, `parfeval` and `parfevalOnAll`.

If no parallel pool exists, `gcp` starts a new parallel pool and returns a pool object for that, unless automatic pool starts are disabled in your parallel preferences. If no parallel pool exists and automatic pool starts are disabled, `gcp` returns an empty pool object.

`p = gcp('nocreate')` returns the current pool if one exists. If no pool exists, the `'nocreate'` option causes `gcp` not to create a pool, regardless of your parallel preferences settings.

Examples

Find Size of Current Pool

Find the number of workers in the current parallel pool.

```
p = gcp('nocreate'); % If no pool, do not create new one.
if isempty(p)
    poolsize = 0;
else
    poolsize = p.NumWorkers
end
```

Delete Current Pool

Use the parallel pool object to delete the current pool.

```
delete(gcp('nocreate'))
```

See Also
Composite | delete | distributed | parfeval | parfevalOnAll | parfor | parpool | spmd

Introduced in R2013b

getAttachedFilesFolder

Folder into which AttachedFiles are written

Syntax

```
folder = getAttachedFilesFolder
folder = getAttachedFilesFolder(FileName)
```

Arguments

folder	Character vector indicating location where files from job's AttachedFiles property are placed
FileName	Character vector specifying all or part of the attached file or folder name

Description

`folder = getAttachedFilesFolder` returns the path to the local folder into which AttachedFiles are written on the worker. This function returns an empty array if it is not called on a MATLAB worker.

`folder = getAttachedFilesFolder(FileName)` returns the path name to the specified attached folder on the worker, or the folder containing the specified attached file. FileName can match either the full name of the attached file or folder, or on the ending part of the name. Multiple match results return a cell array.

If you have attached a folder, this does not match on file names within that folder.

Suppose you attach the folder `'C:\monday\tuesday\wednesday\thursday'`, which on the workers is stored in `/tmp/MJS/tp12345`. The following table displays the results of various match attempts.

Specified Matching Character Vector Argument	Result
getAttachedFilesFolder('C:\monday')	Empty result, because `'C:\monday'` is only the start of the path, and does not include `'thursday'`
getAttachedFilesFolder('wednesday')	Empty result, because `'wednesday'` is in the middle of the path and does not include `'thursday'`
getAttachedFilesFolder('thurs')	Empty result, because `'thurs'` is not the ending of the folder name.
getAttachedFilesFolder('thursday')	`'/tmp/MJS/tp12345'`
getAttachedFilesFolder('wednesday\thursday')	`'/tmp/MJS/tp12345'`

Examples

Attach a folder to a parallel pool, then find its location on the worker to execute one of its files.

```matlab
myPool = parpool;
addAttachedFiles(myPool,'mydir');
spmd
    folder = getAttachedFilesFolder('mydir');
    oldFolder = cd(folder);  % Change to that folder
    [OK,output] = system('myExecutable');
    cd(oldFolder);            % Change to original folder
 end
```

Attach an executable file to a parallel pool, then change to its folder for accessing and processing some data.

```matlab
myPool = parpool;
addAttachedFiles(myPool,'myExecutable');
spmd
    system('myExecutable');   % Now on MATLAB path
    folder = getAttachedFilesFolder('myExecutable');
    oldFolder = cd(folder);
    fid = open('myData.txt'); % Access data file
        % Process fid
    close(fid)
    cd(oldFolder);            % Change back to the original folder
end
```

See Also

Functions
addAttachedFiles | getCurrentCluster | getCurrentJob | getCurrentTask | getCurrentWorker

Introduced in R2012a

getCodistributor

Codistributor object for existing codistributed array

Syntax

```
codist = getCodistributor(D)
```

Description

`codist = getCodistributor(D)` returns the codistributor object of codistributed array D. Properties of the object are `Dimension` and `Partition` for 1-D distribution; and `BlockSize`, `LabGrid`, and `Orientation` for 2-D block cyclic distribution. For any one codistributed array, `getCodistributor` returns the same values on all workers. The returned codistributor object is complete, and therefore suitable as an input argument for `codistributed.build`.

Examples

Get the codistributor object for a 1-D codistributed array that uses default distribution on 4 workers:

```
spmd (4)
    I1 = eye(64,codistributor1d());
    codist1 = getCodistributor(I1)
    dim     = codist1.Dimension
    partn   = codist1.Partition
end
```

Get the codistributor object for a 2-D block cyclic codistributed array that uses default distribution on 4 workers:

```
spmd (4)
    I2 = eye(128,codistributor2dbc());
    codist2 = getCodistributor(I2)
    blocksz = codist2.BlockSize
    partn   = codist2.LabGrid
    ornt    = codist2.Orientation
end
```

Demonstrate that these codistributor objects are complete:

```
spmd (4)
    isComplete(codist1)
    isComplete(codist2)
end
```

See Also
`codistributed` | `codistributed.build` | `getLocalPart` | `redistribute`

Introduced in R2009b

getCurrentCluster

Cluster object that submitted current task

Syntax

```
c = getCurrentCluster
```

Arguments

c	The cluster object that scheduled the task currently being evaluated by the worker session.

Description

`c = getCurrentCluster` returns the `parallel.Cluster` object that has sent the task currently being evaluated by the worker session. Cluster object `c` is the `Parent` of the task's parent job.

Examples

Find the current cluster.

```
myCluster = getCurrentCluster;
```

Get the host on which the cluster is running.

```
host = myCluster.Host;
```

Tips

If this function is executed in a MATLAB session that is not a worker, you get an empty result.

See Also
getAttachedFilesFolder | getCurrentJob | getCurrentTask | getCurrentWorker

Introduced in R2012a

getCurrentJob

Job object whose task is currently being evaluated

Syntax

```
job = getCurrentJob
```

Arguments

job	The job object that contains the task currently being evaluated by the worker session.

Description

`job = getCurrentJob` returns the `parallel.Job` object that is the `Parent` of the task currently being evaluated by the worker session.

Tips

If the function is executed in a MATLAB session that is not a worker, you get an empty result.

See Also

getAttachedFilesFolder | getCurrentCluster | getCurrentTask | getCurrentWorker

Introduced before R2006a

getCurrentTask

Task object currently being evaluated in this worker session

Syntax

```
task = getCurrentTask
```

Arguments

task	The task object that the worker session is currently evaluating.

Description

`task = getCurrentTask` returns the `parallel.Task` object whose function is currently being evaluated by the MATLAB worker session on the cluster.

Tips

If the function is executed in a MATLAB session that is not a worker, you get an empty result.

See Also

getAttachedFilesFolder | getCurrentCluster | getCurrentJob | getCurrentWorker

Introduced before R2006a

getCurrentWorker

Worker object currently running this session

Syntax

```
worker = getCurrentWorker
```

Arguments

worker	The worker object that is currently evaluating the task that contains this function.

Description

`worker = getCurrentWorker` returns the `parallel.Worker` object representing the MATLAB worker session that is currently evaluating the task function that contains this call.

If the function runs in a MATLAB session that is not a worker, it returns an empty result.

Examples

Find the `Host` property of a worker that runs a task. The file `identifyWorkerHost.m` contains the following function code.

```
function localHost = identifyWorkerHost()
  thisworker = getCurrentWorker; % Worker object
  localHost = thisworker.Host;   % Host property
end
```

Create a job with a task to execute this function on a worker and return the worker's host name. This example manually attaches the necessary code file.

```
c = parcluster();
j = createJob(c);
j.AttachedFiles = {'identifyWorkerHost.m'};
t = createTask(j,@identifyWorkerHost,1,{});
submit(j)
wait(j)
workerhost = fetchOutputs(j)
```

See Also
getAttachedFilesFolder | getCurrentCluster | getCurrentJob | getCurrentTask

Introduced before R2006a

getDebugLog

Read output messages from job run in CJS cluster

Syntax

```
str = getDebugLog(cluster,job_or_task)
```

Arguments

`str`	Variable to which messages are returned as a character vector expression.
`cluster`	Cluster object referring to Microsoft Windows HPC Server (or CCS), Platform LSF, PBS Pro, or TORQUE cluster, created by `parcluster`.
`job_or_task`	Object identifying job or task whose messages you want.

Description

`str = getDebugLog(cluster,job_or_task)` returns any output written to the standard output or standard error stream by the job or task identified by `job_or_task`, being run in the cluster identified by `cluster`.

Examples

This example shows how to create and submit a communicating job, and how to retrieve the job's debug log. Assume that you already have a cluster profile called `My3pCluster` that defines the properties of the cluster.

```
c = parcluster('My3pCluster');

j = createCommunicatingJob(c);
createTask(j,@labindex,1,{});
submit(j);

getDebugLog(c,j);
```

See Also
createCommunicatingJob | createJob | createTask | parcluster

Introduced before R2006a

getJobClusterData

Get specific user data for job on generic cluster

Syntax

```
userdata = getJobClusterData(cluster,job)
```

Arguments

userdata	Information that was previously stored for this job
cluster	Cluster object identifying the generic third-party cluster running the job
job	Job object identifying the job for which to retrieve data

Description

`userdata = getJobClusterData(cluster,job)` returns data stored for the job `job` that was derived from the generic cluster `cluster`. The information was originally stored with the function `setJobClusterData`. For example, it might be useful to store the third-party scheduler's external ID for this job, so that the function specified in `GetJobStateFcn` can later query the scheduler about the state of the job.

For more information and examples on using these functions and properties, see "Plugin Scripts for Generic Schedulers" on page 6-17.

See Also
setJobClusterData

Introduced in R2012a

getJobFolder

Folder on client where jobs are stored

Syntax

```
joblocation = getJobFolder(cluster,job)
```

Description

`joblocation = getJobFolder(cluster,job)` returns the path to the folder on disk where files are stored for the specified job and cluster. This folder is valid only the client MATLAB session, not necessarily the workers. This method exists only on clusters using the generic interface.

See Also
getJobFolderOnCluster | parcluster

Introduced in R2012a

getJobFolderOnCluster

Folder on cluster where jobs are stored

Syntax

```
joblocation = getJobFolderOnCluster(cluster,job)
```

Description

`joblocation = getJobFolderOnCluster(cluster,job)` returns the path to the folder on disk where files are stored for the specified job and cluster. This folder is valid only in worker MATLAB sessions. An error results if the `HasSharedFilesystem` property of the cluster is `false`. This method exists only on clusters using the generic interface.

See Also
getJobFolder | parcluster

Introduced in R2012a

getLocalPart

Local portion of codistributed array

Syntax

```
L = getLocalPart(A)
```

Description

`L = getLocalPart(A)` returns the local portion of a codistributed array.

Examples

With four workers,

```
A = magic(4);    %replicated on all workers
D = codistributed(A, codistributor1d(1));
L = getLocalPart(D)
```

returns

```
Lab 1: L = [16  2  3 13]
Lab 2: L = [ 5 11 10  8]
Lab 3: L = [ 9  7  6 12]
Lab 4: L = [ 4 14 15  1]
```

See Also
`codistributed` | `codistributor`

Introduced in R2009b

getLogLocation

Log location for job or task

Syntax

```
logfile = getLogLocation(cluster,cj)
logfile = getLogLocation(cluster,it)
```

Description

`logfile = getLogLocation(cluster,cj)` for a generic cluster `cluster` and communicating job `cj`, returns the location where the log data should be stored for the whole job `cj`.

`logfile = getLogLocation(cluster,it)` for a generic cluster `cluster` and task `it` of an independent job returns the location where the log data should be stored for the task `it`.

This function can be useful during submission, to instruct the third-party cluster to put worker output logs in the correct location.

See Also
`parcluster`

Introduced in R2012a

getTaskSchedulerIDs

Package: `parallel.job`

Scheduler IDs of tasks in job

Syntax

```
schedulerIDs = getTaskSchedulerIDs(job)
```

Description

`schedulerIDs = getTaskSchedulerIDs(job)` returns the `SchedulerID` of each task on the job `job`. Note that `SchedulerID` applies only to third-party schedulers.

Examples

Get Scheduler IDs of Tasks

Create a cluster object by using `parcluster`. In the code below, change `MyThirdPartyScheduler` to the name of the profile of your third-party scheduler.

```
c = parcluster('MyThirdPartyScheduler');
```

Create a job and create some tasks for it. Then, submit the job.

```
job = createJob(c);
for idx = 1:2
    createTask(job,@ode45,2,{@vdp1,[0,10],[idx,0]});
end
submit(job)
```

To get the scheduler IDs of the tasks on the job, use `getTaskSchedulerIDs`. You can use these IDs to refer to the corresponding jobs on the third-party scheduler.

```
getTaskSchedulerIDs(job)
```

```
ans = 1×1 cell array
    {'4933'}
```

In this case, the scheduler has assigned the ID 4933 to this job.

Wait for the job to finish and fetch its outputs.

```
wait(job);
out = fetchOutputs(job)
```

```
out = 2×2 cell array
    {121×1 double}    {121×2 double}
    {129×1 double}    {129×2 double}
```

When you are done retrieving information from the job, delete it to clean up its data.

```
delete(job);
clear job
```

Input Arguments

job — Job
parallel.Job object

Job object that represents the job on the third-party scheduler, specified as a `parallel.Job` object.

Example: `job = createJob(parcluster);`

Data Types: `parallel.Job`

Output Arguments

schedulerIDs — Scheduler IDs
cell array of character vectors

`SchedulerID` of each task on `job`, returned as a cell array of character vectors.

See Also
parallel.Job | parcluster

Topics
"Install and Configure MATLAB Parallel Server for Third-Party Schedulers" (MATLAB Parallel Server)

Introduced in R2019b

globalIndices

Global indices for local part of codistributed array

Syntax

```
K = globalIndices(C,dim)
K = globalIndices(C,dim,lab)
[E,F] = globalIndices(C,dim)
[E,F] = globalIndices(C,dim,lab)
K = globalIndices(codist,dim,lab)
[E,F] = globalIndices(codist,dim,lab)
```

Description

globalIndices tells you the relationship between indices on a local part and the corresponding index range in a given dimension on the codistributed array. The globalIndices method on a codistributor object allows you to get this relationship without actually creating the array.

K = globalIndices(C,dim) or K = globalIndices(C,dim,lab) returns a vector K so that getLocalPart(C) = C(...,K,...) in the specified dimension dim of codistributed array C on the specified worker. If the lab argument is omitted, the default is labindex.

[E,F] = globalIndices(C,dim) or [E,F] = globalIndices(C,dim,lab) returns two integers E and F so that getLocalPart(C) = C(...,E:F,...) of codistributed array C in the specified dimension dim on the specified worker. If the lab argument is omitted, the default is labindex.

K = globalIndices(codist,dim,lab) is the same as K = globalIndices(C,dim,lab), where codist is the codistributor to be used for C, or codist = getCodistributor(C). This allows you to get the global indices for a codistributed array without having to create the array itself.

[E,F] = globalIndices(codist,dim,lab) is the same as [E,F] = globalIndices(C,dim,lab), where codist is the codistributor to be used for C, or codist = getCodistributor(C). This allows you to get the global indices for a codistributed array without having to create the array itself.

Examples

Create a 2-by-22 codistributed array among four workers, and view the global indices on each lab:

```
spmd
    C = zeros(2,22,codistributor1d(2,[6 6 5 5]));
    if labindex == 1
      K = globalIndices(C,2)      % returns K = 1:6.
    elseif labindex == 2
      [E,F] = globalIndices(C,2) % returns E = 7, F = 12.
    end
    K = globalIndices(C,2,3)      % returns K = 13:17.
    [E,F] = globalIndices(C,2,4)  % returns E = 18, F = 22.
 end
```

Use `globalIndices` to load data from a file and construct a codistributed array distributed along its columns, i.e., dimension 2. Notice how `globalIndices` makes the code not specific to the number of workers and alleviates you from calculating offsets or partitions.

```
spmd
    siz = [1000,1000];
    codistr = codistributor1d(2,[],siz);

    % Use globalIndices to figure out which columns
    % each worker should load.
    [firstCol,lastCol] = globalIndices(codistr,2);

    % Call user-defined function readRectangleFromFile to
    % load all the values that should go into
    % the local part for this worker.
    labLocalPart = readRectangleFromFile(fileName, ...
                            1,siz(1),firstCol,lastCol);

    % With the local part and codistributor,
    % construct the corresponding codistributed array.
    C = codistributed.build(labLocalPart,codistr);
end
```

See Also
getLocalPart | labindex

Introduced in R2008a

gop

Global operation across all workers

Syntax

```
res = gop(FUN,x)
res = gop(FUN,x,targetlab)
```

Arguments

FUN	Function to operate across workers.
x	Argument to function F, should be the same variable on all workers, but can have different values.
res	Variable to hold reduction result.
targetlab	Lab to which reduction results are returned. This value is returned by that worker's labindex.

Description

`res = gop(FUN,x)` is the reduction via the function `FUN` of the quantities `x` from each worker. The result is duplicated on all workers.

`FUN` can be a handle to any function, including user-written functions and user-defined anonymous functions. It should accept two arguments of the same type, and return one result of that same type, so it can be used iteratively in the form:

```
FUN(FUN(x1,x2),FUN(x3,x4))
```

The function `FUN` should be associative, that is,

```
FUN(FUN(x1,x2),x3) = FUN(x1,FUN(x2,x3))
```

`res = gop(FUN,x,targetlab)` performs the reduction, and places the result into `res` only on the worker indicated by `targetlab`. `res` is set to `[ ]` on all other workers.

Examples

This example shows how to calculate the sum and maximum values for `x` among all workers.

```
p = parpool('local',4);
x = Composite();
x{1} = 3;
x{2} = 1;
x{3} = 4;
x{4} = 2;
spmd
    xsum = gop(@plus,x);
    xmax = gop(@max,x);
end
xsum{1}
```

```
10
```

```
xmax{1}
```

```
4
```

This example shows how to horizontally concatenate the column vectors of x from all workers into a matrix. It uses the same 4-worker parallel pool opened by the previous example.

```
x{1} = [3;30];
x{2} = [1;10];
x{3} = [4;40];
x{4} = [2;20];
spmd
    res = gop(@horzcat,x);
end
res{1}

     3     1     4     2
    30    10    40    20
```

This example shows how to use an anonymous function with `gop` to join character vectors with spaces between them. In this case, the character vectors are created from each worker's `labindex` value.

```
afun = @(a,b)[a,' ',b]
spmd
    res = gop(afun,num2str(labindex));
end
res{1}

1 2 3 4
```

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

If x is a `gpuArray`, data transfer between multiple GPUs in a parallel pool uses fast peer-to-peer communication, including NVLink, if available.

See Also
`labBarrier` | `labindex` | `numlabs`

Topics
"Using GOP to Achieve MPI_Allreduce Functionality" on page 9-119

Introduced before R2006a

gplus

Global addition

Syntax

```
S = gplus(X)
S = gplus(X,targetlab)
```

Description

`S = gplus(X)` returns the addition of the variant array X from each worker. The result `S` is replicated on all workers.

`S = gplus(X,targetlab)` performs the addition, and places the result into `S` only on the worker indicated by `targetlab`. `S` is set to `[]` on all other workers.

Examples

With four workers,

```
S = gplus(labindex)
```

calculates `S = 1 + 2 + 3 + 4`, and returns `10` on all four workers.

See Also

gop | labindex

Introduced in R2006b

gpuDeviceCount

Package: `parallel.gpu`

Number of GPU devices present

Syntax

```
n = gpuDeviceCount
```

Description

`n = gpuDeviceCount` returns the number of GPU devices found in your computer.

Examples

Determine how many GPU devices you have available in your computer and examine the properties of each.

```
n = gpuDeviceCount;
for ii = 1:n
    gpuDevice(ii)
end
```

See Also

`arrayfun` | `feval` | `gpuDevice` | `parallel.gpu.CUDAKernel`

Topics
"GPU Support by Release" on page 8-38

Introduced in R2010b

gpurng

Control random number generation for GPU calculations

Syntax

```
gpurng(seed)
gpurng('shuffle')
gpurng(seed,generator)
gpurng('shuffle',generator)
gpurng('default')
S = gpurng
gpurng(S)
S = gpurng( ___ )
```

Description

gpurng(seed) sets the starting point, or seed, of the random number generator used in GPU calculations, so that rand, randi, and randn produce predictable sequences of numbers.

gpurng('shuffle') sets the seed of the random number generator based on the current time so that rand, randi, and randn produce different sequences of numbers after each time you call gpurng.

gpurng(seed,generator) or gpurng('shuffle',generator) selects the type of random number generator used by rand, randi, and randn.

gpurng('default') returns the settings of the random number generator to their default values. The random numbers produced are the same as if you had restarted MATLAB. The default setting is the Threefry generator with seed 0.

S = gpurng returns the current state of the random number generator as a structure with fields 'Type', 'Seed', and 'State'. Use this structure to restore the random number generator to the captured settings at a later time with gpurng(S).

gpurng(S) restores the state of the random number generator using settings previously captured with S = gpurng.

S = gpurng(___) returns the current state of the random number generator as a structure before changing the settings of the seed or generator type.

Examples

Create Predictable Arrays of Random Numbers on the GPU and CPU

Capture the GPU generator settings, and set the state of the CPU random number generator to match the GPU generator settings. Create predictable arrays of random numbers on the CPU and GPU.

Restore the generator type and seed to their default values on both the CPU and the GPU.

```
gpurng('default')
rng('default')
```

Save the default seed and generator type of the GPU random number generator.

```
GPUdef = gpurng
```

```
GPUdef = struct with fields:
     Type: 'threefry'
     Seed: 0
    State: [17×1 uint32]
```

Set the CPU random number generator to match the default GPU settings.

```
rng(GPUdef)
```

Create an array of uniformly distributed random numbers on the GPU.

```
rGPU = rand(1,10, 'gpuArray')
```

```
rGPU =

    0.3640    0.5421    0.6543    0.7436    0.0342    0.8311    0.7040    0.2817    0.1163    0.5
```

Create an array of random numbers on the CPU.

```
rCPU = rand(1,10)
```

```
rCPU = 1×10

    0.3640    0.5421    0.6543    0.7436    0.0342    0.8311    0.7040    0.2817    0.1163    0.5
```

The seed and generator type are the same for both the GPU and the CPU, so the arrays are the same.

```
isequal(rGPU,rCPU)
```

```
ans = logical
   1
```

The `gpurng` state does not save the settings for the transformation applied to generate a normally distributed set of random numbers. Even though the seed and the generator type are the same on the GPU and the CPU, the set of normally distributed random numbers is different.

```
nGPU = randn(1,1000, 'gpuArray');
nCPU = randn(1,1000);

figure
hold on
histogram(nGPU)
histogram(nCPU)
legend('GPU','CPU')
title('Normally Distributed Random Numbers')
xlabel('Value')
ylabel('Count')
hold off
```
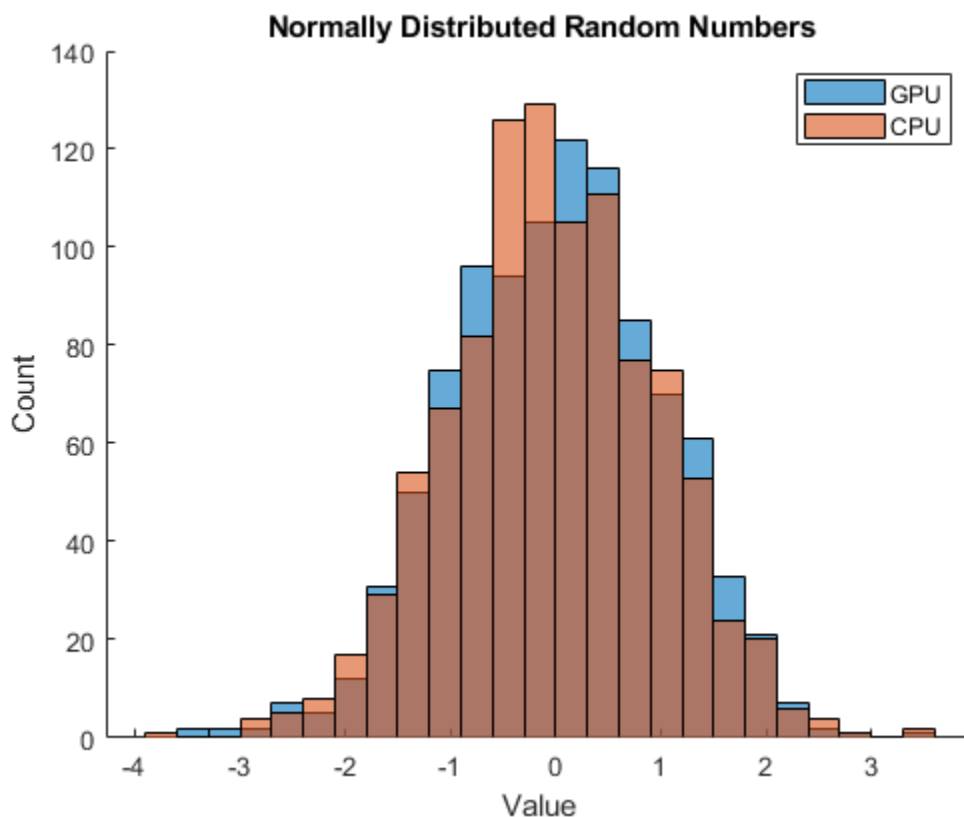
The statistics of the normal distribution of random numbers are the same on the GPU and the CPU.

By default, the CPU uses the `'Ziggurat'` transformation, while the GPU uses the `'BoxMuller'` algorithm for the `'Threefry'` generator. The only transformation method supported on both the CPU and GPU is the `'Inversion'` transform.

You can change the transformation method on the GPU using `parallel.gpu.RandStream`.

Input Arguments

seed — Random number seed
0 (default) | nonnegative integer

Random number seed, specified as a nonnegative integer. The seed specifies the starting point for the algorithm to generate random numbers. Specify the seed when you want reproducible results. The default seed is `0`.

Example: `gpurng(7)`

generator — Random number generator
`'Threefry'` (default) | character vector | string

Random number generator, specified as a character vector or string for any valid random number generator that supports multiple streams and substreams. Three random number generator algorithms are supported on the GPU.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period in Full Precision
`'Threefry'` or `'Threefry4x64_20'`	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})
`'Philox'` or `'Philox4x32_10'`	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
`'CombRecursive'` or `'mrg32k3a'`	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})

The default generator is `Threefry`.

For more information on the differences between generating random numbers on the GPU and CPU, see "Control Random Number Streams on Workers" on page 5-29.

Example: `gpurng(0,'Philox')`

S — Previous random number generator state
structure

Previous random number generator state, specified as a structure previously created using `S = gpurng`.

Example: `S = gpurng` captures the current state of the random number generator, and `gpurng(S)` restores the generator to those settings.

Data Types: `struct`

Output Arguments

S — Random number generator state
structure

Random number generator state, returned as a structure with the fields `'Type'`, `'Seed'`, and `'State'`.

Example: `S = gpurng` captures the current state of the random number generator, and `gpurng(S)` restores the generator to those settings.

Data Types: `struct`

Compatibility Considerations

Default random number generator change for `gpurng`
Behavior changed in R2019a

Starting in R2019a, the default random number generator for parallel computations is changed to `Threefry`. This generator offers performance enhancements for parallel calculations over the previous default. In releases up to R2018b, the default random number generator for parallel computations is `CombRecursive`.

With a different default generator, MATLAB generates different random numbers sequences by default in the context of parallel computations. However, statistics of these calculations remain unaffected. Therefore, you might want to update any code that relies on the specific random numbers being generated, but most calculations on the random numbers are unaffected.

To set the generator to the settings used by default in R2018b and earlier on GPU arrays, use the following command.

```
gpurng(0,"CombRecursive")
```

parallel.gpu.rng is renamed to gpurng
Behavior changed in R2018a

Starting in R2018a, the function `parallel.gpu.rng` is renamed to `gpurng`. Replace all instances of `parallel.gpu.rng` with `gpurng`.

`parallel.gpu.rng` will continue to work but is not recommended.

See Also
`gpuArray` | `parallel.gpu.RandStream` | `rng`

Topics
"Random Number Streams on a GPU" on page 8-6

Introduced in R2011b

gputimeit

Time required to run function on GPU

Syntax

```
t = gputimeit(F)
t = gputimeit(F,N)
```

Description

`t = gputimeit(F)` measures the typical time (in seconds) required to run the function specified by the function handle F. The function handle accepts no external input arguments, but can be defined with input arguments to its internal function call.

`t = gputimeit(F,N)` calls F to return N output arguments. By default, `gputimeit` calls the function F with one output argument, or no output arguments if F does not return any output.

Examples

Measure the time to calculate `sum(A.' .* B, 1)` on a GPU, where A is a 12000-by-400 matrix and B is 400-by-12000.

```
A = rand(12000,400,'gpuArray');
B = rand(400,12000,'gpuArray');
f = @() sum(A.' .* B, 1);
t = gputimeit(f)
```

```
0.0026
```

Compare the time to run `svd` on a GPU, with one versus three output arguments.

```
X = rand(1000,'gpuArray');
f = @() svd(X);
t3 = gputimeit(f,3)
```

```
1.0622
```

```
t1 = gputimeit(f,1)
```

```
0.2933
```

Tips

`gputimeit` is preferable to `timeit` for functions that use the GPU, because it ensures that all operations on the GPU have finished before recording the time and compensates for the overhead. For operations that do not use a GPU, `timeit` offers greater precision.

Note the following limitations:

- The function F should not call `tic` or `toc`.

- You cannot use `tic` and `toc` to measure the execution time of `gputimeit` itself.

See Also
`gpuArray | wait (GPUDevice)`

Introduced in R2013b

help

Help for toolbox functions in Command Window

Syntax

help *class/function*

Arguments

class	A Parallel Computing Toolbox object class, for example, `parallel.cluster`, `parallel.job`, or `parallel.task`.
function	A function or property of the specified class. To see what functions or properties are available for a class, see the `methods` or `properties` reference page.

Description

help *class/function* returns command-line help for the specified function of the given class.

If you do not know the class for the function, use `class(obj)`, where *function* is of the same class as the object `obj`.

Examples

Get help on functions or properties from Parallel Computing Toolbox object classes.

```
help parallel.cluster/createJob
help parallel.job/cancel
help parallel.task/wait

c = parcluster();
j1 = createJob(c);
class(j1)
```

```
parallel.job.CJSIndependentJob
```

```
help parallel.job/createTask
help parallel.job/AdditionalPaths
```

See Also
methods

Introduced before R2006a

Inf

Array of infinity

Syntax

```
A = Inf(sz,arraytype)
A = Inf(sz,datatype,arraytype)

A = Inf(sz,'like',P)
A = Inf(sz,datatype,'like',P)

C = Inf(sz,codist)
C = Inf(sz,datatype,codist)
C = Inf(sz,___,codist,'noCommunication')
C = Inf(sz,___,codist,'like',P)
```

Description

`A = Inf(sz,arraytype)` creates a matrix with underlying class of double, with `Inf` values in all elements.

`A = Inf(sz,datatype,arraytype)` creates a matrix with underlying class of *datatype*, with `Inf` values in all elements.

The size and type of array are specified by the argument options according to the following table.

Argument	Values	Descriptions
sz	n	Specifies size as an n-by-n matrix.
	m,n or [m n]	Specifies size as an m-by-n matrix.
	m,n,...,k or [m n ... k]	Specifies size as an m-by-n-by-...-by-k array.
arraytype	'distributed'	Specifies distributed array.
	'codistributed'	Specifies codistributed array, using the default distribution scheme.
	'gpuArray'	Specifies gpuArray.
datatype	'double' (default), 'single'	Specifies underlying class of the array, i.e., the data type of its elements.

`A = Inf(sz,'like',P)` creates an array of `Inf` values with the same type and underlying class (data type) as array P.

`A = Inf(sz,datatype,'like',P)` creates an array of `Inf` values with the specified underlying class (*datatype*), and the same type as array P.

`C = Inf(sz,codist)` or `C = Inf(sz,datatype,codist)` creates a codistributed array of `Inf` values with the specified size and underlying class (the default *datatype* is `'double'`). The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For

information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = Inf(8,codistributor1d());
end
```

`C = Inf(sz, ___ ,codist,'noCommunication')` specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

`C = Inf(sz, ___ ,codist,'like',P)` creates a codistributed array of `Inf` values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array `P`.

Examples

Create Distributed Inf Matrix

Create a 1000-by-1000 distributed array of `Inf`s with underlying class double:

```
D = Inf(1000,'distributed');
```

Create Codistributed Inf Matrix

Create a 1000-by-1000 codistributed double matrix of `Inf`s, distributed by its second dimension (columns).

```
spmd(4)
    C = Inf(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of `C`.

Create a 1000-by-1000 codistributed `single` matrix of `Inf`s, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = Inf(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-`labindex` local piece of `C`.

Create gpuArray Inf Matrix

Create a 1000-by-1000 gpuArray of `Inf`s with underlying class `double`:

```
G = Inf(1000,'double','gpuArray');
```

See Also
`Inf` | `NaN` | `eye` | `false` | `ones` | `true` | `zeros`

isaUnderlying

(Not recommended) True if distributed array's underlying elements are of specified class

---

Note `isaUnderlying` is not recommended. Use `isUnderlyingType` instead. For more information, see "Compatibility Considerations".

Syntax

```
TF = isaUnderlying(D, 'classname')
```

Description

`TF = isaUnderlying(D, 'classname')` returns true if the elements of distributed or codistributed array D are either an instance of *classname* or an instance of a class derived from *classname*. `isaUnderlying` supports the same values for *classname* as the MATLAB `isa` function does.

Examples

```
N = 1000;
D_uint8  = ones(1,N,'uint8','distributed');
D_cell   = distributed.cell(1,N);
isUint8  = isaUnderlying(D_uint8,'uint8') % returns true
isDouble = isaUnderlying(D_cell,'double') % returns false
```

Compatibility Considerations

classUnderlying and isaUnderlying are not recommended
Not recommended starting in R2020b

`classUnderlying` and `isaUnderlying` are not recommended. Use `underlyingType` and `isUnderlyingType` instead.

See Also
`isUnderlyingType` | `isa` | `mustBeUnderlyingType` | `underlyingType`

Introduced in R2010a

iscodistributed

True for codistributed array

Syntax

```
tf = iscodistributed(X)
```

Description

`tf = iscodistributed(X)` returns `true` for a codistributed array, or `false` otherwise. For a description of codistributed arrays, see "Nondistributed Versus Distributed Arrays" on page 4-2.

Examples

With a running parallel pool,

```
spmd
    L = ones(100,1);
    D = ones(100,1,'codistributed');
    iscodistributed(L) % returns false
    iscodistributed(D) % returns true
end
```

See Also

```
isdistributed
```

Introduced in R2009b

isComplete

True if codistributor object is complete

Syntax

```
tf = isComplete(codist)
```

Description

`tf = isComplete(codist)` returns `true` if `codist` is a completely defined codistributor, or `false` otherwise. For a description of codistributed arrays, see "Nondistributed Versus Distributed Arrays" on page 4-2.

See Also
codistributed | codistributor

Introduced in R2009b

isdistributed

True for distributed array

Syntax

```
tf = isdistributed(X)
```

Description

`tf = isdistributed(X)` returns `true` for a distributed array, or `false` otherwise. For a description of a distributed array, see "Nondistributed Versus Distributed Arrays" on page 4-2.

Examples

With a running parallel pool,

```
L = ones(100,1);
D = ones(100,1,'distributed');
isdistributed(L) % returns false
isdistributed(D) % returns true
```

See Also
iscodistributed

Introduced in R2006b

isequal

Package: parallel

True if clusters have same property values

Syntax

```
isequal(C1,C2)
isequal(C1,C2,C3,...)
```

Description

isequal(C1,C2) returns logical 1 (true) if clusters C1 and C2 have the same property values, or logical 0 (false) otherwise.

isequal(C1,C2,C3,...) returns true if all clusters are equal. isequal can operate on arrays of clusters. In this case, the arrays are compared element by element.

When comparing clusters, isequal does not compare the contents of the clusters' Jobs property.

Examples

Compare clusters after some properties are modified.

```
c1 = parcluster('local');
c1.NumWorkers = 2;          % Modify cluster
c1.saveAsProfile('local2') % Create new profile
c2 = parcluster('local2'); % Make cluster from new profile
isequal(c1,c2)

1

c0 = parcluster('local')   % Use original profile
isequal(c0,c1)

0
```

See Also
parcluster

Introduced in R2012a

isequal

Package: `parallel`

True if futures have same ID

Syntax

```
eq = isequal(F1,F2)
```

Description

`eq = isequal(F1,F2)` returns logical `1` (`true`) if futures `F1` and `F2` have the same `ID` property value, or logical `0` (`false`) otherwise.

Examples

Compare future object in workspace to queued future object.

```
p = parpool('local',2);
q = p.FevalQueue;
Fp = parfevalOnAll(p,@pause,0,30);
F1 = parfeval(p,@magic,1,10);
F2 = q.QueuedFutures;
isequal(F1,F2)
```

```
1
```

See Also
`cancel` | `fetchNext` | `fetchOutputs` | `parfeval` | `wait`

Introduced in R2014a

isgpuarray

Determine whether input is `gpuArray`

Syntax

`TF = isgpuarray(X)`

Description

`TF = isgpuarray(X)` returns logical 1 (`true`) if X is a `gpuArray`, and logical 0 (`false`) otherwise. You can use this function with an `if` statement to avoid executing code that expects `gpuArray` input.

Examples

Determine if Array is `gpuArray`

Create an array of random numbers.

`X = rand(3,3);`

Copy the array onto the GPU.

`Y = gpuArray(X);`

Use the function `isgpuArray` to verify that Y is a `gpuArray`.

`isgpuArray(Y)`

```
ans =
    1
```

Verify that X is not a `gpuArray`.

`isgpuarray(X)`

```
ans =
    0
```

Input Arguments

X — Input variable
workspace variable

Input variable, specified as a workspace variable. X can be any data type.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

See Also

gpuArray | canUseGPU | existsOnGPU | gather

Topics
"Establish Arrays on a GPU" on page 8-3
"Run MATLAB Functions on a GPU" on page 8-9

Introduced in R2020b

isreplicated

True for replicated array

Syntax

```
tf = isreplicated(X)
```

Description

`tf = isreplicated(X)` returns `true` for a replicated array, or `false` otherwise. For a description of a replicated array, see "Nondistributed Versus Distributed Arrays" on page 4-2. `isreplicated` also returns true for a Composite X if all its elements are identical.

Examples

With an open parallel pool,

```
spmd
    A = magic(3);
    t = isreplicated(A) % returns t = true
    B = magic(labindex);
    f = isreplicated(B) % returns f = false
end
```

Tips

`isreplicated(X)` requires checking for equality of the array X across all workers. This might require extensive communication and time. `isreplicated` is most useful for debugging or error checking small arrays. A codistributed array is not replicated.

See Also

iscodistributed | isdistributed

Introduced in R2006b

jobStartup

File for user-defined options to run when job starts

Syntax

```
jobStartup(job)
```

Arguments

job	The job for which this startup is being executed.

Description

`jobStartup(job)` runs automatically on a worker the first time that worker evaluates a task for a particular job. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the `jobStartup.m` file to define job initialization actions on the worker. The worker looks for `jobStartup.m` in the following order, executing the one it finds first:

1 Included in the job's `AttachedFiles` property.
2 In a folder included in the job's `AdditionalPaths` property.
3 In the worker's MATLAB installation at the location

   matlabroot/toolbox/parallel/user/jobStartup.m

To create a version of `jobStartup.m` for `AttachedFiles` or `AdditionalPaths`, copy the provided file and modify it as required. For further details on `jobStartup` and its implementation, see the text in the installed `jobStartup.m` file.

See Also
poolStartup | taskFinish | taskStartup

Introduced before R2006a

labBarrier

Block execution until all workers reach this call

Syntax

```
labBarrier
```

Description

`labBarrier` blocks execution of a parallel algorithm until all workers have reached the call to `labBarrier`. This is useful for coordinating access to shared resources such as file I/O.

Examples

Synchronize Workers for Timing

When timing code execution on the workers, use `labBarrier` to ensure all workers are synchronized and start their timed work together.

```
labBarrier;
tic
    A = rand(1,1e7,'codistributed');
distTime = toc;
```

See Also
labBroadcast | labReceive | labSend | labSendReceive

Introduced before R2006a

labBroadcast

Send data to all workers or receive data sent to all workers

Syntax

```
shared_data = labBroadcast(srcWkrIdx,data)
shared_data = labBroadcast(srcWkrIdx)
```

Arguments

srcWkrIdx	The labindex of the worker sending the broadcast.
data	The data being broadcast. This argument is required only for the worker that is broadcasting. The absence of this argument indicates that a worker is receiving.
shared_data	The broadcast data as it is received on all other workers.

Description

shared_data = labBroadcast(srcWkrIdx,data) sends the specified data to all executing workers. The data is broadcast from the worker with labindex == srcWkrIdx, and is received by all other workers.

shared_data = labBroadcast(srcWkrIdx) receives on each executing worker the specified shared_data that was sent from the worker whose labindex is srcWkrIdx.

If labindex is not srcWkrIdx, then you do not include the data argument. This indicates that the function is to receive data, not broadcast it. The received data, shared_data, is identical on all workers.

This function blocks execution until the worker's involvement in the collective broadcast operation is complete. Because some workers may complete their call to labBroadcast before others have started, use labBarrier if you need to guarantee that all workers are at the same point in a program.

Examples

In this case, the broadcaster is the worker whose labindex is 1.

```
srcWkrIdx = 1;
if labindex == srcWkrIdx
  data = randn(10);
  shared_data = labBroadcast(srcWkrIdx,data);
else
  shared_data = labBroadcast(srcWkrIdx);
end
```

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

If `data` is a `gpuArray`, data transfer between multiple GPUs in a parallel pool uses fast peer-to-peer communication, including NVLink, if available.

See Also
`labBarrier` | `labSendReceive` | `labindex`

Introduced before R2006a

labindex

Index of this worker

Syntax

```
id = labindex
```

Description

`id = labindex` returns the index of the worker currently executing the function. `labindex` is assigned to each worker when a job begins execution, and applies only for the duration of that job. The value of `labindex` spans from 1 to n, where n is the number of workers running the current job, defined by `numlabs`.

Examples

Obtain Worker Index with `labindex`

View `labindex` in `spmd` blocks and `parfor`-loops.

```
p = parpool('local',2);
spmd
    labindex
end

Lab 1:
        1

Lab 2:
        2
```

Using the same two-worker pool, `p`:

```
parfor a=1:4
    [a,labindex]
end

ans =
    3    1
ans =
    2    1
ans =
    1    1
ans =
    4    1
```

Tips

In an `spmd` block, because you have access to all workers individually and control what gets executed on them, each worker has a unique `labindex`.

However, inside a `parfor`-loop, `labindex` always returns a value of 1 on all workers in all iterations.

See Also

labSendReceive | numlabs

Introduced before R2006a

labProbe

Test to see if messages are ready to be received from other worker

Syntax

```
isDataAvail = labProbe
isDataAvail = labProbe(srcWkrIdx)
isDataAvail = labProbe('any',tag)
isDataAvail = labProbe(srcWkrIdx,tag)
[isDataAvail,srcWkrIdx,tag] = labProbe
```

Arguments

srcWkrIdx	labindex of a particular worker from which to test for a message.
tag	Tag defined by the sending worker's labSend function to identify particular data.
'any'	Character vector to indicate that all workers should be tested for a message.
isDataAvail	Logical indicating if a message is ready to be received.

Description

isDataAvail = labProbe returns a logical value indicating whether any data is available for this worker to receive with the labReceive function.

isDataAvail = labProbe(srcWkrIdx) tests for a message only from the specified worker.

isDataAvail = labProbe('any',tag) tests only for a message with the specified tag, from any worker.

isDataAvail = labProbe(srcWkrIdx,tag) tests for a message from the specified worker and tag.

[isDataAvail,srcWkrIdx,tag] = labProbe returns labindex of the workers and tags of ready messages. If no data is available, srcWkrIdx and tag are returned as [].

See Also
labReceive | labSend | labSendReceive | labindex

Introduced before R2006a

labReceive

Receive data from another worker

Syntax

```
data = labReceive
data = labReceive(srcWkrIdx)
data = labReceive('any',tag)
data = labReceive(srcWkrIdx,tag)
[data,srcWkrIdx,tag] = labReceive
```

Arguments

srcWkrIdx	labindex of a particular worker from which to receive data.
tag	Tag defined by the sending worker's labSend function to identify particular data.
'any'	Character vector to indicate that data can come from any worker.
data	Data sent by the sending worker's labSend function.

Description

data = labReceive receives data from any worker with any tag.

data = labReceive(srcWkrIdx) receives data from the specified worker with any tag

data = labReceive('any',tag) receives data from any worker with the specified tag.

data = labReceive(srcWkrIdx,tag) receives data from only the specified worker with the specified tag.

[data,srcWkrIdx,tag] = labReceive returns the source worker labindex and tag with the data.

Tips

This function blocks execution in the worker until the corresponding call to labSend occurs in the sending worker.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

If data is a gpuArray, data transfer between multiple GPUs in a parallel pool uses fast peer-to-peer communication, including NVLink, if available.

See Also

labBarrier | labProbe | labSend | labSendReceive | labindex

Introduced before R2006a

labSend

Send data to another worker

Syntax

```
labSend(data,rcvWkrIdx)
labSend(data,rcvWkrIdx,tag)
```

Arguments

data	Data sent to the other workers; any MATLAB data type.
rcvWkrIdx	labindex of receiving worker or workers.
tag	Nonnegative integer to identify data.

Description

labSend(data,rcvWkrIdx) sends the data to the specified destination. data can be any MATLAB data type. rcvWkrIdx identifies the labindex of the receiving worker, and must be either a scalar or a vector of integers between 1 and numlabs; it cannot be labindex of the current (sending) worker.

labSend(data,rcvWkrIdx,tag) sends the data to the specified destination with the specified tag value. tag can be any integer from 0 to 32767, with a default of 0.

Tips

This function might or might not return before the corresponding labReceive completes in the receiving worker.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" on page 8-9.

If data is a gpuArray, data transfer between multiple GPUs in a parallel pool uses fast peer-to-peer communication, including NVLink, if available.

See Also
labBarrier | labProbe | labReceive | labSendReceive | labindex | numlabs

Introduced before R2006a

labSendReceive

Simultaneously send data to and receive data from another worker

Syntax

```
dataReceived = labSendReceive(rcvWkrIdx,srcWkrIdx,dataSent)
dataReceived = labSendReceive(rcvWkrIdx,srcWkrIdx,dataSent,tag)
```

Arguments

dataSent	Data on the sending worker that is sent to the receiving worker; any MATLAB data type.
dataReceived	Data accepted on the receiving worker.
rcvWkrIdx	labindex of the receiving worker to which data is sent.
srcWkrIdx	labindex of the source worker from which data is sent.
tag	Nonnegative integer to identify data.

Description

`dataReceived = labSendReceive(rcvWkrIdx,srcWkrIdx,dataSent)` sends `dataSent` to the worker whose `labindex` is `rcvWkrIdx`, and receives `dataReceived` from the worker whose `labindex` is `srcWkrIdx`. The values for arguments `rcvWkrIdx` and `srcWkrIdx` must be scalars. This function is conceptually equivalent to the following sequence of calls:

```
labSend(dataSent,rcvWkrIdx);
dataReceived = labReceive(srcWkrIdx);
```

with the important exception that both the sending and receiving of data happens concurrently. This can eliminate deadlocks that might otherwise occur if the equivalent call to `labSend` would block.

If `rcvWkrIdx` is an empty array, `labSendReceive` does not send data, but only receives. If `srcWkrIdx` is an empty array, `labSendReceive` does not receive data, but only sends.

`dataReceived = labSendReceive(rcvWkrIdx,srcWkrIdx,dataSent,tag)` uses the specified tag for the communication. `tag` can be any integer from `0` to `32767`.

Examples

Create a unique set of data on each worker, and transfer each worker's data one worker to the right (to the next higher `labindex`).

First use the `magic` function to create a unique value for the variant array `mydata` on each worker.

```
mydata = magic(labindex)
```

```
Lab 1:
  mydata =
       1
```

```
Lab 2:
  mydata =
        1      3
        4      2
Lab 3:
  mydata =
        8      1      6
        3      5      7
        4      9      2
```

Define the worker on either side, so that each worker will receive data from the worker on its "left," while sending data to the worker on its "right," cycling data from the end worker back to the beginning worker.

```
rcvWkrIdx = mod(labindex, numlabs) + 1; % one worker to the right
srcWkrIdx = mod(labindex - 2, numlabs) + 1; % one worker to the left
```

Transfer the data, sending each worker's `mydata` into the next worker's `otherdata` variable, wrapping the third worker's data back to the first worker.

```
otherdata = labSendReceive(rcvWkrIdx,srcWkrIdx,mydata)
```

```
Lab 1:
  otherdata =
        8      1      6
        3      5      7
        4      9      2
Lab 2:
  otherdata =
        1
Lab 3:
  otherdata =
        1      3
        4      2
```

Transfer data to the next worker without wrapping data from the last worker to the first worker.

```
if labindex < numlabs; rcvWkrIdx = labindex + 1; else rcvWkrIdx = []; end;
if labindex > 1; srcWkrIdx = labindex - 1; else srcWkrIdx = []; end;
otherdata = labSendReceive(rcvWkrIdx,srcWkrIdx,mydata)
```

```
Lab 1:
  otherdata =
       []
Lab 2:
  otherdata =
        1
Lab 3:
  otherdata =
        1      3
        4      2
```

See Also

labBarrier | labProbe | labReceive | labSend | labindex | numlabs

Introduced in R2006b

length

Length of object array

Syntax

```
length(obj)
```

Arguments

obj	An object or an array of objects.

Description

`length(obj)` returns the length of `obj`. It is equivalent to the command `max(size(obj))`.

Examples

Examine how many tasks are in the job `j1`.

```
length(j1.Tasks)

ans =
     9
```

Introduced before R2006a

listAutoAttachedFiles

List of files automatically attached to job, task, or parallel pool

Syntax

```
listAutoAttachedFiles(obj)
```

Description

`listAutoAttachedFiles(obj)` performs a dependency analysis on all the task functions, or on the batch job script or function. Then it displays a list of the code files that are already or going to be automatically attached to the job or task object `obj`.

If `obj` is a parallel pool, the output lists the files that have already been attached to the parallel pool following an earlier dependency analysis. The dependency analysis runs if a `parfor` or `spmd` block errors due to an undefined function. At that point any files, functions, or scripts needed by the `parfor` or `spmd` block are attached if possible.

Examples

Automatically Attach Files via Cluster Profile

Employ a cluster profile to automatically attach code files to a job. Set the `AutoAttachFiles` property for a job in the cluster's profile. If this property value is true, then all jobs you create on that cluster with this profile will have the necessary code files automatically attached. This example assumes that the cluster profile `myAutoCluster` has that setting.

Create batch job, applying your cluster.

```
obj = batch(myScript,'profile','myAutoCluster');
```

Verify attached files by viewing list.

```
listAutoAttachedFiles(obj)
```

Automatically Attach Files Programmatically

Programmatically set a job to automatically attach code files, and then view a list of those files for one of the tasks in the job.

```
c = parcluster(); % Use default profile
j = createJob(c);
j.AutoAttachFiles = true;
obj = createTask(j,myFun,OutNum,ArgCell);
listAutoAttachedFiles(obj) % View attached list
```

The files returned in the output listing are those that analysis has determined to be required for the workers to evaluate the function `myFun`, and which automatically attach to the job.

Input Arguments

`obj` — Job, task, or pool to which files automatically attach
job object | task object | parallel pool object

Job, task, or pool to which code files are automatically attached, specified as a parallel.Job, parallel.Task, or parallel.Pool object. The `AutoAttachFiles` property of the job object must be `true`; if the input is a task object, then this applies to its parent job object.

Example: `obj = createJob(cluster);`

Example: `obj = gcp`

See Also

batch | createCommunicatingJob | createJob | createTask | parcluster | parpool

Topics
"Add and Modify Cluster Profiles" on page 5-14

Introduced in R2013a

load

Package: `parallel`

Load workspace variables from batch job

Syntax

```
load(job)
load(job,'X')
load(job,'X','Y','Z*')
load(job,'-regexp','PAT1','PAT2')
S = load(job ...)
```

Arguments

`job`	Job from which to load workspace variables.
`'X' , 'Y', 'Z*'`	Variables to load from the job. Wildcards allow pattern matching in MAT-file style.
`'-regexp'`	Indication to use regular expression pattern matching.
`S`	Struct containing the variables after loading.

Description

`load(job)` retrieves all variables from a batch job and assigns them into the current workspace. `load` throws an error if the batch runs a function (instead of a script), the job is not finished, or the job encountered an error while running, .

`load(job,'X')` loads only the variable named `X` from the job.

`load(job,'X','Y','Z*')` loads only the specified variables. The wildcard `'*'` loads variables that match a pattern (MAT-file only).

`load(job,'-regexp','PAT1','PAT2')` can be used to load all variables matching the specified patterns using regular expressions. For more information on using regular expressions, type `doc regexp` at the command prompt.

`S = load(job ...)` returns the contents of `job` into variable `S`, which is a struct containing fields matching the variables retrieved.

Examples

Run a batch job and load its results into your client workspace.

```
j = batch('myScript');
wait(j)
load(j)
```

Load only variables whose names start with `'a'`.

```
load(job,'a*')
```

Load only variables whose names contain any digits.

```
load(job,'-regexp','\d')
```

See Also
batch | fetchOutputs

Introduced in R2008a

logout

Log out of MATLAB Job Scheduler cluster

Syntax

```
logout(c)
```

Description

`logout(c)` logs you out of the MATLAB Job Scheduler cluster specified by cluster object `c`. Any subsequent call to a privileged action requires you to re-authenticate with a valid password. Logging out might be useful when you are finished working on a shared machine.

See Also

`changePassword`

Introduced in R2012a

mapreducer

Define parallel execution environment for mapreduce and tall arrays

Syntax

```
mapreducer
mapreducer(0)
mapreducer(poolobj)
mapreducer(hadoopCluster)
mapreducer(mr)
mr = mapreducer( ___ )
mr = mapreducer( ___ ,'ObjectVisibility','Off')
```

Description

`mapreducer` defines the execution environment for `mapreduce` or tall arrays. Use the `mapreducer` function to change the execution environment to use a different cluster or to switch between serial and parallel development.

The default execution environment uses either the local MATLAB session, or a parallel pool if you have Parallel Computing Toolbox. If you have Parallel Computing Toolbox installed, when you use the `tall` or `mapreduce` functions, MATLAB automatically starts a parallel pool of workers, unless you have changed the default preferences. By default, a parallel pool uses local workers, typically one worker for each core in your machine. If you turn off the **Automatically create a parallel pool** option, then you must explicitly start a pool if you want to use parallel resources. See "Specify Your Parallel Preferences" on page 5-9.

When working with tall arrays, use `mapreducer` to set the execution environment prior to creating the tall array. Tall arrays are bound to the current global execution environment when they are constructed. If you subsequently change the global execution environment, then the tall array is invalid, and you must recreate it.

Note In MATLAB, you do not need to specify configuration settings using `mapreducer` because `mapreduce` algorithms and tall array calculations automatically run in the local MATLAB session only. If you also have Parallel Computing Toolbox, then you can use the additional `mapreducer` configuration options listed on this page for running in parallel. If you have MATLAB Compiler, then you can use separate `mapreducer` configuration options for running in deployed environments.

See: `mapreducer` in the MATLAB documentation, or `mapreducer` in the MATLAB Compiler documentation.

`mapreducer` with no input arguments creates a new `mapreducer` execution environment with all the defaults and sets this to be the current `mapreduce` or tall array execution environment. You can use `gcmr` to get the current mapreducer configuration.

- If you have default preferences (**Automatically create a parallel pool** is enabled), and you have not opened a parallel pool, then `mapreducer` opens a pool using the default cluster profile, sets `gcmr` to a mapreducer based on this pool and returns this mapreducer.

- If you have opened a parallel pool, then `mapreducer` sets `gcmr` to a mapreducer based on the current pool and returns this mapreducer.
- If you have disabled **Automatically create a parallel pool**, and you have not opened a parallel pool, then `mapreducer` sets `gcmr` to a mapreducer based on the local MATLAB session, and `mapreducer` returns this mapreducer.

`mapreducer(0)` specifies that `mapreduce` or `tall` array calculations run in the MATLAB client session without using any parallel resources.

`mapreducer(poolobj)` specifies a parallel pool for parallel execution of `mapreduce` or tall arrays. `poolobj` is a parallel.Pool object. The default pool is the current pool that is returned or opened by `gcp`.

`mapreducer(hadoopCluster)` specifies a Hadoop cluster for parallel execution of `mapreduce` or tall arrays. `hadoopCluster` is a parallel.cluster.Hadoop object.

`mapreducer(mr)` sets the global execution environment for `mapreduce` or tall arrays, using a previously created MapReducer object, `mr`, if its `ObjectVisibility` property is `'On'`.

`mr = mapreducer( ___ )` returns a MapReducer object to specify the execution environment. You can define several MapReducer objects, which enables you to swap execution environments by passing one as an input argument to `mapreduce` or `mapreducer`.

`mr = mapreducer( ___ ,'ObjectVisibility','Off')` hides the visibility of the MapReducer object, `mr`, using any of the previous syntaxes. Use this syntax to create new MapReducer objects without affecting the global execution environment of `mapreduce`.

Examples

Develop in Serial and Then Use Local Workers or Cluster

If you want to develop in serial and not use local workers or your specified cluster, enter:

`mapreducer(0);`

If you use `mapreducer` to change the execution environment after creating a tall array, then the tall array is invalid and you must recreate it. To use local workers or your specified cluster again, enter:

`mapreducer(gcp);`

mapreducer with Automatically Create a Parallel Pool Switched Off

If you have turned off the **Automatically create a parallel pool** option, then you must explicitly start a pool if you want to use parallel resources. See "Specify Your Parallel Preferences" on page 5-9 for details.

The following code shows how you can use `mapreducer` without input arguments to set the execution environment to your local MATLAB session and then specify a local parallel pool:

```
>> mapreducer
>> parpool('local',1);

Starting parallel pool (parpool) using the 'local' profile ... connected to 1 workers.

>> gather(min(tall(rand(1000,1))))
```

```
Evaluating tall expression using the Local MATLAB Session:
Evaluation completed in 0 sec

ans =

    5.2238e-04
```

Input Arguments

`poolobj` — Pool for parallel execution
`gcp` (default) | parallel.Pool object

Pool for parallel execution, specified as a parallel.Pool object.

Example: `poolobj = gcp`

`hadoopCluster` — Hadoop cluster for parallel execution
parallel.cluster.Hadoop object

Hadoop cluster for parallel execution, specified as a parallel.cluster.Hadoop object.

Example: `hadoopCluster = parallel.cluster.Hadoop`

Output Arguments

`mr` — Execution environment for `mapreduce` and tall arrays
MapReducer object

Execution environment for `mapreduce` and tall arrays, returned as a MapReducer object.

If the `ObjectVisibility` property of `mr` is set to `'On'`, then `mr` defines the default execution environment for all `mapreduce` algorithms and tall array calculations. If the `ObjectVisibility` property is `'Off'`, you can pass `mr` as an input argument to `mapreduce` to explicitly specify the execution environment for that particular call.

You can define several MapReducer objects, which enables you to swap execution environments by passing one as an input argument to `mapreduce` or `mapreducer`.

Tips

One of the benefits of developing your algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, then use `mapreducer` to scale up and take advantage of the capabilities offered by Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler, without needing to rewrite your algorithm.

See Also
`gcmr` | `gcp` | `mapreduce` | `parallel.cluster.Hadoop` | `tall`

Topics

Introduced in R2014b

methods

List functions of object class

Syntax

```
methods(obj)
out = methods(obj)
```

Arguments

obj	An object or an array of objects.
out	Cell array of vectors.

Description

methods(obj) returns the names of all methods for the class of which obj is an instance.

out = methods(obj) returns the names of the methods as a cell array of vectors.

Examples

Create cluster, job, and task objects, and examine what methods are available for each.

```
c = parcluster();
methods(c)

j1 = createJob(c);
methods(j1)

t1 = createTask(j1,@rand,1,{3});
methods(t1)
```

See Also
help

Introduced before R2006a

mexcuda

Compile MEX-function for GPU computation

Syntax

```
mexcuda filenames
mexcuda
mexcuda option1 ... optionN filenames
```

Description

`mexcuda filenames` compiles and links source files into a shared library called a MEX-file, executable from within MATLAB. The function compiles MEX-files written using the CUDA C++ framework with the NVIDIA `nvcc` compiler, allowing the files to define and launch GPU kernels. In addition, the `mexcuda` function exposes the GPU MEX API to allow the MEX-file to read and write gpuArrays.

`mexcuda` is an extension of the MATLAB `mex` function. Only a subset of the compilers supported by `mex` is supported for `mexcuda`. The supported compilers depend on the CUDA Toolkit version supported by MATLAB.

`mexcuda option1 ... optionN filenames` builds with the specified build options. The `option1 ... optionN` arguments supplement or override the default `mexcuda` build configuration. You can use the most of the options available in `mex` with `mexcuda`.

Examples

Compile Simple MEX-Function

Compile a simple MEX-function to create the function `myMexFunction` from a CUDA C++ source file.

```
mexcuda myMexFunction.cu
```

An example source file is available at *matlabroot*`/toolbox/parallel/gpu/extern/src/mex/mexGPUExample.cu`.

Display Detailed Build and Troubleshooting Information

Use verbose mode to display the compile and link commands and other information useful for troubleshooting.

```
mexcuda -v myMexFunction.cu
```

Compile and Link Multiple Source Files

Compile and link multiple source files with one command.

```
mexcuda myMexFunction.cu otherSource1.cpp otherSource2.cpp
```

Compile and Link in Two Stages

First compile, then link to create a function.

```
mexcuda -c myMexFunction.cu
mexcuda myMexFunction.obj
```

The first line compiles to `myMexFunction.obj` (Windows) or `myMexFunction.o` (UNIX), and the second links to create the function `myMexFunction`.

Compile with Dynamic Parallelism

Compile code that uses dynamic parallelism, defining kernels that launch other kernels.

```
mexcuda -dynamic myMexFunction.cu
```

Link to Third-Party Library

Compile a MEX-function that makes use of the CUDA image primitives library, `npp`, which is installed at `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\lib\x64\nppi.lib`.

```
mexcuda '-LC:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\lib\x64' -lnppi myMexFunction.cu
```

Input Arguments

`filenames` — One or more file names
character vector

One or more file names, including name and file extension, specified as a character vector. If the file is not in the current folder, specify the full path to the file. File names can be any combination of:

- C or C++ language source files
- object files
- library files

The first source code file listed in `filenames` is the name of the binary MEX-file. To override this naming convention, use the `'-output'` option.

Data Types: `char`

`option1 ... optionN` — One or more build options
character vectors corresponding to valid option flags

One or more build options, specified as one of these values. Options can appear in any order on any platform, except where indicated.

Most options available for the `mex` function are supported. In addition, the following options are also available.

Option	Description
`-dynamic`	Dynamic parallelism: compiles MEX-files that define kernels that launch other kernels.
`-G`	Generate debug information for device code. This makes it possible to step through kernel code line by line in one of NVIDIA's debugging applications (NSight or cuda-gdb). To enable debugging of host code use `-g`.

The following `mex` function option is not supported.

Option	Reason
`-compatibleArrayDims`	Use of the MATLAB large-array-handling API is implicit, and cannot be overridden.

All other options for `mex` are supported for `mexcuda`. See the documentation for `mex` for details.

Tips

- If the CUDA toolkit is not detected or is not a supported version, MATLAB compiles the CUDA code using the NVIDIA `nvcc` compiler installed with MATLAB. To check which compiler `mexcuda` is using, use the `-v` flag for verbose output in the `mexcuda` command.
- The CUDA toolkit installed with MATLAB does not contain all libraries that are available in the CUDA toolkit. If you want to link a specific library that is not installed with MATLAB, install the CUDA toolkit. You can check which CUDA toolkit version MATLAB requires using `gpuDevice`. For more information about the CUDA Toolkit, see "CUDA Toolkit" on page 8-39.

See Also

`mex`

Topics
"Accessing Advanced CUDA Features Using MEX" on page 9-203
"Run MEX-Functions Containing CUDA Code" on page 8-28

Introduced in R2015b

mpiLibConf

Location of MPI implementation

Syntax

```
[primaryLib,extras] = mpiLibConf
```

Arguments

primaryLib	MPI implementation library used by a communicating job.
extras	Cell array of other required library names.

Description

[primaryLib,extras] = mpiLibConf returns the MPI implementation library to be used by a communicating job. primaryLib is the name of the shared library file containing the MPI entry points. extras is a cell array of other library names required by the MPI library.

To supply an alternative MPI implementation, create a file named mpiLibConf.m, and place it on the MATLAB path. The recommended location is *matlabroot*/toolbox/parallel/user. Your mpiLibConf.m file must be higher on the cluster workers' path than *matlabroot*/toolbox/parallel/mpi. (Sending mpiLibConf.m as a file dependency for this purpose does not work.) After your mpiLibConf.m file is in place, update the toolbox path caching with the following command in MATLAB:

```
rehash toolboxcache
```

Examples

Use the mpiLibConf function to view the current MPI implementation library:

```
mpiLibConf
```

```
mpich2.dll
```

Tips

Under all circumstances, the MPI library must support all MPI-1 functions. Additionally, the MPI library must support null arguments to MPI_Init as defined in section 4.2 of the MPI-2 standard. The library must also use an mpi.h header file that is fully compatible with MPICH2.

When used with the MATLAB Job Scheduler or the local cluster, the MPI library must support the following additional MPI-2 functions:

- MPI_Open_port
- MPI_Comm_accept
- MPI_Comm_connect

When used with any third-party scheduler, it is important to launch the workers using the version of mpiexec corresponding to the MPI library being used. Also, you might need to launch the corresponding process management daemons on the cluster before invoking mpiexec.

See Also

rehash

Topics
"Use Different MPI Builds on UNIX Systems" (MATLAB Parallel Server)
"Toolbox Path Caching in MATLAB"

Introduced before R2006a

mpiprofile

Profile parallel communication and execution times

Syntax

```
mpiprofile
mpiprofile on <options>
mpiprofile off
mpiprofile reset
mpiprofile viewer
mpiprofile resume
mpiprofile clear
mpiprofile status
stats = mpiprofile('info')
mpiprofile('viewer',stats)
```

Description

`mpiprofile` enables or disables the parallel profiler data collection on a MATLAB worker running in a parallel pool. You can use `mpiprofile` either from the MATLAB client or directly from the worker from within an `spmd` block. When you run `mpiprofile` from the MATLAB client, `mpiprofile` performs the action on the MATLAB workers.

`mpiprofile` aggregates statistics on execution time and communication times. `mpiprofile` collects statistics in a manner similar to running the `profile` command on each MATLAB worker. By default, the parallel profiling extensions include array fields that collect information on communication with each of the other workers.

`mpiprofile on <options>` starts the parallel profiler and clears previously recorded profile statistics.

`mpiprofile` takes the following options.

Option	Description

`-messagedetail default`
 	
`-messagedetail simplified`
 	
`-messagedetail off`
	This option specifies the detail at which communication information is stored.
	
	`-messagedetail default` collects information on a per-worker instance.
	
	`-messagedetail simplified` turns off collection for *PerLab data fields, which reduces the profiling overhead. If you have a very large cluster, you might want to use this option; however, you will not get all the detailed inter-worker communication plots in the viewer.
	
	Note that changing `-messagedetail` will clear any previously stored data.
	
	For information about the structure of returned data, see `mpiprofile info` below.

`-history`
 	
`-nohistory`
 	
`-historysize <size>`
	`mpiprofile` supports these options in the same way as the standard `profile`.
	
	No other `profile` options are supported by `mpiprofile`. These three options have no effect on the data displayed by `mpiprofile viewer`.

`mpiprofile off` stops the parallel profiler. To reset the state of the profiler and disable collecting communication information, use `mpiprofile reset`.

`mpiprofile reset` turns off the parallel profiler and resets the data collection back to the standard profiler. If you do not call `reset`, subsequent `profile` commands will collect MPI information.

`mpiprofile viewer` stops the profiler and opens the graphical profile browser with parallel options. The output is an HTML report displayed in the profiler window. The file listing at the bottom of the function profile page shows several columns to the left of each line of code. In the summary page:

- Column 1 indicates the number of calls to that line.
- Column 2 indicates total time spent on the line in seconds.
- Columns 3–6 contain the communication information specific to the parallel profiler.

`mpiprofile resume` restarts the profiler without clearing previously recorded function statistics.

`mpiprofile clear` clears the profile information.

`mpiprofile status` returns the status of the parallel profiler.

`stats = mpiprofile('info')` stops the parallel profiler and returns a structure containing the profiler statistics. `stats` contains the same fields as returned by `profile('info')`, with the following additional fields in the `FunctionTable` entry. All these fields are recorded on a per-function and per-line basis, except for the *PerLab fields.

Field	Description
BytesSent	Records the quantity of data sent
BytesReceived	Records the quantity of data received
TimeWasted	Records communication waiting time
CommTime	Records the communication time
CommTimePerLab	Vector of communication receive time for each worker
TimeWastedPerLab	Vector of communication waiting time for each worker
BytesReceivedPerLab	Vector of data received from each worker

The three *PerLab fields are collected only on a per-function basis, and you can turn them off by typing the following command:

```
mpiprofile on -messagedetail simplified
```

When you run it from the MATLAB client, `stats = mpiprofile('info')` returns information from all workers. When you run it on a worker, `mpiprofile('info')` returns the profile information specific to that worker.

`mpiprofile('viewer',stats)` opens the graphical profile browser showing the profiling information contained in `stats`. You can use `stats = mpiprofile('info')` on the client to create the structure array.

`mpiprofile` does not accept `-timer clock` options, because the communication timer clock must be real.

For more information and examples on using the parallel profiler, see "Profiling Parallel Code" on page 5-32.

Examples

Profile Parallel Code

Turn on the profiler. With default preferences, turning on the profiler will create a parallel pool automatically if there is not one already created.

```
mpiprofile on
```

Run your parallel code.

```
A = rand(1000,'distributed');
b = sum(A, 2);
x = A\b;
```

Show the collected profile information.

```
mpiprofile viewer
```

Analyze Parallel Profiler Data

The parallel profiler collects information about the execution of code on each worker and the communications between the workers. After you profile your parallel code with `mpiprofile`, start the graphical viewer by calling `mpiprofile viewer`.

```
R = rand(1e3,'distributed');
mpiprofile on
R = R*R;
mpiprofile viewer
```

The viewer has three types of pages or *views*.

The parallel profile viewer, opens with the **Function Summary Report** first, in the same way as the standard profiler. In this view you can see profile information from any single lab or from multiple labs simultaneously. It initially shows all functions executed on lab 1. You can then choose via the listbox options to see profiling information from any lab.

In this view you can see Aggregate information using the **Manual Comparison Selection** listboxes. When you select an Aggregate in this view, the profiler accumulates and displays the specified data about all of the executed functions. For example,

- **max Time Aggregate** lists every function called in the program, and for each function, the data from the lab that spent the most time executing it.
- **min Time >0 Aggregate** lists every function called in the program, and for each function, the statistics from the lab that spent the least time executing it.
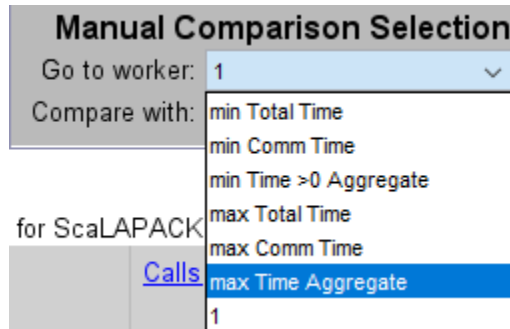
Here are a few things to keep in mind in this view:

- To re-sort the table by a different field simply click the related column title (e.g. **Total Comm Time**).
- To select a function and go to the Function Detail Report, click any function name that appears in the **Function Name** column.
- To compare profiling information from different labs, use the **Comparison** listboxes and buttons (found in the top of each page). Comparison information always displays in a maroon color font.
- To find which lab the main profiling data (black or blue text) comes from, look at the orange highlighted text at the top of the displayed page or in the top toolbar.

The **Function Detail Report** displays information on the individual lines executed inside the current function for each lab. This includes a **Busy Lines** table which can be used to compare the top five lines of code on different labs. This report is only fully functional if the profiled MATLAB files are available on the client MATLAB path.

The Function Detail Report is different from the Function Summary Report in several ways:

- The report is generated for one function at a time. The function name is displayed at the top of the page, in green or orange. Green highlighting indicates that the function spent very little (or no) time in communication. Orange highlighting indicates more than 20% of the time was spent in communication or waiting for communication.

- Every listbox option takes into account the last function you clicked. The current function can be changed, if need be, by clicking the **Home** button in the top toolbar, which also takes you back to the Function Summary Report.

- Profile information Aggregates from multiple labs are calculated only on a *per function* basis. Therefore in the Function Detail Report, selecting **max Time Aggregate** displays information from the one lab that took longest executing the current function.

- Comparisons of profiling information are only available in the top five lines shown in the **Busy Lines** table (the first table from the top unless there is a parents table).

The **Plot View** is shown whenever you click a plot option in the **Show Figures** listbox. The plots show communication and timing information from all the labs for the given function. There are two types of plots (Histograms and Per Worker Images). The **Plot Time Histograms** and **Plot All Per Worker Communication** options show three figures using the corresponding communication fields returned by the `mpiprofile info` command.

- **Plot Time Histograms** shows histograms for Total Time, Total Communication Time, and Communication Waiting Time.

- **Plot All Per Worker Communication** shows 2D color coded image plots for Data Received, Receive Communication Time, and Communication Waiting Time.

- The **Plot Communication Time Per Worker** option in the **Show Figures** listbox shows only the **Receive Communication Time** chart and therefore is faster to generate.

Plotting Per Function or Per Session

There are two modes for the plots:

- *Overall session* plots: If you are in the Function Summary Report *and* you have not selected any function the plots are generated for all code executed (with the profiler on).

- *Per function* plots : If you are in the Function Detail Report after having selected a function, clicking any of the available plots (in the listbox shown below) will take you to the function specific **Plot View**.

You can always tell which plot mode you are using by looking at the titles of each figure. The titles show the name of the current function (or *all functions* when showing data for the entire session). The titles also include the name of the profiling field being displayed (e.g., Communication Time, Total Time). If you want to see the data for the entire profiling session after having selected a specific function, click **Home** to go back to the Function Summary Report then select your desired plot (using the listbox shown below).
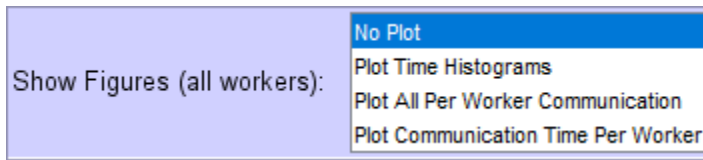
See Also

`mpiSettings` | `profile`

Topics
"Profile Parallel Code" on page 9-3

Introduced in R2007b

mpiSettings

Configure options for MPI communication

Syntax

```
mpiSettings('DeadlockDetection','on')
mpiSettings('MessageLogging','on')
mpiSettings('MessageLoggingDestination','CommandWindow')
mpiSettings('MessageLoggingDestination','stdout')
mpiSettings('MessageLoggingDestination','File','filename')
```

Description

`mpiSettings('DeadlockDetection','on')` turns on deadlock detection during calls to `labSend` and `labReceive`. If deadlock is detected, a call to `labReceive` might cause an error. Although it is not necessary to enable deadlock detection on all workers, this is the most useful option. The default value is `'off'` for communicating jobs, and `'on'` inside `spmd` statements. Once the setting has been changed within an `spmd` statement, the setting stays in effect until the parallel pool is closed.

If you are using a large number of workers, you might experience a performance increase by disabling deadlock detection.

If some workers do not call `labSend` or `labReceive` for long periods of times, deadlock detection can cause communication errors. If you encounter errors, try disabling deadlock detection.

`mpiSettings('MessageLogging','on')` turns on MPI message logging. The default is `'off'`. The default destination is the MATLAB Command Window.

`mpiSettings('MessageLoggingDestination','CommandWindow')` sends MPI logging information to the MATLAB Command Window. If the task within a communicating job is set to capture Command Window output, the MPI logging information will be present in the task's `CommandWindowOutput` property.

`mpiSettings('MessageLoggingDestination','stdout')` sends MPI logging information to the standard output for the MATLAB process. If you are using a MATLAB Job Scheduler, this is the mjs service log file.

`mpiSettings('MessageLoggingDestination','File','filename')` sends MPI logging information to the specified file.

Examples

Set deadlock detection for a communicating job inside the `jobStartup.m` file for that job:

```
% Inside jobStartup.m for the communicating job
mpiSettings('DeadlockDetection','on');
myLogFname = sprintf('%s_%d.log',tempname,labindex);
mpiSettings('MessageLoggingDestination','File',myLogFname);
mpiSettings('MessageLogging','on');
```

Turn off deadlock detection for all subsequent `spmd` statements that use the same parallel pool:

```
spmd;mpiSettings('DeadlockDetection','off');end
```

Tips

Setting the MessageLoggingDestination does not automatically enable message logging. A separate call is required to enable message logging.

mpiSettings has to be called on the worker, not the client. That is, it should be called within the task function, within jobStartup.m, or within taskStartup.m.

Introduced before R2006a

mxGPUCopyFromMxArray (C)

Copy mxArray to mxGPUArray

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyFromMxArray(mxArray const * const mp)
```

Arguments

mp

    Pointer to an `mxArray` that contains either GPU or CPU data.

Returns

Pointer to an `mxGPUArray`.

Description

`mxGPUCopyFromMxArray` produces a new `mxGPUArray` object with the same characteristics as the input `mxArray`.

- If the input `mxArray` contains a `gpuArray`, the output is a new copy of the data on the GPU.
- If the input `mxArray` contains numeric or logical CPU data, the output is copied to the GPU.

Either way, this function always allocates memory on the GPU and allocates a new `mxGPUArray` object on the CPU. Use `mxGPUDestroyGPUArray` to delete the result when you are done with it.

See Also

`mxGPUCopyGPUArray` | `mxGPUDestroyGPUArray`

Introduced in R2013a

mxGPUCopyGPUArray (C)

Duplicate (deep copy) mxGPUArray object

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyGPUArray(mxGPUArray const * const mgp)
```

Arguments

mgp
    Pointer to an mxGPUArray.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyGPUArray produces a new array on the GPU and copies the data, and then returns a new mxGPUArray that refers to the copy. Use mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also

mxGPUCopyFromMxArray | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUCopyImag (C)

Copy imaginary part of mxGPUArray

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyImag(mxGPUArray const * const mgp)
```

Arguments

mgp

>   Pointer to an mxGPUArray. The target gpuArray must be full, not sparse.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyImag copies the imaginary part of GPU data, and returns a new mxGPUArray object that refers to the copy. The returned array is real, with element values equal to the imaginary values of the input, similar to how the MATLAB imag function behaves. If the input is real rather than complex, the function returns an array of zeros.

Use mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also
mxGPUCopyReal | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUCopyReal (C)

Copy real part of mxGPUArray

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyReal(mxGPUArray const * const mgp)
```

Arguments

mgp
> Pointer to an mxGPUArray. The target gpuArray must be full, not sparse.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyReal copies the real part of GPU data, and returns a new mxGPUArray object that refers to the copy. If the input is real rather than complex, the function returns a copy of the input.

Use mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also
mxGPUCopyImag | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUCreateComplexGPUArray (C)

Create complex GPU array from two real gpuArrays

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCreateComplexGPUArray(mxGPUArray const * const mgpR,
                                       mxGPUArray const * const mgpI)
```

Arguments

`mgpRmgpI`

> Pointers to `mxGPUArray` data containing real and imaginary coefficients. The target gpuArrays must be full, not sparse.

Returns

Pointer to an `mxGPUArray`.

Description

`mxGPUCreateComplexGPUArray` creates a new complex `mxGPUArray` from two real `mxGPUArray` objects. The function allocates memory on the GPU and copies the data. The inputs must both be real, and have matching sizes and classes. Use `mxGPUDestroyGPUArray` to delete the result when you are done with it.

See Also
`mxGPUDestroyGPUArray`

Introduced in R2013a

mxGPUCreateFromMxArray (C)

Create read-only mxGPUArray object from input mxArray

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray const * mxGPUCreateFromMxArray(mxArray const * const mp)
```

Arguments

mp

  Pointer to an mxArray that contains either GPU or CPU data.

Returns

Pointer to a read-only mxGPUArray object.

Description

mxGPUCreateFromMxArray produces a read-only mxGPUArray object from an mxArray.

- If the input mxArray contains a gpuArray, this function extracts a reference to the GPU data from an mxArray passed as an input to the function.
- If the input mxArray contains CPU data, the data is copied to the GPU, but the returned object is still read-only.

If you need a writable copy of the array, use mxGPUCopyFromMxArray instead.

This function allocates a new mxGPUArray object on the CPU. Use mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also
mxGPUCopyFromMxArray | mxGPUCreateGPUArray | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUCreateGPUArray (C)

Create mxGPUArray object, allocating memory on GPU

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCreateGPUArray(mwSize const ndims,
                               mwSize const * const dims,
                               mxClassID const cid,
                               mxComplexity const ccx,
                               mxGPUInitialize const init0)
```

Arguments

ndims

   mwSize type specifying the number of dimensions in the created mxGPUArray.

dims

   Pointer to an mwSize vector specifying the sizes of each dimension in the created mxGPUArray.

cid

   mxClassID type specifying the element class of the created mxGPUArray.

ccx

   mxComplexity type specifying the complexity of the created mxGPUArray.

init0

   mxGPUInitialize type specifying whether to initialize elements values to 0 in the created mxGPUArray.

   • A value of MX_GPU_INITIALIZE_VALUES specifies that elements are to be initialized to 0.

   • A value of MX_GPU_DO_NOT_INITIALIZE specifies that elements are not to be initialized.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCreateGPUArray creates a new mxGPUArray object with the specified size, type, and complexity. It also allocates the required memory on the GPU, and initializes the memory if requested.

This function allocates a new mxGPUArray object on the CPU. Use mxGPUDestroyGPUArray to delete the object when you are done with it.

See Also
mxGPUCreateFromMxArray | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUCreateMxArrayOnCPU (C)

Create mxArray for returning CPU data to MATLAB with data from GPU

C Syntax

```
#include "gpu/mxGPUArray.h"
mxArray* mxGPUCreateMxArrayOnCPU(mxGPUArray const * const mgp)
```

Arguments

mgp

   Pointer to an mxGPUArray.

Returns

Pointer to an mxArray object containing CPU data that is a copy of the GPU data.

Description

mxGPUCreateMxArrayOnCPU copies the GPU data from the specified mxGPUArray into an mxArray on the CPU for return to MATLAB. This is similar to the gather function. After calling this function, the input mxGPUArray object is no longer needed and you can delete it with mxGPUDestroyGPUArray.

See Also
mxGPUCreateMxArrayOnGPU | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUCreateMxArrayOnGPU (C)

Create mxArray for returning GPU data to MATLAB

C Syntax

```
#include "gpu/mxGPUArray.h"
mxArray* mxGPUCreateMxArrayOnGPU(mxGPUArray const * const mgp)
```

Arguments

mgp

>   Pointer to an mxGPUArray.

Returns

Pointer to an mxArray object containing GPU data.

Description

mxGPUCreateMxArrayOnGPU puts the mxGPUArray into an mxArray for return to MATLAB. The data remains on the GPU and the returned class in MATLAB is gpuArray. After this call, the mxGPUArray object is no longer needed and can be destroyed.

See Also
mxGPUCreateMxArrayOnCPU | mxGPUDestroyGPUArray

Introduced in R2013a

mxGPUDestroyGPUArray (C)

Delete mxGPUArray object

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUDestroyGPUArray(mxGPUArray const * const mgp)
```

Arguments

mgp

    Pointer to an mxGPUArray.

Description

mxGPUDestroyGPUArray deletes an mxGPUArray object on the CPU. Use this function to delete an mxGPUArray object you created with:

- mxGPUCreateGPUArray
- mxGPUCreateFromMxArray
- mxGPUCopyFromMxArray
- mxGPUCopyReal
- mxGPUCopyImag, or
- mxGPUCreateComplexGPUArray.

This function clears memory on the GPU, unless some other mxArray holds a reference to the same data. For example, if the mxGPUArray was extracted from an input mxArray, or wrapped in an mxArray for an output, then the data remains on the GPU.

See Also

mxGPUCopyFromMxArray | mxGPUCopyImag | mxGPUCopyReal | mxGPUCreateComplexGPUArray | mxGPUCreateFromMxArray | mxGPUCreateGPUArray

Introduced in R2013a

mxGPUGetClassID (C)

mxClassID associated with data on GPU

C Syntax

```
#include "gpu/mxGPUArray.h"
mxClassID mxGPUGetClassID(mxGPUArray const * const mgp)
```

Arguments

mgp
   Pointer to an mxGPUArray.

Returns

mxClassID type.

Description

mxGPUGetClassID returns an mxClassID type indicating the underlying class of the input data.

See Also
mxGPUGetComplexity

Introduced in R2013a

mxGPUGetComplexity (C)

Complexity of data on GPU

C Syntax

```
#include "gpu/mxGPUArray.h"
mxComplexity mxGPUGetComplexity(mxGPUArray const * const mgp)
```

Arguments

mgp
    Pointer to an mxGPUArray.

Returns

mxComplexity type.

Description

mxGPUGetComplexity returns an mxComplexity type indicating the complexity of the GPU data.

See Also
mxGPUGetClassID

Introduced in R2013a

mxGPUGetData (C)

Raw pointer to underlying data

C Syntax

```
#include "gpu/mxGPUArray.h"
void* mxGPUGetData(mxGPUArray const * const mgp)
```

Arguments

mgp

 Pointer to an `mxGPUArray` on the GPU. The target gpuArray must be full, not sparse.

Returns

Pointer to data.

Description

`mxGPUGetData` returns a raw pointer to the underlying data. Cast this pointer to the type of data that you want to use on the device. It is your responsibility to check that the data inside the array has the appropriate type, for which you can use `mxGPUGetClassID`.

See Also
`mxGPUGetClassID` | `mxGPUGetDataReadOnly`

Introduced in R2013a

mxGPUGetDataReadOnly (C)

Read-only raw pointer to underlying data

C Syntax

```
#include "gpu/mxGPUArray.h"
void const* mxGPUGetDataReadOnly(mxGPUArray const * const mgp)
```

Arguments

mgp

    Pointer to an `mxGPUArray` on the GPU. The target gpuArray must be full, not sparse.

Returns

Read-only pointer to data.

Description

`mxGPUGetDataReadOnly` returns a read-only raw pointer to the underlying data. Cast it to the type of data that you want to use on the device. It is your responsibility to check that the data inside the array has the appropriate type, for which you can use `mxGPUGetClassID`.

See Also
`mxGPUGetClassID` | `mxGPUGetData`

Introduced in R2013a

mxGPUGetDimensions (C)

mxGPUArray dimensions

C Syntax

```
#include "gpu/mxGPUArray.h"
mwSize const * mxGPUGetDimensions(mxGPUArray const * const mgp)
```

Arguments

mgp

    Pointer to an mxGPUArray.

Returns

Pointer to a read-only array of mwSize type.

Description

mxGPUGetDimensions returns a pointer to an array of mwSize indicating the dimensions of the input argument. Use mxFree to delete the output.

See Also

mxGPUGetComplexity | mxGPUGetNumberOfDimensions

Introduced in R2013a

mxGPUGetNumberOfDimensions (C)

Size of dimension array for mxGPUArray

C Syntax

```
#include "gpu/mxGPUArray.h"
mwSize mxGPUGetNumberOfDimensions(mxGPUArray const * const mgp)
```

Arguments

mgp
    Pointer to an mxGPUArray.

Returns

mwSize type.

Description

mxGPUGetNumberOfDimensions returns the size of the dimension array for the mxGPUArray input argument, indicating the number of its dimensions.

See Also
mxGPUGetComplexity | mxGPUGetDimensions

Introduced in R2013a

mxGPUGetNumberOfElements (C)

Number of elements on GPU for array

C Syntax

```
#include "gpu/mxGPUArray.h"
mwSize mxGPUGetNumberOfElements(mxGPUArray const * const mgp)
```

Arguments

mgp
   Pointer to an mxGPUArray.

Returns

mwSize type.

Description

mxGPUGetNumberOfElements returns the total number of elements on the GPU for this array.

See Also
mxGPUGetComplexity | mxGPUGetDimensions | mxGPUGetNumberOfDimensions

Introduced in R2013a

mxGPUIsSame (C)

Determine if two mxGPUArrays refer to same GPU data

C Syntax

```
#include "gpu/mxGPUArray.h"
int mxGPUIsSame(mxGPUArray const * const mgp1,
                mxGPUArray const * const mgp2)
```

Arguments

mgp1mgp2
    Pointers to `mxGPUArray`.

Returns

`int` type.

Description

`mxGPUIsSame` returns an integer indicating if two `mxGPUArray` pointers refer to the same GPU data:

- 1 (true) indicates that the inputs refer to the same data.
- 0 (false) indicates that the inputs do not refer to the same data.

See Also
mxGPUIsValidGPUData

Introduced in R2013a

mxGPUIsSparse (C)

Determine if mxGPUArray contains sparse GPU data

C Syntax

```
#include "gpu/mxGPUArray.h"
int mxGPUIsSparse(mxGPUArray const * mp);
```

Arguments

mp
>   Pointer to an mxGPUArray to be queried for sparse data.

Returns

Integer indicating true result:

- 1 indicates the input is a sparse gpuArray.
- 0 indicates the input is not a sparse gpuArray.

See Also
mxGPUIsValidGPUData | mxIsGPUArray

Introduced in R2015a

mxGPUIsValidGPUData (C)

Determine if mxArray is pointer to valid GPU data

C Syntax

```
#include "gpu/mxGPUArray.h"
int mxGPUIsValidGPUData(mxArray const * const mp)
```

Arguments

mgp
   Pointer to an mxArray.

Returns

int type.

Description

mxGPUIsValidGPUData indicates if the mxArray is a pointer to valid GPU data

If the GPU device is reinitialized in MATLAB with gpuDevice, all data on the device becomes invalid, but the CPU data structures that refer to the GPU data still exist. This function checks whether the mxArray is a container of valid GPU data, and returns one of the following values:

- 0 (false) for CPU data or for invalid GPU data.
- 1 (true) for valid GPU data.

See Also
mxIsGPUArray

Introduced in R2013a

mxGPUSetDimensions (C)

Modify number of dimensions and size of each dimension

C Syntax

```
#include "gpu/mxGPUArray.h"
void mxGPUSetDimensions(mxGPUArray * const mgp, mwSize const * const dims, mwSize const ndims);
```

Arguments

mgp

Pointer to an mxGPUArray

dims

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxGPUArray.

The dims array must not increase the overall size of the mxGPUArray. This array must contain at least ndims elements.

ndims

Number of dimensions.

Description

Call mxGPUSetDimensions to reshape an existing mxGPUArray. mxGPUSetDimensions does not reallocate memory.

See Also

mxGPUGetDimensions (C)

Introduced in R2018b

mxInitGPU (C)

Initialize MATLAB GPU library on currently selected device

C Syntax

```
#include "gpu/mxGPUArray.h"
 int mxInitGPU()
```

Returns

`int` type with one of the following values:

- `MX_GPU_SUCCESS` if the MATLAB GPU library is successfully initialized.
- `MX_GPU_FAILURE` if not successfully initialized.

Description

Before using any CUDA code in your MEX file, initialize the MATLAB GPU library if you intend to use any `mxGPUArray` functionality in MEX or any GPU calls in MATLAB. There are many ways to initialize the MATLAB GPU API, including:

- Call `mxInitGPU` at the beginning of your MEX file before any CUDA code.
- Call `gpuDevice(deviceIndex)` in MATLAB before running any MEX code.
- Create a `gpuArray` in MATLAB before running any MEX code.

You should call `mxInitGPU` at the beginning of your MEX file, unless you have an alternate way of guaranteeing that the MATLAB GPU library is initialized at the start of your MEX file.

If the library is initialized, this function returns without doing any work. If the library is not initialized, the function initializes the default device. Note: At present, a MATLAB MEX file can work with only one GPU device at a time.

See Also
`gpuArray` | `gpuDevice`

Introduced in R2013a

mxIsGPUArray (C)

Determine if mxArray contains GPU data

C Syntax

```
#include "gpu/mxGPUArray.h"
int mxIsGPUArray(mxArray const * const mp);
```

Arguments

mp

Pointer to an `mxArray` that might contain gpuArray data.

Returns

Integer indicating true result:

- 1 indicates the input is a gpuArray.
- 0 indicates the input is not a gpuArray.

See Also
`mxGPUIsSparse` | `mxGPUIsValidGPUData`

Introduced in R2013a

NaN

Create array of Not-a-Numbers for GPU or distributed arrays

Syntax

```
A = NaN(n,arraytype)
A = NaN(size1,...,sizeN,arraytype)
A = NaN(size,arraytype)

A = NaN(n,datatype,arraytype)
A = NaN(size1,...,sizeN,datatype,arraytype)
A = NaN(size,datatype,arraytype)

A = NaN(n,'like',P)
A = NaN(size1,...,sizeN,'like',P)
A = NaN(size,'like',P)

A = NaN(n,datatype,'like',P)
A = NaN(size1,...,sizeN,datatype'like',P)
A = NaN(size,datatype,'like',P)

C = NaN(n,codist)
C = NaN(size1,...,sizeN,codist)
C = NaN(size,codist)

C = NaN(n,datatype,codist)
C = NaN(size1,...,sizeN,datatype,codist)
C = NaN(size,datatype,codist)

C = NaN( ___ ,codist,'noCommunication')
C = NaN( ___ ,codist,'like',P)
```

Description

A = NaN(n,arraytype) returns an *n*-by-*n* matrix with underlying class of double, NaN values in all elements, and the type specified by arraytype.

A = NaN(size1,...,sizeN,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of double, NaN values in all elements, and the type specified by arraytype.

A = NaN(size,arraytype) returns an array which size is defined by the size vector *size* with underlying class of double, NaN values in all elements, and the type specified by arraytype.

A = NaN(n,datatype,arraytype) returns an *n*-by-*n* matrix with underlying class of datatype, NaN values in all elements, and the type specified by arraytype.

A = NaN(size1,...,sizeN,datatype,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of datatype, NaN values in all elements, and the type specified by arraytype.

`A = NaN(size,datatype,arraytype)` returns an array which size is defined by the size vector *size* with underlying class of `datatype`, NaN values in all elements, and the type specified by `arraytype`.

`A = NaN(n,'like',P)` returns an *n*-by-*n* array of NaN values with the same underlying class and type as the prototype array, `P`.

`A = NaN(size1,...,sizeN,'like',P)` returns a *size1*-by-...-by-*sizeN* array of NaN values with the same underlying class and type as the prototype array, `P`.

`A = NaN(size,'like',P)` returns an array of NaN values which size is defined by the size vector *size* with the same underlying class and type as the prototype array, `P`.

`A = NaN(n,datatype,'like',P)` returns an *n*-by-*n* array with underlying class of `datatype`, NaN values in all elements, and the same type as the prototype array, `P`.

`A = NaN(size1,...,sizeN,datatype'like',P)` returns a *size1*-by-...-by-*sizeN* array with underlying class of `datatype`, NaN values in all elements, and the same type as the prototype array, `P`.

`A = NaN(size,datatype,'like',P)` returns an array which size is defined by the size vector *size* with underlying class of `datatype`, NaN values in all elements, and the same type as the prototype array, `P`.

`C = NaN(n,codist)` returns an *n*-by-*n* codistributed array with underlying class of double and NaN values in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = NaN(size1,...,sizeN,codist)` returns a *size1*-by-...-by-*sizeN* codistributed array with underlying class of double and NaN values in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = NaN(size,codist)` returns a codistributed array which size is defined by the size vector *size* with underlying class of double and NaN values in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = NaN(n,datatype,codist)` returns an *n*-by-*n* codistributed array with underlying class of `datatype` and NaN values in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = NaN(size1,...,sizeN,datatype,codist)` returns a *size1*-by-...-by-*sizeN* codistributed array with underlying class of `datatype` and NaN values in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = NaN(size,datatype,codist)` returns a codistributed array which size is defined by the size vector *size* with underlying class of `datatype` and NaN values in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

C = NaN(___ ,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = NaN(___ ,codist,'like',P) creates a codistributed array of NaN values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

Examples

Create Distributed NaN Matrix

Create a 1000-by-1000 distributed array of NaN values with underlying class double.

```
D = NaN(1000,'distributed');
```

Create Codistributed NaN Matrix

Create a 1000-by-1000 codistributed double matrix of NaN values, distributed by its second dimension (columns).

```
spmd(4)
    C = NaN(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed uint16 matrix of NaN values, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = NaN(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray NaN Matrix

Create a 1000-by-1000 gpuArray of NaNvalues, with underlying class double:

```
G = NaN(1000,'double','gpuArray');
```

Input Arguments

n — Size of array
integer value

Size of the generated array, specified as an integer value.

size1,...,sizeN — Size of each dimension (as separate arguments)
integer values

Size of each dimension of the generated array, specified as separate arguments of two or more integer values.

`size` — Size of each dimension (as a row vector)
vector

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

`arraytype` — Array type
`'distributed'` | `'codistributed'` | `'gpuArray'`

Type of the generated array, specified as either `'distributed'`, `'codistributed'`, or `'gpuArray'`.

`datatype` — Array data type
`'double'` (default) | `'single'`

Underlying class of the array, that is the data type of its elements, specified as either `'double'` or `'single'`.

`codist` — Codistributor
codistributor object

Distribution scheme for creating the codistributed array, specified as a codistributor object. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. To use the default distribution scheme, you can specify a codistributor constructor without arguments.

`P` — Prototype array
array

Prototype of array to create, specified as an array.

Output Arguments

`A` — Array
distributed array | codistributed array | gpuArray

Array of NaN values, returned as either a distributed array, a codistributed array, or a gpuArray.

`C` — Codistributed array
codistributed array

Array of NaN values, returned as a codistributed array.

See Also

`Inf` | `eye` | `false` | `ones` | `true` | `zeros`

Introduced in R2006b

numlabs

Total number of workers operating in parallel on current job

Syntax

```
n = numlabs
```

Description

`n = numlabs` returns the total number of workers currently operating on the current job. This value is the maximum value that can be used with `labSend` and `labReceive`.

Tips

In an `spmd` block, `numlabs` on each worker returns the parallel pool size.

However, inside a `parfor`-loop, `numlabs` always returns a value of 1.

See Also
`labSendReceive` | `labindex`

Introduced before R2006a

ones

Create array of all ones for GPU or distributed arrays

Syntax

```
N = ones(n,arraytype)
N = ones(size1,...,sizeN,arraytype)
N = ones(size,arraytype)

N = ones(n,datatype,arraytype)
N = ones(size1,...,sizeN,datatype,arraytype)
N = ones(size,datatype,arraytype)

N = ones(n,'like',P)
N = ones(size1,...,sizeN,'like',P)
N = ones(size,'like',P)

N = ones(n,datatype,'like',P)
N = ones(size1,...,sizeN,datatype'like',P)
N = ones(size,datatype,'like',P)

C = ones(n,codist)
C = ones(size1,...,sizeN,codist)
C = ones(size,codist)

C = ones(n,datatype,codist)
C = ones(size1,...,sizeN,datatype,codist)
C = ones(size,datatype,codist)

C = ones( ___ ,codist,'noCommunication')
C = ones( ___ ,codist,'like',P)
```

Description

N = ones(n,arraytype) returns an *n*-by-*n* matrix with underlying class of double, ones in all elements, and the type specified by arraytype.

N = ones(size1,...,sizeN,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of double, ones in all elements, and the type specified by arraytype.

N = ones(size,arraytype) returns an array which size is defined by the size vector *size* with underlying class of double, ones in all elements, and the type specified by arraytype.

N = ones(n,datatype,arraytype) returns an *n*-by-*n* matrix with underlying class of datatype, ones in all elements, and the type specified by arraytype.

N = ones(size1,...,sizeN,datatype,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of datatype, ones in all elements, and the type specified by arraytype.

N = ones(size,datatype,arraytype) returns an array which size is defined by the size vector *size* with underlying class of datatype, ones in all elements, and the type specified by arraytype.

N = ones(n,'like',P) returns an *n*-by-*n* array of ones with the same underlying class and type as the prototype array, P.

N = ones(size1,...,sizeN,'like',P) returns a *size1*-by-...-by-*sizeN* array of ones with the same underlying class and type as the prototype array, P.

N = ones(size,'like',P) returns an array of ones which size is defined by the size vector *size* with the same underlying class and type as the prototype array, P.

N = ones(n,datatype,'like',P) returns an *n*-by-*n* array with underlying class of datatype, ones in all elements, and the same type as the prototype array, P.

N = ones(size1,...,sizeN,datatype'like',P) returns a *size1*-by-...-by-*sizeN* array with underlying class of datatype, ones in all elements, and the same type as the prototype array, P.

N = ones(size,datatype,'like',P) returns an array which size is defined by the size vector *size* with underlying class of datatype, ones in all elements, and the same type as the prototype array, P.

C = ones(n,codist) returns an *n*-by-*n* codistributed array with underlying class of double and ones in all elements. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = ones(size1,...,sizeN,codist) returns a *size1*-by-...-by-*sizeN* codistributed array with underlying class of double and ones in all elements. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = ones(size,codist) returns a codistributed array which size is defined by the size vector *size* with underlying class of double and ones in all elements. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = ones(n,datatype,codist) returns an *n*-by-*n* codistributed array with underlying class of datatype and ones in all elements. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = ones(size1,...,sizeN,datatype,codist) returns a *size1*-by-...-by-*sizeN* codistributed array with underlying class of datatype and ones in all elements. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = ones(size,datatype,codist) returns a codistributed array which size is defined by the size vector *size* with underlying class of datatype and ones in all elements. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = ones(___ ,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = ones(___ ,codist,'like',P) creates a codistributed array of ones with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

Examples

Create Distributed Ones Matrix

Create a 1000-by-1000 distributed array of ones with underlying class double.

```
D = ones(1000,'distributed');
```

Create Codistributed Ones Matrix

Create a 1000-by-1000 codistributed double matrix of ones, distributed by its second dimension (columns).

```
spmd(4)
    C = ones(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed uint16 matrix of ones, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = ones(1000,1000,'uint16',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray Ones Matrix

Create a 1000-by-1000 gpuArray of ones, with underlying class uint32:

```
G = ones(1000,'uint32','gpuArray');
```

Input Arguments

n — Size of array
integer value

Size of the generated array, specified as an integer value.

size1,...,sizeN — Size of each dimension (as separate arguments)
integer values

Size of each dimension of the generated array, specified as separate arguments of two or more integer values.

size — Size of each dimension (as a row vector)
vector

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

arraytype — Array type
'distributed' | 'codistributed' | 'gpuArray'

Type of the generated array, specified as either 'distributed', 'codistributed', or 'gpuArray'.

datatype — Array data type
'double' (default) | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64'

Underlying class of the array, that is the data type of its elements, specified as one of these options:

- 'double'
- 'single'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'

codist — Codistributor
codistributor object

Distribution scheme for creating the codistributed array, specified as a codistributor object. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments.

P — Prototype array
array

Prototype of array to create, specified as an array.

Output Arguments

N — Array
distributed array | codistributed array | gpuArray

Array of ones, returned as either a distributed array, a codistributed array, or a gpuArray.

C — Codistributed array
codistributed array

Array of ones, returned as a codistributed array.

See Also

Inf | NaN | eye | false | true | zeros

Introduced in R2006b

pagefun

Apply function to each page of distributed array or gpuArray

Syntax

```
A = pagefun(FUN,B)
A = pagefun(FUN,B1,...,Bn)
[A1,...,Am] = pagefun(FUN, ___ )
```

Description

`A = pagefun(FUN,B)` applies the function specified by `FUN` to each page of the distributed array or gpuArray `B`. The result `A` contains each page of results such that `A(:,:,I,J,...) = FUN(B(:,:,I,J,...))`. `A` is a distributed array or a gpuArray, depending on the array type of `B`. `FUN` is a handle to a function that takes a two-dimensional input argument.

`A = pagefun(FUN,B1,...,Bn)` evaluates `FUN` using pages of the arrays `B1,...,Bn` as input arguments with scalar expansion enabled. Any of the input page dimensions that are scalar are virtually replicated to match the size of the other arrays in that dimension so that `A(:,:,I,J,...) = FUN(B1(:,:,I,J,...),...,Bn(:,:,I,J,...))`. The input pages `B(:,:,I,J,...),...,Bn(:,:,I,J,...)`, must satisfy all of the input requirements of `FUN`.

If you plan to make several calls to `pagefun`, it is more efficient to first convert that array to a distributed array or gpuArray.

`[A1,...,Am] = pagefun(FUN, ___ )` returns multiple output arrays `A1,...,Am` when the function `FUN` returns m output values. `pagefun` calls `FUN` each time with as many outputs as there are in the call to `pagefun`, that is, m times. `FUN` can return output arguments having different data types, but the data type of each output must be the same each time `FUN` is called.

Examples

Apply the `mtimes` Function to Each Page of the B and C GPU Arrays Using `pagefun`

This example shows how to use `pagefun` to apply the `mtimes` function to each page of two GPU arrays, `B` and `C`.

Create the two GPU arrays, `B` and `C`.

```
M = 3;          % output number of rows
K = 6;          % matrix multiply inner dimension
N = 2;          % output number of columns
P1 = 10;        % size of first page dimension
P2 = 17;        % size of second page dimension
P3 = 4;         % size of third page dimension
P4 = 12;        % size of fourth page dimension
A = rand(M,K,P1,1,P3,'gpuArray');
B = rand(K,N,1,P2,P3,P4,'gpuArray');
```

Call `pagefun` to apply the `mtimes` function to each page of these two arrays.

```
C = pagefun(@mtimes,A,B);
s = size(C)     % M-by-N-by-P1-by-P2-by-P3-by-P4

s =
    3     2    10    17     4    12
```

This code shows another similar use-case of the `pagefun` function.

```
M = 300;        % output number of rows
K = 500;        % matrix multiply inner dimension
N = 1000;       % output number of columns
P = 200;        % number of pages
A = rand(M,K,'gpuArray');
B = rand(K,N,P,'gpuArray');
C = pagefun(@mtimes,A,B);
s = size(C)     % returns M-by-N-by-P

s =
     300        1000         200
```

Input Arguments

FUN — Function
function handle

Function applied to each page of the inputs, specified as a function handle. For each output argument, FUN must return values of the same class each time it is called.

The supported values for FUN include:

- Most element-wise distributed array and gpuArray functions
- @ctranspose
- @fliplr
- @flipud
- @inv
- @mldivide
- @mrdivide
- @mtimes
- @rot90
- @transpose
- @tril
- @triu

If the inputs are distributed arrays, the supported values for FUN also include:

- @lu
- @qr
- @norm

B — Input array
distributed array | gpuArray

Input array, specified as a distributed array or gpuArray.

B1,...,Bn — Input arrays
distributed array | gpuArray | array

Input arrays, specified as distributed arrays, gpuArrays, or arrays. At least one of the inputs B1,...,Bn, must be a distributed array or gpuArray. Using both distributed array and gpuArray as inputs is not supported. Each array that is stored in CPU memory is converted to a distributed array or gpuArray before the function is evaluated. If you plan to make several calls to `pagefun` with the same array, it is more efficient to first convert that array to a distributed array or a gpuArray.

Output Arguments

A — Output array
distributed array | gpuArray

Output array, returned as a distributed array or gpuArray.

See Also
`arrayfun` | `bsxfun` | `gather` | `gpuArray`

Introduced in R2013b

parallel.cluster.generic.awsbatch.deleteBatchJob

Package: `parallel.cluster.generic.awsbatch`

Terminate job in AWS Batch

Syntax

`parallel.cluster.generic.awsbatch.deleteBatchJob(jobID)`

Description

`parallel.cluster.generic.awsbatch.deleteBatchJob(jobID)` terminates the AWS Batch job with the ID `jobID`.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

jobID — ID of the AWS Batch job
character vector | string scalar

ID of the AWS Batch job to terminate.

Data Types: `char` | `string`

See Also
`parallel.cluster.generic.awsbatch.deleteJobFilesFromS3` |
`parallel.cluster.generic.awsbatch.getBatchJobInfo` |
`parallel.cluster.generic.awsbatch.submitBatchJob`

Introduced in R2019b

parallel.cluster.generic.awsbatch.deleteJobFilesFromS3

Package: `parallel.cluster.generic.awsbatch`

Delete job files from Amazon S3

Syntax

`parallel.cluster.generic.awsbatch.deleteJobFilesFromS3(job,s3Bucket,s3Prefix)`

Description

`parallel.cluster.generic.awsbatch.deleteJobFilesFromS3(job,s3Bucket,s3Prefix)` deletes the files for `job`, which are located in the folder `s3://s3Bucket/s3Prefix`, from Amazon S3™.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

job — MATLAB job
`parallel.job.CJSIndependentJob` object

MATLAB job, specified as a `parallel.job.CJSIndependentJob` object.

Data Types: `parallel.job.CJSIndependentJob`

s3Bucket — S3 bucket
character vector | string scalar

S3 bucket where the job files are stored, specified as a character vector or string scalar.

Data Types: `char` | `string`

s3Prefix — Prefix of S3 location
character vector | string scalar

Prefix of the S3 location that contains the job files, specified as a character vector or a string scalar.

Data Types: `char` | `string`

See Also
`parallel.cluster.generic.awsbatch.downloadJobFilesFromS3` |
`parallel.cluster.generic.awsbatch.uploadJobFilesToS3`

Introduced in R2019b

parallel.cluster.generic.awsbatch.downloadJobFilesFromS3

Package: `parallel.cluster.generic.awsbatch`

Download job output files from Amazon S3

Syntax

`parallel.cluster.generic.awsbatch.downloadJobFilesFromS3(job,s3Bucket, s3Prefix)`

Description

`parallel.cluster.generic.awsbatch.downloadJobFilesFromS3(job,s3Bucket, s3Prefix)` downloads the output files for job `job` from the Amazon S3 bucket `s3Bucket` and saves them to the `JobStorageLocation` of the cluster. This function expects output files stored in zip files under the prefix `s3Prefix/stageOut` in the Amazon S3 bucket `s3Bucket`.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

job — MATLAB job
`parallel.job.CJSIndependentJob` object

MATLAB job, specified as a `parallel.job.CJSIndependentJob` object.

Data Types: `parallel.job.CJSIndependentJob`

s3Bucket — S3 bucket
character vector | string scalar

S3 bucket to download job files from, specified as a character vector or string scalar.

Data Types: `char` | `string`

s3Prefix — Prefix of the S3 location
character vector | string scalar

Prefix of the S3 location in the S3 bucket `s3Bucket` that contains the output files for `job`, specified as a character vector or string array.

Data Types: `char` | `string`

See Also
`parallel.cluster.generic.awsbatch.deleteJobFilesFromS3` | `parallel.cluster.generic.awsbatch.uploadJobFilesToS3`

Introduced in R2019b

parallel.cluster.generic.awsbatch.downloadJobLog Files

Package: `parallel.cluster.generic.awsbatch`

Download AWS Batch job log files

Syntax

`parallel.cluster.generic.awsbatch.downloadJobLogFiles(job,taskIDs,logStreams)`

Description

`parallel.cluster.generic.awsbatch.downloadJobLogFiles(job,taskIDs,logStreams)` downloads log files for the tasks in `job` with the IDs `taskIDs` from the log streams `logStreams` in AWS CloudWatch Logs, and saves them to the `JobStorageLocation` of the cluster.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

job — MATLAB AWS Batch job
`parallel.job.CJSIndependentJob` object

MATLAB AWS Batch job, specified as a `parallel.job.CJSIndependentJob` object.

Data Types: `parallel.job.CJSIndependentJob`

taskIDs — ID of the tasks
numeric vector

ID of the tasks to download logs for, specified as a numeric vector. Each task ID must have a corresponding log stream in `logStreams`.

Data Types: `double`

logStreams — Amazon CloudWatch log streams
cell array of character vectors | string array

Amazon CloudWatch log streams that contain the log information for each task, specified as a cell array of character vectors or string array. Each log stream must have a corresponding task ID in `taskIDs`. You can get this information from the output of `parallel.cluster.generic.awsbatch.getBatchJobInfo`.

For more information on log streams, see the Amazon CloudWatch documentation.

Data Types: `string` | `cell`

See Also
parallel.cluster.generic.awsbatch.getBatchJobInfo |
parallel.cluster.generic.awsbatch.submitBatchJob

Introduced in R2019b

parallel.cluster.generic.awsbatch.getBatchJobInfo

Package: `parallel.cluster.generic.awsbatch`

Get AWS Batch job information

Syntax

```
info = parallel.cluster.generic.awsbatch.getBatchJobInfo(job)
```

Description

`info = parallel.cluster.generic.awsbatch.getBatchJobInfo(job)` returns a table with information on each task in the MATLAB AWS Batch job.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

job — MATLAB AWS Batch job
`parallel.job.CJSIndependentJob` object

MATLAB AWS Batch job, specified as a `parallel.job.CJSIndependentJob` object.

Data Types: `parallel.job.CJSIndependentJob`

Output Arguments

info — Task information
`table`

Task information, returned as a table with the following variables.

Variable	Data Type	Value
TaskID	string	The ID of the task in `job`. `getBatchJobInfo` collects information only on tasks with a `schedulerID`.
Status	string	The status for the corresponding AWS Batch job of the task, as reported by AWS Batch. If AWS Batch does not provide a state for the job, `Status` is set to `"UNKNOWN"`. Note that `"UNKNOWN"` is not a state defined by AWS.

11-245

Variable	Data Type	Value
LogStreamName	string	The log stream name in Amazon CloudWatch Logs for the corresponding AWS Batch job of the task. If AWS Batch does not provide a log stream name, then LogStreamName is "".

Note that AWS only returns information for AWS Batch jobs in the SUCCEEDED or FAILED state over the last 24 hours. After 24 hours elapses, Status is "UNKNOWN" and LogStreamName is "".

Data Types: table

See Also

parallel.cluster.generic.awsbatch.submitBatchJob

Introduced in R2019b

parallel.cluster.generic.awsbatch.submitBatchJob

Package: `parallel.cluster.generic.awsbatch`

Submit job to AWS Batch

Syntax

```
schedulerID = parallel.cluster.generic.awsbatch.submitBatchJob(arraySize,
jobName,jobQueue,jobDefinition,command,environmentVariableNames,
environmentVariableValues)
```

Description

`schedulerID = parallel.cluster.generic.awsbatch.submitBatchJob(arraySize, jobName,jobQueue,jobDefinition,command,environmentVariableNames, environmentVariableValues)` submits a job of size `arraySize` to the AWS Batch job queue `jobQueue`. The job has the name `jobName`, job definition `jobDefinition`. The container that runs the AWS Batch job receives and processes the command `command`. The job runs with the environment variables `environmentVariableNames` and values `environmentVariableValues`. This function returns an AWS Batch job ID.

For information about AWS Batch job queues, job definitions, and the command passed to the container that runs the AWS Batch job, see the AWS Batch documentation.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

`arraySize` — Size of job
positive integer

Size of the job, specified as a positive integer. If `arraySize` is greater than `1`, then `submitBatchJob` submits an array job. Otherwise, `submitBatchJob` submits a nonarray job.

`jobName` — Job name
character vector | string scalar

Job name for the AWS Batch job, specified as a character vector or string scalar. For more information, see the AWS Batch documentation.

Data Types: `char` | `string`

`jobQueue` — AWS Batch job queue
character vector | string scalar

AWS Batch job queue to submit the AWS Batch job to, specified as a character vector or string scalar. For more information, see the AWS Batch documentation.

Data Types: `char` | `string`

jobDefinition — AWS Batch job definition
character vector | string scalar

AWS Batch job definition for the AWS Batch job, specified as a character vector or string scalar. For more information, see the AWS Batch documentation.

Data Types: `char` | `string`

command — Command to pass
character vector | string scalar

Command to pass to the container that runs the AWS Batch job, specified as a character vector or string scalar. For more information, see the AWS Batch documentation.

Data Types: `char` | `string`

environmentVariableNames — Names of environment variables
cell array of character vectors | string array

Names of the environment variables to create on the AWS Batch job, specified as a cell array of character vectors or string array. Each variable must have a corresponding value in `environmentVariableValues`.

Data Types: `cell` | `string`

environmentVariableValues — Values of environment variables
cell array of character vectors | string array

Values of the environment variables to create on the AWS Batch job, specified as a cell array of character vectors or string array. Each value must have a corresponding variable in `environmentVariableValues`.

Data Types: `cell` | `string`

Output Arguments

schedulerID — Scheduler ID
string scalar

Scheduler ID of the AWS Batch job, returned as a string scalar.

Data Types: `char`

See Also
`parallel.cluster.generic.awsbatch.deleteBatchJob` |
`parallel.cluster.generic.awsbatch.downloadJobFilesFromS3` |
`parallel.cluster.generic.awsbatch.downloadJobLogFiles` |
`parallel.cluster.generic.awsbatch.getBatchJobInfo` |
`parallel.cluster.generic.awsbatch.uploadJobFilesToS3`

Introduced in R2019b

parallel.cluster.generic.awsbatch.uploadJobFilesToS3

Package: parallel.cluster.generic.awsbatch

Upload job input files to Amazon S3

Syntax

s3Prefix = parallel.cluster.generic.awsbatch.uploadJobFilesToS3(job,s3Bucket)

Description

s3Prefix = parallel.cluster.generic.awsbatch.uploadJobFilesToS3(job,s3Bucket)
uploads the input files for job to the Amazon S3 bucket s3Bucket under the prefix *s3Prefix*/
stageIn/, where *s3Prefix* is a randomly generated string.

Note This function requires the Parallel Computing Toolbox plugin for MATLAB Parallel Server with
AWS Batch. For an example of how to use the function, see the plugin scripts.

Input Arguments

job — MATLAB job
parallel.job.CJSIndependentJob object

MATLAB job, specified as a parallel.job.CJSIndependentJob object.

Data Types: parallel.job.CJSIndependentJob

s3Bucket — S3 bucket
character vector | string scalar

S3 bucket to upload job input files to, specified as a character vector or string scalar.

Data Types: char | string

Output Arguments

s3Prefix — Prefix of S3 location
string scalar

Prefix of the S3 location in the S3 bucket s3Bucket to which uploadJobFilesToS3 uploads files.
s3Prefix is a randomly generated string.

Data Types: char | string

See Also

parallel.cluster.generic.awsbatch.deleteJobFilesFromS3 |
parallel.cluster.generic.awsbatch.downloadJobFilesFromS3 |
parallel.cluster.generic.awsbatch.submitBatchJob

Introduced in R2019b

parallel.cluster.Hadoop

Create Hadoop cluster object

Syntax

```
hadoopCluster = parallel.cluster.Hadoop
hadoopCluster = parallel.cluster.Hadoop(Name,Value)
```

Description

`hadoopCluster = parallel.cluster.Hadoop` creates a parallel.cluster.Hadoop object representing the Hadoop cluster.

You use the resulting object as input to the `mapreduce` and `mapreducer` functions, for specifying the Hadoop cluster as the parallel execution environment for tall arrays and `mapreduce`.

`hadoopCluster = parallel.cluster.Hadoop(Name,Value)` uses the specified names and values to set properties on the created parallel.cluster.Hadoop object.

Examples

Set Hadoop Cluster as Execution Environment for mapreduce and mapreducer

This example shows how to create and use a parallel.cluster.Hadoop object to set a Hadoop cluster as the mapreduce parallel execution environment.

```
hadoopCluster = parallel.cluster.Hadoop('HadoopInstallFolder','/host/hadoop-install');
mr = mapreducer(hadoopCluster);
```

Set Hadoop Cluster as Execution Environment for tall arrays

This example shows how to create and use a parallel.cluster.Hadoop object to set a Hadoop cluster as the tall array parallel execution environment.

```
hadoopCluster = parallel.cluster.Hadoop(...
    'HadoopInstallFolder','/host/hadoop-install', ...
    'SparkInstallFolder','/host/spark-install');
mr = mapreducer(hadoopCluster);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'HadoopInstallFolder','/share/hadoop/a1.2.1'`

11-251

ClusterMatlabRoot — Path to MATLAB for workers
character vector

Path to MATLAB for workers, specified as the comma-separated pair consisting of `'ClusterMatlabRoot'` and a character vector. This points to the installation of MATLAB Parallel Server for the workers, whether local to each machine or on a network share.

HadoopConfigurationFile — Path to Hadoop application configuration file
character vector

Path to Hadoop application configuration file, specified as the comma-separated pair consisting of `'HadoopConfigurationFile'` and a character vector.

HadoopInstallFolder — Path to Hadoop installation on the local machine
character vector

Path to Hadoop installation on the local machine, specified as the comma-separated pair consisting of `'HadoopInstallFolder'` and a character vector. If this property is not set, the default is the value specified by the environment variable HADOOP_PREFIX, or if that is not set, then HADOOP_HOME.

SparkInstallFolder — Path to Spark enabled Hadoop installation on worker machines
character vector

Path to Spark enabled Hadoop installation on worker machines, specified as the comma-separated pair consisting of `'SparkInstallFolder'` and a character vector. If this property is not set, the default is the value specified by the environment variable SPARK_PREFIX, or if that is not set, then SPARK_HOME.

Output Arguments

hadoopCluster — Hadoop cluster
parallel.cluster.Hadoop object

Hadoop cluster, returned as a parallel.cluster.Hadoop object.

See Also
mapreduce | mapreducer

Topics
"Use Tall Arrays on a Spark Enabled Hadoop Cluster" on page 5-52
"Run mapreduce on a Hadoop Cluster" on page 5-58
"Read and Analyze Hadoop Sequence File"

Introduced in R2014b

parallel.clusterProfiles

Names of all available cluster profiles

Syntax

```
ALLPROFILES = parallel.clusterProfiles
[ALLPROFILES, DEFAULTPROFILE] = parallel.clusterProfiles
```

Description

`ALLPROFILES = parallel.clusterProfiles` returns a cell array containing the names of all available profiles.

`[ALLPROFILES, DEFAULTPROFILE] = parallel.clusterProfiles` returns a cell array containing the names of all available profiles, and separately the name of the default profile.

The cell array `ALLPROFILES` always contains a profile called `local` for the local cluster, and always contains the default profile. If the default profile has been deleted, or if it has never been set, `parallel.clusterProfiles` returns `local` as the default profile.

You can create and change profiles using the `saveProfile` or `saveAsProfile` methods on a cluster object. Also, you can create, delete, and change profiles through the Cluster Profile Manager.

Examples

Display the names of all the available profiles and set the first in the list to be the default profile.

```
allNames = parallel.clusterProfiles()
parallel.defaultClusterProfile(allNames{1});
```

Display the names of all the available profiles and get the cluster identified by the last profile name in the list.

```
allNames = parallel.clusterProfiles()
lastCluster = parcluster(allNames{end});
```

See Also
parallel.defaultClusterProfile | parallel.exportProfile | parallel.importProfile

Introduced in R2012a

11-253

parallel.defaultClusterProfile

Examine or set default cluster profile

Syntax

```
p = parallel.defaultClusterProfile
oldprofile = parallel.defaultClusterProfile(newprofile)
```

Description

`p = parallel.defaultClusterProfile` returns the name of the default cluster profile.

`oldprofile = parallel.defaultClusterProfile(newprofile)` sets the default profile to be `newprofile` and returns the previous default profile. It might be useful to keep the old profile so that you can reset the default later.

If the default profile has been deleted, or if it has never been set, `parallel.defaultClusterProfile` returns `'local'` as the default profile.

You can save modified profiles with the `saveProfile` or `saveAsProfile` method on a cluster object. You can create, delete, import, and modify profiles with the Cluster Profile Manager, accessible from the MATLAB desktop **Home** tab **Environment** area by selecting **Parallel > Create and Manage Clusters**.

Examples

Display the names of all available profiles and set the first in the list to be the default.

```
allProfiles = parallel.clusterProfiles
parallel.defaultClusterProfile(allProfiles{1});
```

First set the profile named `'MyProfile'` to be the default, and then set the profile named `'Profile2'` to be the default.

```
parallel.defaultClusterProfile('MyProfile');
oldDefault = parallel.defaultClusterProfile('Profile2');
strcmp(oldDefault,'MyProfile') % returns true
```

See Also
`parallel.clusterProfiles` | `parallel.importProfile`

Introduced in R2012a

parallel.exportProfile

Export one or more profiles to file

Syntax

```
parallel.exportProfile(profileName, filename)
parallel.exportProfile({profileName1,profileName2,...,profileNameN},filename)
```

Description

`parallel.exportProfile(profileName, filename)` exports the profile with the name `profileName` to specified filename. The extension `.mlsettings` is appended to the filename, unless already there.

`parallel.exportProfile({profileName1,profileName2,...,profileNameN},filename)` exports the profiles with the specified names to `filename`.

To import a profile, use `parallel.importProfile` or the Cluster Profile Manager.

Examples

Export the profile named `MyProfile` to the file `MyExportedProfile.mlsettings`.

```
parallel.exportProfile('MyProfile','MyExportedProfile')
```

Export the default profile to the file `MyDefaultProfile.mlsettings`.

```
def_profile = parallel.defaultClusterProfile();
parallel.exportProfile(def_profile,'MyDefaultProfile')
```

Export all profiles except for `local` to the file `AllProfiles.mlsettings`.

```
allProfiles = parallel.clusterProfiles();
% Remove 'local' from allProfiles
notLocal = ~strcmp(allProfiles,'local');
profilesToExport = allProfiles(notLocal);
if ~isempty(profilesToExport)
  parallel.exportProfile(profilesToExport,'AllProfiles');
end
```

See Also
parallel.clusterProfiles | parallel.importProfile

Introduced in R2012a

parallel.gpu.CUDAKernel

Create GPU CUDA kernel object from PTX and CU code

Syntax

```
KERN = parallel.gpu.CUDAKernel(PTXFILE,CPROTO)
KERN = parallel.gpu.CUDAKernel(PTXFILE,CPROTO,FUNC)
KERN = parallel.gpu.CUDAKernel(PTXFILE,CUFILE)
KERN = parallel.gpu.CUDAKernel(PTXFILE,CUFILE,FUNC)
```

Description

`KERN = parallel.gpu.CUDAKernel(PTXFILE,CPROTO)` and `KERN = parallel.gpu.CUDAKernel(PTXFILE,CPROTO,FUNC)` create a `CUDAKernel` object that you can use to call a CUDA kernel on the GPU. `PTXFILE` is the name of the file that contains the PTX code, or the contents of a PTX file as a character vector; and `CPROTO` is the C prototype for the kernel call that `KERN` represents. If specified, `FUNC` must be a character vector that unambiguously defines the appropriate kernel entry name in the PTX file. If `FUNC` is omitted, the PTX file must contain only a single entry point.

`KERN = parallel.gpu.CUDAKernel(PTXFILE,CUFILE)` and `KERN = parallel.gpu.CUDAKernel(PTXFILE,CUFILE,FUNC)` create a kernel object that you can use to call a CUDA kernel on the GPU. In addition, they read the CUDA source file `CUFILE`, and look for a kernel definition starting with `'__global__'` to find the function prototype for the CUDA kernel that is defined in `PTXFILE`.

For information on executing your kernel object, see "Run a CUDAKernel" on page 8-24.

Examples

If `simpleEx.cu` contains the following:

```
/*
 * Add a constant to a vector.
 */
__global__ void addToVector(float * pi, float c, int vecLen)  {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < vecLen) {
        pi[idx] += c;
    }
}
```

and `simpleEx.ptx` contains the PTX resulting from compiling `simpleEx.cu` into PTX, both of the following statements return a kernel object that you can use to call the `addToVector` CUDA kernel.

```
kern = parallel.gpu.CUDAKernel('simpleEx.ptx', ...
                                        'simpleEx.cu');
kern = parallel.gpu.CUDAKernel('simpleEx.ptx', ...
                                'float *,float,int');
```

See Also
arrayfun | existsOnGPU | feval | gpuArray | reset

Introduced in R2010b

parallel.gpu.enableCUDAForwardCompatibility

Query and set forward compatibility for GPU devices

Syntax

```
tf = parallel.gpu.enableCUDAForwardCompatibility()
parallel.gpu.enableCUDAForwardCompatibility(tf)
```

Description

`tf = parallel.gpu.enableCUDAForwardCompatibility()` returns `true` if forward compatibility for GPU devices is enabled and `false` otherwise. The default is `false`.

When forward compatibility is disabled, you cannot perform computations using a GPU device with an architecture that was released after the version of MATLAB you are using was built.

`parallel.gpu.enableCUDAForwardCompatibility(tf)` enables or disables forward compatibility for GPU devices. `tf` must be `true` (1) or `false` (0).

If you enable forward compatibility, the CUDA driver recompiles the GPU libraries the first time you access a device with an architecture newer than your MATLAB version. Recompilation can take several minutes.

Enabling forward compatibility is not persistent between MATLAB sessions.

Caution Enabling forward compatibility can result in wrong answers and unexpected behavior during GPU computations.

For more information, see "Forward Compatibility for GPU Devices" on page 8-40.

Examples

Check and Enable Forward Compatibility

If you have a GPU with an architecture that was released after the version of MATLAB you are using, by default, you cannot use that GPU to perform computations in MATLAB. To use that GPU in MATLAB, enable forward compatibility for GPU devices.

Check whether forward compatibility is enabled.

```
tf = parallel.gpu.enableCUDAForwardCompatibility()
```

```
tf =
    0
```

Enable forward compatibility.

```
parallel.gpu.enableCUDAForwardCompatibility(1)
```

Select and use the GPU device.

```
gpuDevice(2);
A = ones(100,'gpuArray');
```

The first time you access the GPU from MATLAB, the CUDA driver recompiles the libraries. Recompilation can take several minutes.

Input Arguments

tf — Forward compatibility status to set
true or 1 | false or 0

Forward compatibility status to set, specified as a numeric or logical 1 (true) or 0 (false).

Example: 0

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

See Also
gpuArray | gpuDevice | gpuDeviceCount

Topics
"GPU Support by Release" on page 8-38
"Run MATLAB Functions on a GPU" on page 8-9
"Identify and Select a GPU Device" on page 8-19
"Establish Arrays on a GPU" on page 8-3

Introduced in R2020b

parallel.gpu.RandStream.create

Package: `parallel.gpu`

Create independent random number streams on a GPU

Syntax

```
s = parallel.gpu.RandStream.create('gentype')
[s1,s2,...] = parallel.gpu.RandStream.create('gentype','NumStreams',n)
[ ___ ] = parallel.gpu.RandStream.create('gentype',Name,Value)
```

Description

`s = parallel.gpu.RandStream.create('gentype')` creates a single random number stream that uses the random number generator algorithm specified by `'gentype'`.

Note The `parallel.gpu.RandStream` object creation function is a more concise alternative when you want to create a single stream.

`[s1,s2,...] = parallel.gpu.RandStream.create('gentype','NumStreams',n)` creates n random number streams that use the random number generator algorithm specified by `'gentype'`. The streams are independent in a pseudorandom sense. The streams are not necessarily independent from streams created at other times.

`[ ___ ] = parallel.gpu.RandStream.create('gentype',Name,Value)` also specifies additional `Name,Value` pairs to control the creation of the stream, including the number of independent streams to create.

Examples

Create Multiple Random Number Streams Simultaneously

You can create multiple independent random number streams that have the same generator, seed, and normal transformations. Here, several independent streams are created and then used to generate independent streams of random numbers.

First, create the streams as a cell array.

```
streams = parallel.gpu.RandStream.create('Philox', 'NumStreams',3,'Seed',1,'NormalTransform','Inv

streams =

  1×3 cell array

    {1×1 parallel.gpu.RandStream}    {1×1 parallel.gpu.RandStream}    {1×1 parallel.gpu.RandStrea
```

Now, you can use each stream to generate random numbers. In this example, you create a matrix in which each row is generated from a different random number stream.

```
x = zeros(3,10,'gpuArray');
for i=1:3
    x(i,:) = rand(streams{i},1,10);
end
x

x =

    0.5361    0.2319    0.7753    0.2390    0.0036    0.5262    0.8629    0.9974    0.9576    0.0
    0.3084    0.3396    0.6758    0.5145    0.7909    0.7709    0.3386    0.1168    0.3694    0.0
    0.5218    0.5625    0.7090    0.5854    0.5067    0.6528    0.5095    0.8777    0.3094    0.1
```

Input Arguments

`'gentype'` — Random number generator algorithm
character vector | string

Random number generator, specified as a character vector or string for any valid random number generator that supports multiple streams and substreams. Three random number generator algorithms are supported on the GPU.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period in Full Precision
`'Threefry'` or `'Threefry4x64_20'`	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})
`'Philox'` or `'Philox4x32_10'`	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
`'CombRecursive'` or `'mrg32k3a'`	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})

For more information on the differences between generating random numbers on the GPU and CPU, see "Random Number Streams on a GPU" on page 8-6.

Example: `parallel.gpu.RandStream.create('Philox')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `parallel.gpu.RandStream.create('Philox','Seed',10)` creates a single random number stream using the `'Philox'` generator algorithm with seed `10`.

NumStreams — Number of streams
1 (default) | positive integer

Number of independent streams to be created, specified as the comma-separated pair consisting of `'NumStreams'` and a non-negative integer. The streams are independent in a pseudorandom sense. The streams are not necessarily independent from streams created at other times.

StreamIndices — Indices of the streams
[1:N] (default) | vector of integers | integer

Indices of the streams created in this function call, specified as the comma-separated pair consisting of `'StreamIndices'` and a nonnegative integer or a vector of nonnegative integers. The default value is `1:N`, where `N` is the value specified with the `'NumStreams'` parameter.

The values provided for `'StreamIndices'` must be less than or equal to the value provided for `'NumStreams'`.

Seed — Random number seed
0 (default) | nonnegative integer | `'shuffle'`

Random number seed for all streams initialized, specified as the comma-separated pair consisting of `'Seed'` and a nonnegative integer. The seed specifies the starting point for the algorithm to generate random numbers.

NormalTransform — Transformation algorithm for normally distributed random numbers
`'BoxMuller'` | `'Inversion'`

Transformation algorithm for normally distributed random numbers generated using the `randn` function, specified as the comma-separated pair `'NormalTransform'` and the algorithm names `'BoxMuller'` or `'Inversion'`. The `'BoxMuller'` algorithm is supported for the `'Threefry` and `'Philox'` generators. The `'Inversion'` algorithm is supported for the `'CombRecursive'` generator. No other transformation algorithms are supported on the GPU.

CellOutput — Return streams as cell array
0 (default) | 1

Logical flag indicating whether to return the stream objects as elements of a cell array, specified as the comma-separated pair `'CellOutput'` and the logical value `0` or `1`. The default value is false.

Output Arguments

s — Random number stream
`parallel.gpu.RandStream` object

Random number stream for generating random numbers on a GPU, returned as a `parallel.gpu.RandStream` object.

Tips

- If you create multiple streams by calling `parallel.gpu.RandStream.create` several times, the streams are not necessarily independent of each other. To create independent streams from separate calls of `parallel.gpu.RandStream.create`:

  - Specify the same set of values for `gentype`, `'NumStreams'`, and `'Seed'` in each case.

  - Specify a different value for `'StreamIndices'` that is between `1` and the `'NumStreams'` value in each case.

See Also
`gpurng` | `parallel.gpu.RandStream` | `parallel.gpu.RandStream.getGlobalStream` | `parallel.gpu.RandStream.list` | `parallel.gpu.RandStream.setGlobalStream`

Introduced in R2011b

parallel.gpu.RandStream.getGlobalStream

Package: `parallel.gpu`

Current global GPU random number stream

Syntax

```
stream = parallel.gpu.RandStream.getGlobalStream
```

Description

`stream = parallel.gpu.RandStream.getGlobalStream` returns the current global random number stream on the GPU.

> **Note** The `gpurng` function is a more concise alternative for many uses of `parallel.gpu.RandStream.setGlobalStream`.

Examples

Save the Default Global Stream

Use `parallel.gpu.RandStream.getGlobalStream` to save the default stream settings.

```
defaultStr = parallel.gpu.RandStream.getGlobalStream

defaultStr =

Threefry4x64_20 random stream on the GPU (current global stream)
             Seed: 0
  NormalTransform: BoxMuller
```

If you change the global stream, you can use the stream `defaultStr` to restore the default settings. For example, suppose that you change the global stream to a different stream.

```
newStr = parallel.gpu.RandStream('CombRecursive', 'NormalTransform','Inversion');
defaultStr = parallel.gpu.RandStream.setGlobalStream(newStr)

defaultStr =

Threefry4x64_20 random stream on the GPU
             Seed: 0
  NormalTransform: BoxMuller
```

`defaultStr` is no longer the current global GPU stream. Once you finish your calculations using the new global stream settings, you can reset the stream to the default settings.

```
newStr = parallel.gpu.RandStream.setGlobalStream(defaultStr)

newStr =
```

MRG32K3A random stream on the GPU
               Seed: 0
  NormalTransform: Inversion

defaultStr

defaultStr =

Threefry4x64_20 random stream on the GPU (current global stream)
               Seed: 0
  NormalTransform: BoxMuller

defaultStr is once again the current global stream.

Output Arguments

stream — Global GPU random number stream
parallel.gpu.RandStream object

Global random number stream for generating random numbers on a GPU, returned as a parallel.gpu.RandStream object.

See Also
gpurng | parallel.gpu.RandStream | parallel.gpu.RandStream.create | parallel.gpu.RandStream.setGlobalStream

Introduced in R2011b

parallel.gpu.RandStream.list

Package: `parallel.gpu`

Random number generator algorithms on the GPU

Syntax

`parallel.gpu.RandStream.list`

Description

`parallel.gpu.RandStream.list` lists the generator algorithms that can be used when creating a random number stream with `parallel.gpu.RandStream` or `parallel.gpu.RandStream.create`.

Examples

Available Generators on the GPU

When you use `parallel.gpu.RandStream.list`, MATLABdisplays a list of the available random number generators.

`parallel.gpu.RandStream.list`

```
The following random number generator algorithms are available:

MRG32K3A:        Combined multiple recursive generator (supports parallel streams)
Philox4x32_10:   Philox 4x32 generator with 10 rounds (supports parallel streams)
Threefry4x64_20: Threefry 4x64 generator with 20 rounds (supports parallel streams)
```

Each of these generators supports multiple parallel streams.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period in Full Precision
`'Threefry'` or `'Threefry4x64_20'`	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})
`'Philox'` or `'Philox4x32_10'`	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
`'CombRecursive'` or `'mrg32k3a'`	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})

For more information on the differences between generating random numbers on the GPU and CPU, see "Random Number Streams on a GPU" on page 8-6.

See Also

`gpurng` | `parallel.gpu.RandStream` | `parallel.gpu.RandStream.create` | `parallel.gpu.RandStream.getGlobalStream` | `parallel.gpu.RandStream.setGlobalStream`

Introduced in R2011b

parallel.gpu.RandStream.setGlobalStream

Package: parallel.gpu

Set GPU global random number stream

Syntax

prevStream = parallel.gpu.RandStream.setGlobalStream(stream)

Description

prevStream = parallel.gpu.RandStream.setGlobalStream(stream) replaces the global random number stream with the stream specified by stream.

Examples

Change the Global Stream

You can change the global random number stream on the GPU and store the old settings for the global stream. First, define the random number stream that you want to set as the new global stream.

newStr = parallel.gpu.RandStream('Philox','Seed',1,'NormalTransform','Inversion')

newStr =

Philox4x32_10 random stream on the GPU
              Seed: 1
  NormalTransform: Inversion

Next, set this new stream to be the global stream.

oldStr = parallel.gpu.RandStream.setGlobalStream(newStr)

oldStr =

Threefry4x64_20 random stream on the GPU
              Seed: 0
  NormalTransform: BoxMuller

oldStr holds the settings for the previous global random number stream on the GPU. The new global stream is newStr.

newStr

newStr =

Philox4x32_10 random stream on the GPU (current global stream)
              Seed: 1
  NormalTransform: Inversion

On a GPU, the functions `rand`, `randi`, and `randn` draw random numbers from the new global stream using the `'Philox'` generator algorithm.

Input Arguments

`stream` — **New global random number stream**
`parallel.gpu.RandStream` object

New global random number stream on the GPU, specified as a `parallel.gpu.RandStream` object. `stream` replaces the previous global stream.

Output Arguments

`prevStream` — **Previous global random number stream**
`parallel.gpu.RandStream` object

Previous global random number stream on the GPU, specified as a `parallel.gpu.RandStream` object.

See Also
gpurng | parallel.gpu.RandStream | parallel.gpu.RandStream.create | parallel.gpu.RandStream.getGlobalStream

Introduced in R2011b

parallel.importProfile

Import cluster profiles from file

Syntax

```
prof = parallel.importProfile(filename)
```

Description

`prof = parallel.importProfile(filename)` imports the profiles stored in the specified file and returns the names of the imported profiles. If `filename` has no extension, `.mlsettings` is assumed; configuration files must be specified with the `.mat` extension. Configuration `.mat` files contain only one profile, but profile `.mlsettings` files can contain one or more profiles. If only one profile is defined in the file, then `prof` is a character vector reflecting the name of the profile; if multiple profiles are defined in the file, then `prof` is a cell array of character vectors. If a profile with the same name as an imported profile already exists, an extension is added to the name of the imported profile.

You can use the imported profile with any functions that support profiles. `parallel.importProfile` does not set any of the imported profiles as the default; you can set the default profile by using the `parallel.defaultClusterProfile` function.

Profiles that were exported in a previous release are upgraded during import. Configurations are automatically converted to cluster profiles.

Imported profiles are saved as a part of your MATLAB settings, so these profiles are available in subsequent MATLAB sessions without importing again.

Examples

Import a profile from file `ProfileMaster.mlsettings` and set it as the default cluster profile.

```
profile_master = parallel.importProfile('ProfileMaster');
parallel.defaultClusterProfile(profile_master)
```

Import all the profiles from the file `ManyProfiles.mlsettings`, and use the first one to open a parallel pool.

```
profs = parallel.importProfile('ManyProfiles');
parpool(profs{1})
```

Import a configuration from the file `OldConfiguration.mat`, and set it as the default parallel profile.

```
old_conf = parallel.importProfile('OldConfiguration.mat')
parallel.defaultClusterProfile(old_conf)
```

See Also
`parallel.clusterProfiles` | `parallel.defaultClusterProfile` | `parallel.exportProfile`

Introduced in R2012a

parallel.pool.Constant

Build parallel.pool.Constant from data or function handle

Syntax

```
C = parallel.pool.Constant(X)
C = parallel.pool.Constant(FH)
C = parallel.pool.Constant(FH,CLEANUP)
C = parallel.pool.Constant(COMP)
```

Description

`C = parallel.pool.Constant(X)` copies the value X to each worker and returns a parallel.pool.Constant object, C, which allows each worker to access the value X within a parallel language construct (`parfor`, `spmd`, `parfeval`) using the property `C.Value`. This can improve performance when you have multiple parfor-loops accessing the same constant set of data, because X is transferred only once to the workers.

`C = parallel.pool.Constant(FH)` evaluates function handle FH on each worker and stores the result in `C.Value`. This is also useful for creating and using any handle-type resources on the workers, such as file handles and database connections.

`C = parallel.pool.Constant(FH,CLEANUP)` evaluates function handle FH on each worker and stores the result in `C.Value`. When C is cleared, the function handle CLEANUP is evaluated with a single argument `C.Value` on each worker.

`C = parallel.pool.Constant(COMP)` uses the values stored in the Composite COMP, and stores them in `C.Value` on each worker. This is especially useful when the data that you need to use inside a parfor-loop can be constructed only on the workers, such as when the data is too large to conveniently fit in the client, or when it is being loaded from a file that only the workers can access. If COMP does not have a defined value on every worker, an error results.

Tips

`parallel.pool.Constant` must be called in the MATLAB client session.

You can use a `parallel.pool.Constant` with an already running parallel pool or subsequent parallel pools.

Examples

Make parallel.pool.Constant from array in client

This example shows how to create a numeric parallel.pool.Constant, and use it in multiple `parfor`-loops on the same pool.

First, create some large data on the client, then build a parallel.pool.Constant, transferring the data to the pool only once.

```matlab
data = rand(1000);
c = parallel.pool.Constant(data);
for ii = 1:10
    % Run multiple PARFOR loops accessing the data.
    parfor jj = 1:10
        x(ii,jj) = c.Value(ii,jj);
    end
end
```

Make parallel.pool.Constant from function handle

This example shows how to create a parallel.pool.Constant with a function handle and a cleanup function.

Create a temporary file on each worker. By passing @fclose as the second argument, the file is automatically closed when c goes out of scope.

```matlab
c = parallel.pool.Constant(@() fopen(tempname(pwd),'wt'),@fclose);
spmd
    disp(fopen(c.Value)); % Displays the temporary filenames.
end

parfor idx = 1:1000
    fprintf(c.Value,'Iteration: %d\n',idx);
end
clear c;    % Closes the temporary files.
```

Make parallel.pool.Constant from Composite

This example shows how to build large data sets as a Composite on pool workers inside an spmd block, and then use that data as a parallel.pool.Constant inside a parfor-loop.

```matlab
spmd
  if labindex == 1
    x = labBroadcast(1,rand(5000));
  else
    x = labBroadcast(1);
  end
end
xc = parallel.pool.Constant(x);
parfor idx = 1:10
  s(idx) = sum(xc.Value(:,idx));
end
s
```

```
s =
   1.0e+03 *

    2.5108    2.5031    2.5123    2.4909    2.4957    2.5462    2.4859    2.5320    2.5076    2.5432
```

See Also
parcluster | parfeval | parfor | parpool | spmd

Introduced in R2015b

parcluster

Create cluster object

Syntax

```
c = parcluster
c = parcluster(profile)
```

Description

`c = parcluster` returns a cluster object representing the cluster identified by the default cluster profile, with the cluster object properties set to the values defined in that profile. Use a cluster object in functions such as `parpool` or `batch`.

`c = parcluster(profile)` returns a cluster object representing the cluster identified by the specified cluster profile, with the cluster object properties set to the values defined in that profile.

Examples

Create Cluster Object from Default Profile

This examples shows different ways of creating a cluster object from the default profile.

Find the cluster identified by the default parallel computing cluster profile, with the cluster object properties set to the values defined in that profile.

```
myCluster = parcluster;
```

View the name of the default profile and find the cluster identified by it. Open a parallel pool on the cluster.

```
defaultProfile = parallel.defaultClusterProfile
myCluster = parcluster(defaultProfile);
parpool(myCluster);
```

Create Cluster Object from Cluster Profile

Find a particular cluster using the profile named `'MyProfile'`, and create an independent job on the cluster.

```
myCluster = parcluster('MyProfile');
j = createJob(myCluster);
```

Input Arguments

profile — **Cluster profile**
string scalar | character vector

Cluster profile, specified as a string scalar or character vector.

You can save modified profiles with the `saveProfile` or `saveAsProfile` method on a cluster object. You can create, delete, import, and modify profiles with the Cluster Profile Manager, accessible from the MATLAB desktop **Home** tab **Environment** area by selecting **Parallel > Create and Manage Clusters**. For more information, see "Discover Clusters and Use Cluster Profiles" on page 5-11.

Example: `parcluster('MyCluster')`

Data Types: `char` | `string`

See Also
`createJob` | `parallel.Cluster` | `parallel.clusterProfiles` | `parallel.defaultClusterProfile` | `parpool`

Introduced in R2012a

parfeval

Package: `parallel`

Execute function asynchronously on parallel pool worker

Syntax

```
F = parfeval(p,fcn,numout,in1,in2,...)
F = parfeval(fcn,numout,in1,in2,...)
```

Description

`F = parfeval(p,fcn,numout,in1,in2,...)` requests asynchronous execution of the function `fcn` on a worker contained in the parallel pool `p`, expecting `numout` output arguments and supplying as input arguments `in1,in2,...`. The asynchronous evaluation of `fcn` does not block MATLAB. `F` is a parallel.FevalFuture object, from which the results can be obtained when the worker has completed evaluating `fcn`. The evaluation of `fcn` always proceeds unless you explicitly cancel execution by calling `cancel(F)`. To request multiple function evaluations, you must call `parfeval` multiple times. (However, `parfevalOnAll` can run the same function on all workers.)

`F = parfeval(fcn,numout,in1,in2,...)` requests asynchronous execution on the current parallel pool. If no pool exists, it starts a new parallel pool, unless your parallel preferences disable automatic creation of pools.

Examples

Query and Cancel parfeval Futures

When you use `parfeval` or `parfevalOnAll` to run computations in the background, you create objects called futures. You can use the `State` property of a future to find out whether it is running, queued or finished. You can also use the `FevalQueue` property of a parallel pool to access running and queued futures. To cancel futures, you can use the `cancel` function. In this example, you:

- Use `cancel` to cancel futures directly.
- Check completion errors on completed futures.
- Use the `FevalQueue` property to access futures.

Add Work to Queue

Create a parallel pool `p` with two workers.

```
p = parpool(2);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

When you use `parfeval` to run computations in the background, the function creates and adds a future for each computation to the pool queue. Tasks remain in the queue until a worker becomes

idle. When a worker becomes idle, it starts to compute a task if the queue is not empty. When a worker completes a task, the task is removed from the queue and the worker becomes idle.

Use `parfeval` to create an array of futures `f` by instructing workers to execute the function `pause`. Use an argument of `1` for the third future, and an argument of `Inf` for all other futures.

```
for n = 1:5
    if n == 3
        f(n) = parfeval(@pause,0,1);
    else
        f(n) = parfeval(@pause,0,Inf);
    end
end
```

Each use of `parfeval` returns a future object that represents the execution of a function on a worker. Except for the third future, every future will take an infinite amount of time to compute. The future created by `parfeval(@pause,0,Inf)` is an extreme case of a future which can slow down a queue.

Cancel Futures Directly

You can use the `State` property to obtain the status of futures. Construct a cell array of the state of each future in `f`.

```
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'running'}    {'queued'}    {'queued'}    {'queued'}
```

Every task except for the third pauses forever.

Cancel the second future directly with `cancel`.

```
cancel(f(2));
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'running'}    {'queued'}    {'queued'}
```

After you cancel the second future, the third future runs. Wait until the third future completes, then examine the states again.

```
wait(f(3));
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'finished'}    {'running'}    {'queued'}
```

The third future now has the state `'finished'`.

Check Completion Errors

When a future completes, its `State` property becomes `'finished'`. To distinguish between futures which are cancelled and complete normally, use the `Error` property.

```
fprintf("f(2): %s\n", f(2).Error.message)
```

```
f(2): Execution of the future was cancelled.
```

```
fprintf("f(3): %s\n", f(3).Error.message)
```

```
f(3):
```

The code cancels the second future, as the message property indicates. The second future was cancelled, as stated in the `message` property. The third future completes without error, and therefore does not have an error message.

Cancel Futures in Pool Queue

You can use the `FevalQueue` property to access the futures in the pool queue.

```
p.FevalQueue
```

```
ans =
 FevalQueue with properties:

        Number Queued: 1
       Number Running: 2
```

The queue has two properties: `RunningFutures` and `QueuedFutures`. The `RunningFutures` property is an array of futures corresponding to tasks that are currently running.

```
disp(p.FevalQueue.RunningFutures)
```

```
 1x2 FevalFuture array:

        ID            State  FinishDateTime  Function  Error
    ------------------------------------------------------
   1   12             running                @pause
   2   15             running                @pause
```

The `QueuedFutures` property is an array of futures corresponding to tasks that are currently queued and not running.

```
disp(p.FevalQueue.QueuedFutures)
```

```
 FevalFuture with properties:

                   ID: 16
             Function: @pause
       CreateDateTime: 15-Jul-2020 17:29:37
        StartDateTime:
     Running Duration: 0 days 0h 0m 0s
                State: queued
                Error: none
```

You can cancel a single future or an array of futures. Cancel all the futures in `QueuedFutures`.

```
cancel(p.FevalQueue.QueuedFutures);
{f.State}
```

```
ans = 1×5 cell
    {'running'}    {'finished'}    {'finished'}    {'running'}    {'finished'}
```

`RunningFutures` and `QueuedFutures` are sorted from newest to oldest, regardless of whether `f` is in order from newest to oldest. Each future has a unique `ID` property for the lifetime of the client. Check the `ID` property of each of the futures in `f`.

```
disp(f)
```

```
 1x5 FevalFuture array:

        ID                State          FinishDateTime  Function  Error
      --------------------------------------------------------------------
    1   12              running                            @pause
    2   13   finished (unread)   15-Jul-2020 17:29:37      @pause   Error
    3   14   finished (unread)   15-Jul-2020 17:29:39      @pause
    4   15              running                            @pause
    5   16   finished (unread)   15-Jul-2020 17:29:39      @pause   Error
```

Compare the result against the `ID` property of each of the `RunningFutures`.

```
for j = 1:length(p.FevalQueue.RunningFutures)
    rf = p.FevalQueue.RunningFutures(j);
    fprintf("p.FevalQueue.RunningFutures(%i): ID = %i\n", j, rf.ID)
end
```

```
p.FevalQueue.RunningFutures(1): ID = 12
p.FevalQueue.RunningFutures(2): ID = 15
```

Here, `RunningFutures` is an array containing `f(1)` and `f(4)`. If you cancel `RunningFutures(2)`, you cancel the fourth future `f(4)`.

Sometimes, futures are not available in the workspace, for example, if you execute the same piece of code twice before it finishes, or if you use `parfeval` in a function. You can cancel futures that are not available in the workspace.

Clear `f` from the workspace.

```
clear f
```

You can use `RunningFutures` and `QueuedFutures` to access futures that have not yet completed. Use `RunningFutures` to cancel `f(4)`.

```
rf2 = p.FevalQueue.RunningFutures(2);
cancel(rf2)
rf2.State
```

```
ans =
'finished'
```

To cancel all the futures still in the queue, use the following code.

```
cancel([p.FevalQueue.RunningFutures p.FevalQueue.QueuedFutures])
```

Execute Function Asynchronously and Fetch Outputs

Use `parfeval` to request asynchronous execution of a function on a worker.

For example, submit a single request to the parallel pool. Retrieve the outputs by using `fetchOutputs`.

```
f = parfeval(@magic,1,10);
value = fetchOutputs(f);
```

You can also submit a vector of multiple future requests in a `for`-loop and collect the results as they become available. For efficiency, preallocate an array of future objects before.

```
f(1:10) = parallel.FevalFuture;
for idx = 1:10
    f(idx) = parfeval(@magic,1,idx);
end
```

Retrieve the individual future outputs as they become available by using `fetchNext`.

```
magicResults = cell(1,10);
for idx = 1:10
    [completedIdx,value] = fetchNext(f);
    magicResults{completedIdx} = value;
    fprintf('Got result with index: %d.\n', completedIdx);
end
```

Plot During Parameter Sweep with `parfeval`

This example shows how to perform a parallel parameter sweep with `parfeval` and send results back during computations with a `DataQueue` object. `parfeval` does not block MATLAB, so you can continue working while computations take place.

The example performs a parameter sweep on the Lorenz system of ordinary differential equations, on the parameters σ and ρ, and shows the chaotic nature of this system.

$$\frac{\mathrm{d}}{\mathrm{d}t}x = \sigma(y - z)$$

$$\frac{\mathrm{d}}{\mathrm{d}t}y = x(\rho - z) - y$$

$$\frac{\mathrm{d}}{\mathrm{d}t}z = xy - \beta x$$

Create Parameter Grid

Define the range of parameters that you want to explore in the parameter sweep.

```
gridSize = 40;
sigma = linspace(5, 45, gridSize);
rho = linspace(50, 100, gridSize);
beta = 8/3;
```

Create a 2-D grid of parameters by using the `meshgrid` function.

```
[rho,sigma] = meshgrid(rho,sigma);
```

Create a figure object, and set `'Visible'` to `true` so that it opens in a new window, outside of the live script. To visualize the results of the parameter sweep, create a surface plot. Note that initializing the Z component of the surface with `NaN` creates an empty plot.

```
figure('Visible',true);
surface = surf(rho,sigma,NaN(size(sigma)));
```

```
xlabel('\rho','Interpreter','Tex')
ylabel('\sigma','Interpreter','Tex')
```


Set Up Parallel Environment

Create a pool of parallel workers by using the `parpool` function.

```
parpool;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

To send data from the workers, create a `DataQueue` object. Set up a function that updates the surface plot each time a worker sends data by using the `afterEach` function. The `updatePlot` function is a supporting function defined at the end of the example.

```
Q = parallel.pool.DataQueue;
afterEach(Q,@(data) updatePlot(surface,data));
```

Perform Parallel Parameter Sweep

After you define the parameters, you can perform the parallel parameter sweep.

`parfeval` works more efficiently when you distribute the workload. To distribute the workload, group the parameters to explore into partitions. For this example, split into uniform partitions of size `step` by using the colon operator (`:`). The resulting array `partitions` contains the boundaries of the partitions. Note that you must add the end point of the last partition.

```
step = 100;
partitions = [1:step:numel(sigma), numel(sigma)+1]
```

partitions = *1×17*

1	101	201	301	401	501	601	701

For best performance, try to split into partitions that are:

- Large enough that the computation time is large compared to the overhead of scheduling the partition.
- Small enough that there are enough partitions to keep all workers busy.

To represent function executions on parallel workers and hold their results, use future objects.

```
f(1:numel(partitions)-1) = parallel.FevalFuture;
```

Offload computations to parallel workers by using the `parfeval` function. `parameterSweep` is a helper function defined at the end of this script that solves the Lorenz system on a partition of the parameters to explore. It has one output argument, so you must specify 1 as the number of outputs in `parfeval`.

```
for ii = 1:numel(partitions)-1
    f(ii) = parfeval(@parameterSweep,1,partitions(ii),partitions(ii+1),sigma,rho,beta,Q);
end
```

`parfeval` does not block MATLAB, so you can continue working while computations take place. The workers compute in parallel and send intermediate results through the `DataQueue` as soon as they become available.

If you want to block MATLAB until `parfeval` completes, use the `wait` function on the future objects. Using the `wait` function is useful when subsequent code depends on the completion of `parfeval`.

```
wait(f);
```

After `parfeval` finishes the computations, `wait` finishes and you can execute more code. For example, plot the contour of the resulting surface. Use the `fetchOutputs` function to retrieve the results stored in the future objects.

```
results = reshape(fetchOutputs(f),gridSize,[]);
contourf(rho,sigma,results)
xlabel('\rho','Interpreter','Tex')
ylabel('\sigma','Interpreter','Tex')
```

If your parameter sweep needs more computational resources and you have access to a cluster, you can scale up your `parfeval` computations. For more information, see "Scale Up from Desktop to Cluster" on page 9-88.

Define Helper Functions

Define a helper function that solves the Lorenz system on a partition of the parameters to explore. Send intermediate results to the MATLAB client by using the `send` function on the `DataQueue` object.

```
function results = parameterSweep(first,last,sigma,rho,beta,Q)
    results = zeros(last-first,1);
    for ii = first:last-1
        lorenzSystem = @(t,a) [sigma(ii)*(a(2) - a(1)); a(1)*(rho(ii) - a(3)) - a(2); a(1)*a(2)
        [t,a] = ode45(lorenzSystem,[0 100],[1 1 1]);
        result = a(end,3);
        send(Q,[ii,result]);
        results(ii-first+1) = result;
    end
end
```

Define another helper function that updates the surface plot when new data arrives.

```
function updatePlot(surface,data)
    surface.ZData(data(1)) = data(2);
```

```
    drawnow('limitrate');
end
```

Update a User Interface Asynchronously Using afterEach and afterAll

You can perform asynchronous computations on workers using `parfeval` and leave the user interface responsive. Use `afterEach` to update the user interface when intermediate computations are ready. Use `afterAll` to update the user interface when all the computations are ready.

Create a simple user interface using a `waitbar`.

```
h = waitbar(0, 'Waiting...');
```


Use `parfeval` to carry out time-consuming computations in the workers, for example, eigenvalues of random matrices. The computations happen asynchronously and the user interface updates during computation. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```
for idx = 1:100
    f(idx) = parfeval(@(n) real(eig(randn(n))), 1, 5e2);
end
```

Compute the largest value in each of the computations when they become ready using `afterEach`. Update the proportion of finished futures in the waitbar when each of them completes using `afterEach`.

```
maxFuture = afterEach(f, @max, 1);
updateWaitbarFuture = afterEach(f, @(~) waitbar(sum(strcmp('finished', {f.State}))/numel(f), h),
```

Close the waitbar when all the computations are done. Use `afterAll` on `updateWaitbarFuture` to continue automatically with a close operation. `afterAll` obtains the figure handle from `updateWaitbarFuture` and executes its function on it.

```
closeWaitbarFuture = afterAll(updateWaitbarFuture, @(h) delete(h), 0);
```

Show a histogram after all the maximum values are computed. Use `afterAll` on `maxFuture` to continue the operation automatically. `afterAll` obtains the maximum values from `maxFuture` and calls `histogram` on them.

```
showsHistogramFuture = afterAll(maxFuture, @histogram, 0);
```

Input Arguments

p — Parallel pool
parallel.Pool object

Parallel pool of workers, specified as a `parallel.Pool` object. You can create a parallel pool by using the `parpool` function.

Data Types: `parallel.Pool`

`fcn` — Function to execute
function_handle

Function to execute on a worker, specified as a function handle.

Example: `fcn = @sum`

Data Types: `function_handle`

`numout` — Number of output arguments
integer

Number of output arguments that are expected from `fcn`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`in1,in2,...` — Function arguments
comma-separated list of variables or expressions

Function arguments to pass to `fcn`, specified as a comma-separated list of variables or expressions.

Output Arguments

`F` — Future
parallel.FevalFuture

Future object, returned as a `parallel.FevalFuture`, that represents the execution of `fcn` on a parallel worker and holds its results. Use `fetchOutputs` or `fetchNext` to collect the results.

See Also
afterAll | afterEach | cancel | fetchNext | fetchOutputs | isequal | parallel.FevalFuture | parallel.pool.Constant | parfevalOnAll | parpool | ticBytes | tocBytes | wait

Introduced in R2013b

parfevalOnAll

Execute function asynchronously on all workers in parallel pool

Syntax

```
F = parfevalOnAll(p,fcn,numout,in1,in2,...)
F = parfevalOnAll(fcn,numout,in1,in2,...)
```

Description

`F = parfevalOnAll(p,fcn,numout,in1,in2,...)` requests the asynchronous execution of the function `fcn` on all workers in the parallel pool `p`. `parfevalOnAll` evaluates `fcn` on each worker with input arguments `in1,in2,...`, and expects `numout` output arguments. `F` is a parallel.FevalOnAllFuture object, from which you can obtain the results when all workers have completed executing `fcn`.

`F = parfevalOnAll(fcn,numout,in1,in2,...)` requests asynchronous execution on all workers in the current parallel pool. If no pool exists, it starts a new parallel pool, unless your parallel preferences disable automatic creation of pools.

Note Use `parfevalOnAll` instead of `parfor` or `spmd` if you want to use `clear`. This preserves workspace transparency. See "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49.

Examples

Run Functions on All Workers

Unload a `mex` file before deleting temporary folders for distributing simulations, using the `clear` function. Because `clear` has `0` output arguments, specify `0` in the `numout` input argument of `parfevalOnAll`.

```
parfevalOnAll(@clear,0,'mex');
```

Close all Simulink models on all workers:

```
p = gcp(); % Get the current parallel pool
f = parfevalOnAll(p,@bdclose,0,'all');
% No output arguments, but you might want to wait for completion
wait(f);
```

Input Arguments

p — Parallel pool
parallel.Pool object

Parallel pool of workers, specified as a `parallel.Pool` object. You can create a parallel pool by using the `parpool` function.

Data Types: `parallel.Pool`

`fcn` — Function to execute
function handle

Function to execute on the workers, specified as a function handle.

Example: `fcn = @sum`

Data Types: `function_handle`

`numout` — Number of output arguments
integer

Number of output arguments that are expected from `fcn`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`in1,in2,...` — Function arguments
comma-separated list of variables or expressions

Function arguments to pass to `fcn`, specified as a comma-separated list of variables or expressions.

Output Arguments

`F` — Future
`parallel.FevalFuture`

Future object, returned as a `parallel.FevalOnAllFuture`, that represents the execution of `fcn` on the parallel workers and holds their results. Use `fetchOutputs` to collect the results.

See Also
`cancel` | `fetchNext` | `fetchOutputs` | `parallel.pool.Constant` | `parfeval` | `parpool` | `wait`

Introduced in R2013b

parfor

Execute `for`-loop iterations in parallel on workers

Syntax

```
parfor loopVar = initVal:endVal; statements; end
parfor (loopVar = initVal:endVal,M); statements; end
parfor (loopVar = initVal:endVal,opts); statements; end
parfor (loopVar = initVal:endVal,cluster); statements; end
```

Description

`parfor loopVar = initVal:endVal; statements; end` executes `for`-loop iterations in parallel on workers in a parallel pool.

MATLAB executes the loop body commands in `statements` for values of `loopVar` between `initVal` and `endVal`. `loopVar` specifies a vector of integer values increasing by 1. If you have Parallel Computing Toolbox, the iterations of `statements` can execute on a parallel pool of workers on your multi-core computer or cluster. As with a `for`-loop, you can include a single line or multiple lines in `statements`.

To find out how `parfor` can help increase your throughput, see "Decide When to Use parfor" on page 2-2.

`parfor` differs from a traditional `for`-loop in the following ways:

- Loop iterations are executed in parallel in a nondeterministic order. This means that you might need to modify your code to use `parfor`. For more help, see "Convert for-Loops Into parfor-Loops" on page 2-6.
- Loop iterations must be consecutive, increasing integer values.
- The body of the `parfor`-loop must be independent. One loop iteration cannot depend on a previous iteration, because the iterations are executed in a nondeterministic order. For more help, see "Ensure That parfor-Loop Iterations are Independent" on page 2-9.
- You cannot use a `parfor`-loop inside another `parfor`-loop. For more help, see "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12.

`parfor (loopVar = initVal:endVal,M); statements; end` uses M to specify the maximum number of workers from the parallel pool to use in evaluating `statements` in the loop body. M must be a nonnegative integer.

By default, MATLAB uses the available workers in your parallel pool. You can change the number of workers on the **Home** tab in the **Environment** section, by selecting **Parallel > Parallel Preferences**. You can override the default number of workers in a parallel pool by using `parpool`. When no workers are available in the pool or M is zero, MATLAB still executes the loop body in a nondeterministic order, but not in parallel. Use this syntax to switch between parallel and serial execution when testing your code.

With this syntax, to execute the iterations in parallel, you must have a parallel pool of workers. By default, if you execute `parfor`, you automatically create a parallel pool of workers on the cluster

defined by your default cluster profile. The default cluster is **local**. You can change your cluster in **Parallel Preferences**. For more details, see "Specify Your Parallel Preferences" on page 5-9.

`parfor (loopVar = initVal:endVal,opts); statements; end` uses `opts` to specify the resources to use in evaluating `statements` in the loop body. Create a set of `parfor` options using the `parforOptions` function. With this approach, you can run `parfor` on a cluster without first creating a parallel pool and control how `parfor` partitions the iterations into subranges for the workers.

`parfor (loopVar = initVal:endVal,cluster); statements; end` executes `statements` on workers in `cluster` without creating a parallel pool. This is equivalent to executing `parfor (loopVar = initVal:endVal,parforOptions(cluster)); statements; end`.

Examples

Convert a `for`-Loop Into a `parfor`-Loop

Create a `parfor`-loop for a computationally intensive task and measure the resulting speedup.

In the MATLAB Editor, enter the following `for`-loop. To measure the time elapsed, add `tic` and `toc`.

```
tic
n = 200;
A = 500;
a = zeros(1,n);
for i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

Run the script, and note the elapsed time.

```
Elapsed time is 31.935373 seconds.
```

In the script, replace the `for`-loop with a `parfor`-loop.

```
tic
n = 200;
A = 500;
a = zeros(1,n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

Run the new script, and run it again. The first run is slower than the second run, because the parallel pool has to be started, and you have to make the code available to the workers. Note the elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

```
Elapsed time is 10.760068 seconds.
```

Observe that you speed up your calculation by converting the `for`-loop into a `parfor`-loop on four workers. You might reduce the elapsed time further by increasing the number of workers in your

parallel pool. For more information, see "Convert for-Loops Into parfor-Loops" on page 2-6 and "Scale Up parfor-Loops to Cluster and Cloud" on page 2-20.

Test parfor-Loops by Switching Between Parallel and Serial Execution

You can specify the maximum number of workers M for a parfor-loop. Set M = 0 to run the body of the loop in the desktop MATLAB, without using workers, even if a pool is open. When M = 0, MATLAB still executes the loop body in a nondeterministic order, but not in parallel, so that you can check whether your parfor-loops are independent and suitable to run on workers. This is the simplest way to allow you to debug the contents of a parfor-loop. You cannot set breakpoints directly in the body of the parfor-loop, but you can set breakpoints in functions called from the body of the parfor-loop.

Specify M = 0 to run the body of a parfor-loop in the desktop MATLAB, even if a pool is open.

```
M = 0;                      % M specifies maximum number of workers
y = ones(1,100);
parfor (i = 1:100,M)
     y(i) = i;
end
```

To control the number of workers in your parallel pool, see "Specify Your Parallel Preferences" on page 5-9 and parpool.

Measure Data Transferred to Workers Using a parfor-Loop

To measure how much data is transferred to and from the workers in your current parallel pool, add ticBytes(gcp) and tocBytes(gcp) before and after the parfor-loop. Use gcp as an argument to get the current parallel pool.

Delete your current parallel pool if you still have one.

```
delete(gcp('nocreate'))

tic
ticBytes(gcp);
n = 200;
A = 500;
a = zeros(1,n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
tocBytes(gcp)
toc
```

Run the new script, and run it again. The first run is slower than the second run, because the parallel pool has to be started, and you have to make the code available to the workers.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
...
          BytesSentToWorkers      BytesReceivedFromWorkers
```

	_____	_____
1	15340	7024
2	13328	5712
3	13328	5704
4	13328	5728
Total	55324	24168

You can use the `ticBytes` and `tocBytes` results to examine the amount of data transferred to and from the workers in a parallel pool. In this example, the data transfer is small. For more information about `parfor`-loops, see "Decide When to Use parfor" on page 2-2 and "Convert for-Loops Into parfor-Loops" on page 2-6.

Run `parfor` on a Cluster Without a Parallel Pool

Create a cluster object using the `parcluster` function, and create a set of `parfor` options with it. By default, `parcluster` uses your default cluster profile. Check your default profile on the MATLAB **Home** tab, in **Parallel** > **Select a Default Cluster**.

```
cluster = parcluster;
```

To run `parfor` computations directly in the cluster, pass the cluster object as the second input argument to `parfor`.

When you use this approach, `parfor` can use all the available workers in the cluster, and workers become available as soon as the loop completes. This approach is also useful if your cluster does not support parallel pools. If you want to control other options, including partitioning of iterations, use `parforOptions`.

```
values = [3 3 3 7 3 3 3];
parfor (i=1:numel(values),cluster)
    out(i) = norm(pinv(rand(values(i)*1e3)));
end
```

Use this syntax to run parfor on a large cluster without consuming workers for longer than necessary.

Input Arguments

`loopVar` — Loop index
integer

Loop index variable with initial value `initVal` and final value `endVal`. The variable can be any numeric type and the value must be an integer.

Make sure that your `parfor`-loop variables are consecutive increasing integers. For more help, see "Troubleshoot Variables in parfor-Loops" on page 2-28.

The range of the `parfor`-loop variable must not exceed the supported range. For more help, see "Avoid Overflows in parfor-Loops" on page 2-28.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`initVal` — Initial value of loop index
integer

Initial value loop index variable, `loopVar`. The variable can be any numeric type and the value must be an integer. With `endVal`, specifies the `parfor` range vector, which must be of the form `M:N`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

endVal — Final value of loop index
integer

Final value loop index variable, `loopVar`. The variable can be any numeric type and the value must be an integer. With `initVal`, specifies the `parfor` range vector, which must be of the form `M:N`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

statements — Loop body
text

Loop body, specified as text. The series of MATLAB commands to execute in the `parfor`-loop.

You might need to modify your code to use `parfor`-loops. For more help, see "Convert for-Loops Into parfor-Loops" on page 2-6

Do not nest `parfor`-loops, see "Nested parfor and for-Loops and Other parfor Requirements" on page 2-12.

M — Maximum number of workers running in parallel
number of workers in the parallel pool (default) | nonnegative integer

Maximum number of workers running in parallel, specified as a nonnegative integer. If you specify an upper limit, MATLAB uses no more than this number, even if additional workers are available. If you request more workers than the number of available workers, then MATLAB uses the maximum number of workers available at the time of the call. If the loop iterations are fewer than the number of workers, some workers perform no work.

If `parfor` cannot run on multiple workers (for example, if only one core is available or `M` is 0), MATLAB executes the loop in a serial manner. In this case, MATLAB still executes the loop body in a nondeterministic order. Use this syntax to switch between parallel and serial when testing your code.

opts — parfor options
`parforOptions` object

`parfor` options, specified as a `ClusterOptions` object. Use the `parforOptions` function to create a set of `parfor` options.

Example: `opts = parforOptions(parcluster);`

cluster — Cluster
`parallel.Cluster`

Cluster, specified as a `parallel.Cluster` object, on which `parfor` runs. To create a cluster object, use the `parcluster` function.

Example: `cluster = parcluster('local')`

Data Types: `parallel.Cluster`

Tips

- Use a `parfor`-loop when:

  - You have many loop iterations of a simple calculation. `parfor` divides the loop iterations into groups so that each thread can execute one group of iterations.

  - You have some loop iterations that take a long time to execute.

- Do not use a `parfor`-loop when an iteration in your loop depends on the results of other iterations.

  Reductions are one exception to this rule. A reduction variable accumulates a value that depends on all the iterations together, but is independent of the iteration order. For more information, see "Reduction Variables" on page 2-41.

- When you use `parfor`, you have to wait for the loop to complete to obtain your results. Your client MATLAB is blocked and you cannot break out of the loop early. If you want to obtain intermediate results, or break out of a `for`-loop early, try `parfeval` instead.

- Unless you specify a cluster object, a `parfor`-loop runs on the existing parallel pool. If no pool exists, `parfor` starts a new parallel pool, unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `parfor` cannot start one, the loop runs serially in the client session.

- If the `AutoAttachFiles` property in the cluster profile for the parallel pool is set to `true`, MATLAB performs an analysis on a `parfor`-loop to determine what code files are necessary for its execution, see `listAutoAttachedFiles`. Then MATLAB automatically attaches those files to the parallel pool so that the code is available to the workers.

- You cannot call scripts directly in a `parfor`-loop. However, you can call functions that call scripts.

- Do not use `clear` inside a `parfor` loop because it violates workspace transparency. See "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49.

- You can run Simulink models in parallel with the `parsim` command instead of using `parfor`-loops. For more information and examples of using Simulink in parallel, see "Run Multiple Simulations" (Simulink).

See Also

`afterEach` | `for` | `gcp` | `listAutoAttachedFiles` | `parfeval` | `parforOptions` | `parpool` | `send` | `ticBytes` | `tocBytes`

Topics
"Decide When to Use parfor" on page 2-2
"Convert for-Loops Into parfor-Loops" on page 2-6
"Ensure That parfor-Loop Iterations are Independent" on page 2-9
"Nested parfor and for-Loops and Other parfor Requirements" on page 2-12
"Troubleshoot Variables in parfor-Loops" on page 2-28
"Scale Up parfor-Loops to Cluster and Cloud" on page 2-20
"Specify Your Parallel Preferences" on page 5-9
"Run Parallel Simulations" (Simulink)

Introduced in R2008a

parforOptions

Options for `parfor`, such as partitioning iterations

Syntax

```
opts = parforOptions(cluster)
opts = parforOptions(pool)
opts = parforOptions( ___ ,'RangePartitionMethod',method)
opts = parforOptions( ___ ,'RangePartitionMethod','fixed','SubrangeSize',n)
opts = parforOptions(cluster, ___ ,Name,Value)
```

Description

`opts = parforOptions(cluster)` creates a set of options for `parfor` that instructs `parfor` to execute statements on workers in `cluster` without creating a parallel pool. Instead, `parfor` submits independent tasks to the cluster to execute the body of the loop. `parfor` partitions and distributes iterations to workers according to the `NumWorkers` property, the number of workers available, of `cluster`.

`opts = parforOptions(pool)` creates a set of options for `parfor` that instructs `parfor` to execute statements on workers in the parallel pool `pool`.

`opts = parforOptions( ___ ,'RangePartitionMethod',method)` uses `method` to define how to divide `parfor`-loop iterations into subranges. A subrange is a contiguous block of loop iterations that `parfor` executes as a group on a worker.

`opts = parforOptions( ___ ,'RangePartitionMethod','fixed','SubrangeSize',n)` divides `parfor`-loop iterations into subranges of size no larger than `n`.

`opts = parforOptions(cluster, ___ ,Name,Value)` specifies additional options for use with the cluster object `cluster`.

Examples

Run `parfor` on a Cluster Without a Parallel Pool

Create a cluster object using the `parcluster` function, and create a set of `parfor` options with it. By default, `parcluster` uses your default cluster profile. Check your default profile on the MATLAB® **Home** tab, in **Parallel > Select a Default Cluster**.

```
cluster = parcluster;
opts = parforOptions(cluster);
```

To run `parfor` computations directly in the cluster, pass the `parfor` options as the second input argument to `parfor`.

When you use this approach, `parfor` can use all the available workers in the cluster, and workers become available as soon as the loop completes. This approach is also useful if your cluster does not support parallel pools.

```
values = [3 3 3 7 3 3 3];
parfor (i=1:numel(values),opts)
    out(i) = norm(pinv(rand(values(i)*1e3)));
end
```

Use this syntax to run parfor on a large cluster without consuming workers for longer than necessary.

Control `parfor` Range Partitioning

You can control how `parfor` divides iterations into subranges for the workers with `parforOptions`. Controlling the range partitioning can optimize performance of a `parfor`-loop. For best performance, try to split into subranges that are:

- Large enough that the computation time is large compared to the overhead of scheduling the subrange
- Small enough that there are enough subranges to keep all workers busy

To partition iterations into subranges of fixed size, create a set of `parfor` options, set `'RangePartitionMethod'` to `'fixed'`, and specify a subrange size with `'SubrangeSize'`.

```
opts = parforOptions(parcluster,'RangePartitionMethod','fixed','SubrangeSize',2);
```

Pass the `parfor` options as the second input argument to `parfor`. In this case, `parfor` divides iterations into three groups of 2 iterations.

```
values = [3 3 3 3 3 3];
parfor (i=1:numel(values),opts)
    out(i) = norm(pinv(rand(values(i)*1e3)));
end
```

To partition iterations into subranges of varying size, pass a function handle to the `'RangePartitionMethod'` name-value pair. This function must return a vector of subrange sizes, and their sum must be equal to the number of iterations. For more information on this syntax, see "method" on page 11-0 .

```
opts = parforOptions(parcluster,'RangePartitionMethod', @(n,nw) [2 1 1 2]);
```

Pass the `parfor` options as the second input argument to `parfor`. In this case, `parfor` divides iterations into four groups of 2, 1, 1, and 2 iterations.

```
values = [3 3 7 7 3 3];
parfor (i=1:numel(values),opts)
    out(i) = norm(pinv(rand(values(i)*1e3)));
end
```

Run `parfor` on a Parallel Pool and Control Options

You can use `parforOptions` to run `parfor` on the workers of a parallel pool. Use this approach when you want to reserve a fixed number of workers for the `parfor`-loop. You can also have finer control on how `parfor` divides iterations for workers.

Create a parallel pool using the `parpool` function. By default, `parpool` uses your default cluster profile. Check your default profile on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. Create a set of `parfor` options with the parallel pool object, and specify options. For example, specify subranges of fixed size 2 as the partitioning method.

```
p = parpool;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
opts = parforOptions(p,'RangePartitionMethod','fixed','SubrangeSize',2);
```

Pass the `parfor` options as the second input argument to the `parfor` function. `parfor` runs the loop body on the parallel pool and divides iterations according to `opts`.

```
values = [3 3 3 3 3 3];
parfor (i=1:numel(values),opts)
    out(i) = norm(pinv(rand(values(i)*1e3)));
end
```

Transfer Files to `parfor` Workers

When you run `parfor` with or without a parallel pool, by default, MATLAB performs an automatic dependency analysis on the loop body. MATLAB transfers required files to the workers before running the statements. In some cases, you must explicitly transfer those files to the workers. For more information, see "Identify Program Dependencies".

If you are using `parfor` without a parallel pool, use `parforOptions` to transfer files. Create a cluster object using the `parcluster` option. Create a set of `parfor` options with the cluster object using the `parforOptions` function. To transfer files to the workers, use the `'AttachedFiles'` name-value pair.

```
cluster = parcluster;
opts = parforOptions(cluster,'AttachedFiles',{'myFile.dat'});
```

Pass the `parfor` options as the second input argument to the `parfor` function. The workers can access the required files in the loop body.

```
parfor (i=1:2,opts)
    M = csvread('myFile.dat',0,2*(i-1),[0,2*(i-1),1,1+2*(i-1)]);
    out(i) = norm(rand(ceil(norm(M))*1e3));
end
```

Input Arguments

`cluster` — Cluster
parallel.Cluster object

Cluster, specified as a `parallel.Cluster` object, on which `parfor` runs. To create a cluster object, use the `parcluster` function.

Example: `opts = parforOptions(parcluster('local'));`

Data Types: parallel.Cluster

n — Subrange size
positive integer

Subrange size for the fixed partitioning method, specified as a positive integer. You must set the name-value pair `'RangePartitionMethod'` to `'fixed'`.

Example: `opts = parforOptions(cluster,'RangePartitionMethod','fixed','SubrangeSize',5);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

pool — Parallel pool
`parallel.Pool` object

Parallel pool, specified as a `parallel.Pool` object, on which `parfor` runs. To create a parallel pool, use the `parpool` function.

Example: `opts = parforOptions(parpool('local'));`

Data Types: `parallel.Pool`

method — Range partition method
`'auto'` (default) | `'fixed'` | function handle

Range partition method, specified as one of the following:

- `'auto'` – This method divides the loop into subranges of varying sizes. It tries to achieve good performance for a variety of `parfor`-loops.
- `'fixed'` – Partition the `parfor`-loop into subranges of fixed sizes. You must specify the size of each subrange by setting the `'SubrangeSize'` name-value pair.
- function handle – Use a function handle to divide the loop iterations into subranges. The function must be of the form `sizes = customFcn(n,nw)`, where `n` is the number of iterations in the `parfor`-loop, and `nw` is the number of workers available to execute the loop. When the loop is running on a parallel pool, `nw` is the number of workers in the parallel pool. When the loop is running using a cluster, `nw` is the `NumWorkers` property of the cluster. `customFcn` returns a vector `sizes` of subrange sizes. This vector must satisfy `sum(sizes) == n`. For more information on function handles, see "Create Function Handle".

Example: `opts = parforOptions(cluster,'RangePartitionMethod','fixed','SubrangeSize',10);`

Example: `opts = parforOptions(cluster,'RangePartitionMethod',@(n,nw) [2,4,4,n-10]);`

Data Types: `char` | `function_handle`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `opts = parforOptions(cluster,'AttachedFiles',{'myFile.dat'});`

AutoAddClientPath — Flag to specify if client path is added to worker path
`true` (default) | `false`

Flag to specify if client path is added to worker path, specified as the comma-separated pair consisting of `AutoAddClientPath` and `true` or `false`.

Data Types: `logical`

AutoAttachFiles — Flag to enable dependency analysis
`true` (default) | `false`

Flag to enable dependency analysis on the `parfor`-loop body and transfer required files to the workers, specified as the comma-separated pair consisting of `AutoAttachFiles` and `true` or `false`.

Data Types: `logical`

AttachedFiles — Files to transfer
character vector | string scalar | string array | cell array of character vectors

Files to transfer to the workers, specified as the comma-separated pair consisting of `AttachedFiles` and a character vector, string, string array, or cell array of character vectors.

Example: `{'myFun.m','myFun2.m'}`

Data Types: `char` | `string` | `cell`

AdditionalPaths — Paths to add to workers
character vector | string scalar | string array | cell array of character vectors

Paths to add to the MATLAB path of the workers before `parfor` executes, specified as the comma-separated pair consisting of `AdditionalPaths` and a character vector, string, string array, or cell array of character vectors.

Example: `{'some/path/','another/path'}`

Data Types: `char` | `string` | `cell`

Output Arguments

opts — parfor options
`parforOptions` object

`parfor` options, returned as a `parforOptions` object. To specify options for a `parfor`-loop, pass a set of `parfor` options to the second input argument of `parfor`.

Example: `parfor(i=1:10,parforOptions(parcluster)); out(i)=i; end`

See Also
`parcluster` | `parfor` | `parpool`

Topics
"Profile Parallel Code" on page 9-3

Introduced in R2019a

parpool

Create parallel pool on cluster

Syntax

```
parpool
parpool(poolsize)
parpool(resources)
parpool(resources,poolsize)
parpool( ___ ,Name,Value)
poolobj = parpool( ___ )
```

Description

`parpool` starts a parallel pool of workers using the default cluster profile. With default preferences, MATLAB starts a pool on the local machine with one worker per physical CPU core, up to the preferred number of workers. For more information on parallel preferences, see "Specify Your Parallel Preferences" on page 5-9.

In general, the pool size is specified by your parallel preferences and the default profile. `parpool` creates a pool on the default cluster with its `NumWorkers` in the range `[1, preferredNumWorkers]` for running parallel language features. `preferredNumWorkers` is the value defined in your parallel preferences. For all factors that can affect your pool size, see "Pool Size and Cluster Selection" on page 2-58.

`parpool` enables the full functionality of the parallel language features in MATLAB by creating a special job on a pool of workers, and connecting the MATLAB client to the parallel pool. Parallel language features include `parfor`, `parfeval`, `parfevalOnAll`, `spmd`, and `distributed`. If possible, the working folder on the workers is set to match that of the MATLAB client session.

`parpool(poolsize)` creates and returns a pool with the specified number of workers. `poolsize` can be a positive integer or a range specified as a 2-element vector of integers. If `poolsize` is a range, the resulting pool has size as large as possible in the range requested.

Specifying the `poolsize` overrides the number of workers specified in the preferences or profile, and starts a pool of exactly that number of workers, even if it has to wait for them to be available. Most clusters have a maximum number of workers they can start. If the profile specifies a MATLAB Job Scheduler cluster, `parpool` reserves its workers from among those already running and available under that MATLAB Job Scheduler. If the profile specifies a local or third-party scheduler, `parpool` instructs the scheduler to start the workers for the pool.

`parpool(resources)` or `parpool(resources,poolsize)` starts a worker pool on the resources specified by `resources`.

`parpool( ___ ,Name,Value)` applies the specified values for certain properties when starting the pool.

`poolobj = parpool( ___ )` returns a parallel.Pool object to the client workspace representing the pool on the cluster. You can use the pool object to programmatically delete the pool or to access its properties. Use `delete(pool)` to shut down the parallel pool.

Examples

Create Pool from Default Profile

Start a parallel pool using the default profile to define the number of workers. With default preferences, the default pool is on the local machine.

```
parpool
```

Create Pool on Local Machine

You can create pools on different types of parallel environments on your local machine.

*   Start a parallel pool of process workers.

    ```
    parpool('local')
    ```
*   Start a parallel pool of thread workers.

    ```
    parpool('threads')
    ```

Fore more information on parallel environments, see "Choose Between Thread-Based and Process-Based Environments" on page 2-60.

Create Pool from Specified Profile

Start a parallel pool of 16 workers using a profile called `myProf`.

```
parpool('myProf',16)
```

Create Pool on Specified Cluster

Create an object representing the cluster identified by the default profile, and use that cluster object to start a parallel pool. The pool size is determined by the default profile.

```
c = parcluster
parpool(c)
```

Create Pool and Attach Files

Start a parallel pool with the default profile, and pass two code files to the workers.

```
parpool('AttachedFiles',{'mod1.m','mod2.m'})
```

Return Pool Object and Delete Pool

Create a parallel pool with the default profile, and later delete the pool.

```
poolobj = parpool;

delete(poolobj)
```

Determine Size of Current Pool

Find the number of workers in the current parallel pool.

```
poolobj = gcp('nocreate'); % If no pool, do not create new one.
if isempty(poolobj)
    poolsize = 0;
else
    poolsize = poolobj.NumWorkers
end
```

Input Arguments

`poolsize` — Size of parallel pool
positive integer | 2-element vector of integers

Size of the parallel pool, specified as a positive integer or a range specified as a 2-element vector of integers. If `poolsize` is a range, the resulting pool has size as large as possible in the range requested. Set the default preferred number of workers in the parallel preferences or parallel profile.

Example: `parpool('local',2)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`resources` — Resources to start pool on
`'local'` (default) | `'threads'` | profile name | cluster object

Resources to start the pool on, specified as `'local'`, `'threads'`, a cluster profile name or cluster object.

- `'local'` – Starts a pool of process workers on the local machine. For more information on process-based environments, see "Choose Between Thread-Based and Process-Based Environments" on page 2-60.
- `'threads'` – Starts a pool of thread workers on the local machine. For more information on thread-based environments, see "Choose Between Thread-Based and Process-Based Environments" on page 2-60.
- Profile name – Starts a pool on the cluster specified by the profile. For more information on cluster profiles, see "Discover Clusters and Use Cluster Profiles" on page 5-11.
- Cluster object – Starts a pool on the cluster specified by the cluster object. Use `parcluster` to get a cluster object.

Example: `parpool('local')`

Example: `parpool('threads')`

Example: `parpool('myClusterProfile',16)`

Example: `c = parcluster; parpool(c)`

Data Types: `char` | `string` | `parallel.Cluster`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AttachedFiles',{'myFun.m'}`

`AttachedFiles` — Files to attach to pool
character vector | string scalar | string array | cell array of character vectors

Files to attach to pool, specified as a character vector, string or string array, or cell array of character vectors.

With this argument pair, `parpool` starts a parallel pool and passes the identified files to the workers in the pool. The files specified here are appended to the `AttachedFiles` property specified in the applicable parallel profile to form the complete list of attached files. The `'AttachedFiles'` property name is case sensitive, and must appear as shown.

Example: `{'myFun.m','myFun2.m'}`

Data Types: `char` | `cell`

`AutoAddClientPath` — Flag to specify if client path is added to worker path
`true` (default) | `false`

Flag to specify if user-added entries on the client path are added to path of each worker at startup, specified as a logical value.

Data Types: `logical`

`EnvironmentVariables` — Environment variables copied to workers
character vector | string scalar | string array | cell array of character vectors

Names of environment variables to copy from the client session to the workers, specified as a character vector, string or string array, or cell array of character vectors. The names specified here are appended to the `'EnvironmentVariables'` property specified in the applicable parallel profile to form the complete list of environment variables. Any variables listed which are not set are not copied to the workers. These environment variables are set on the workers for the duration of the parallel pool.

Data Types: `char` | `cell`

`SpmdEnabled` — Flag to specify if spmd is supported on pool
`true` (default) | `false`

Flag to specify if `spmd` support is enabled on the pool, specified as a logical value. You can disable support only on a local or MATLAB Job Scheduler cluster. `parfor` iterations do not involve communication between workers. Therefore, if `'SpmdEnabled'` is `false`, a `parfor`-loop continues even if one or more workers aborts during loop execution.

Data Types: `logical`

`IdleTimeout` — Time after which the pool shuts down if idle
nonnegative integer

Time in minutes after which the pool shuts down if idle, specified as an integer greater than zero. A pool is idle if it is not running code on the workers. By default `'IdleTimeout'` is the same as the

value in your parallel preferences. For more information on parallel preferences, see "Specify Your Parallel Preferences" on page 5-9.

Example: `pool = parpool('IdleTimeout',120)`

Output Arguments

`poolobj` — Access to parallel pool from client
parallel.Pool object

Access to parallel pool from client, returned as a parallel.Pool object.

Tips

- The pool status indicator in the lower-left corner of the desktop shows the client session connection to the pool and the pool status. Click the icon for a menu of supported pool actions.

With a pool running: With no pool running:

- If you set your parallel preferences to automatically create a parallel pool when necessary, you do not need to explicitly call the `parpool` command. You might explicitly create a pool to control when you incur the overhead time of setting it up, so the pool is ready for subsequent parallel language constructs.
- `delete(poolobj)` shuts down the parallel pool. Without a parallel pool, `spmd` and `parfor` run as a single thread in the client, unless your parallel preferences are set to automatically start a parallel pool for them.
- When you use the MATLAB editor to update files on the client that are attached to a parallel pool, those updates automatically propagate to the workers in the pool. (This automatic updating does not apply to Simulink model files. To propagate updated model files to the workers, use the `updateAttachedFiles` function.)
- If possible, the working folder on the workers is initially set to match that of the MATLAB client session. Subsequently, the following commands entered in the client Command Window also execute on all the workers in the pool:

  - `cd`
  - `addpath`
  - `rmpath`

  This behavior allows you to set the working folder and the command search path on all the workers, so that subsequent pool activities such as `parfor`-loops execute in the proper context.

  When changing folders or adding a path with `cd` or `addpath` on clients with Windows operating systems, the value sent to the workers is the UNC path for the folder if possible. For clients with Linux operating systems, it is the absolute folder location.

  If any of these commands does not work on the client, it is not executed on the workers either. For example, if `addpath` specifies a folder that the client cannot access, the `addpath` command is not executed on the workers. However, if the working folder can be set on the client, but cannot be set as specified on any of the workers, you do not get an error message returned to the client Command Window.

Be careful of this slight difference in behavior in a mixed-platform environment where the client is not the same platform as the workers, where folders local to or mapped from the client are not available in the same way to the workers, or where folders are in a nonshared file system. For example, if you have a MATLAB client running on a Microsoft Windows operating system while the MATLAB workers are all running on Linux operating systems, the same argument to `addpath` cannot work on both. In this situation, you can use the function `pctRunOnAll` to assure that a command runs on all the workers.

Another difference between client and workers is that any `addpath` arguments that are part of the `matlabroot` folder are not set on the workers. The assumption is that the MATLAB install base is already included in the workers' paths. The rules for `addpath` regarding workers in the pool are:

- Subfolders of the `matlabroot` folder are not sent to the workers.

- Any folders that appear before the first occurrence of a `matlabroot` folder are added to the top of the path on the workers.

- Any folders that appear after the first occurrence of a `matlabroot` folder are added after the `matlabroot` group of folders on the workers' paths.

For example, suppose that `matlabroot` on the client is `C:\Applications\matlab\`. With an open parallel pool, execute the following to set the path on the client and all workers:

```
addpath('P1',
        'P2',
        'C:\Applications\matlab\T3',
        'C:\Applications\matlab\T4',
        'P5',
        'C:\Applications\matlab\T6',
        'P7',
        'P8');
```

Because `T3`, `T4`, and `T6` are subfolders of `matlabroot`, they are not set on the workers' paths. So on the workers, the pertinent part of the path resulting from this command is:

```
P1
P2
<worker original matlabroot folders...>
P5
P7
P8
```

- If you are using Macintosh or Linux, and see problems during large parallel pool creation, see "Recommended System Limits for Macintosh and Linux" on page 2-70.

See Also
`Composite` | `delete` | `distributed` | `gcp` | `parallel.defaultClusterProfile` | `parallel.pool.Constant` | `parcluster` | `parfeval` | `parfevalOnAll` | `parfor` | `pctRunOnAll` | `spmd`

Topics
"Specify Your Parallel Preferences" on page 5-9
"Discover Clusters and Use Cluster Profiles" on page 5-11
"Pass Data to and from Worker Sessions" on page 6-14

Introduced in R2013b

pause

Pause MATLAB Job Scheduler queue

Syntax

```
pause(mjs)
```

Arguments

mjs   MATLAB Job Scheduler object whose queue is paused.

Description

pause(mjs) pauses the MATLAB Job Scheduler's queue so that jobs waiting in the queued state will not run. Jobs that are already running also pause, after completion of tasks that are already running. No further jobs or tasks will run until the resume function is called for the MATLAB Job Scheduler.

The pause function does nothing if the MATLAB Job Scheduler is already paused.

See Also
resume | wait

Topics
"Program Independent Jobs for a Supported Scheduler" on page 6-7

Introduced before R2006a

pctconfig

Configure settings for Parallel Computing Toolbox client session

Syntax

```
pctconfig('p1',v1,...)
config = pctconfig('p1',v1,...)
config = pctconfig()
```

Arguments

p1	Property to configure. Supported properties are `'portrange'`, `'hostname'`.
v1	Value for corresponding property.
config	Structure of configuration value.

Description

`pctconfig('p1',v1,...)` sets the client configuration property *p1* with the value v1.

Note that the property value pairs can be in any format supported by the `set` function, i.e., param-value character vector pairs, structures, and param-value cell array pairs. If a structure is used, the structure field names are the property names and the field values specify the property values.

If the property is `'portrange'`, the specified value is used to set the range of ports to be used by the client session of Parallel Computing Toolbox software. This is useful in environments with a limited choice of ports. The value of `'portrange'` should either be a 2-element vector `[minport, maxport]` specifying the range, or `0` to specify that the client session should use ephemeral ports. By default, the client session searches for available ports to communicate with the other sessions of MATLAB Parallel Server software.

If the property is `'hostname'`, the specified value is used to set the hostname for the client session of Parallel Computing Toolbox software. This is useful when the client computer is known by more than one hostname. The value you should use is the hostname by which the cluster nodes can contact the client computer. The toolbox supports both short hostnames and fully qualified domain names.

`config = pctconfig('p1',v1,...)` returns a structure to `config`. The field names of the structure reflect the property names, while the field values are set to the property values.

`config = pctconfig()`, without any input arguments, returns all the current values as a structure to `config`. If you have not set any values, these are the defaults.

Examples

View the current settings for hostname and ports.

```
config = pctconfig()
```

```
config =
    portrange: [27370 27470]
     hostname: 'machine32'
```

Set the current client session port range to `21000-22000` with hostname `fdm4`.

```
pctconfig('hostname','fdm4','portrange',[21000 22000]);
```

Set the client hostname to a fully qualified domain name.

```
pctconfig('hostname','desktop24.subnet6.companydomain.com');
```

Tips

The values set by this function do not persist between MATLAB sessions. To guarantee its effect, call `pctconfig` before calling any other Parallel Computing Toolbox functions.

Introduced in R2008a

pctRunDeployedCleanup

Clean up after deployed parallel applications

Syntax

```
pctRunDeployedCleanup
```

Description

pctRunDeployedCleanup performs necessary cleanup so that the client JVM can properly terminate when the deployed application exits. All deployed applications that use Parallel Computing Toolbox functionality need to call pctRunDeployedCleanup after the last call to Parallel Computing Toolbox functionality.

After calling pctRunDeployedCleanup, you should not use any further Parallel Computing Toolbox functionality in the current MATLAB session.

Introduced in R2010a

pctRunOnAll

Run command on client and all workers in parallel pool

Syntax

```
pctRunOnAll command
```

Description

`pctRunOnAll command` runs the specified `command` on all the workers of the parallel pool as well as the client, and prints any command-line output back to the client Command Window. The specified `command` runs in the base workspace of the workers and does not have any return variables. This is useful if there are setup changes that need to be performed on all the workers and the client.

Note If you use `pctRunOnAll` to run a command such as `addpath` in a mixed-platform environment, it can generate a warning on the client while executing properly on the workers. For example, if your workers are all running on Linux operating systems and your client is running on a Microsoft Windows operating system, an `addpath` argument with Linux-based paths will warn on the Windows-based client.

Examples

Clear all loaded functions on all workers:

```
pctRunOnAll clear functions
```

Change the directory on all workers to the project directory:

```
pctRunOnAll cd /opt/projects/c1456
```

Add some directories to the paths of all the workers:

```
pctRunOnAll addpath({'/usr/share/path1' '/usr/share/path2'})
```

See Also
parpool

Introduced in R2008a

pload

(To be removed) Load file into parallel session

Note `pload` will be removed in a future release. Use `dload` instead. For more information, see "pload and psave will be removed".

Syntax

```
pload(fileroot)
```

Arguments

fileroot   Part of filename common to all saved files being loaded.

Description

`pload(fileroot)` loads the data from the files named `[fileroot num2str(labindex)]` into the workers running a communicating job. The files should have been created by the `psave` command. The number of workers should be the same as the number of files. The files should be accessible to all the workers. Any codistributed arrays are reconstructed by this function. If `fileroot` contains an extension, the character representation of the `labindex` will be inserted before the extension. Thus, `pload('abc')` attempts to load the file `abc1.mat` on worker 1, `abc2.mat` on worker 2, and so on.

Examples

Create three variables — one replicated, one variant, and one codistributed. Then save the data. (This example works in a communicating job or in pmode, but not in a `parfor` or `spmd` block.)

```
clear all;
rep = speye(numlabs);
var = magic(labindex);
D = eye(numlabs,codistributor());
psave('threeThings');
```

This creates three files (`threeThings1.mat`, `threeThings2.mat`, `threeThings3.mat`) in the current working directory.

Clear the workspace on all the workers and confirm there are no variables.

```
clear all
whos
```

Load the previously saved data into the workers. Confirm its presence.

```
pload('threeThings');
whos
isreplicated(rep)
iscodistributed(D)
```

Compatibility Considerations

pload and psave will be removed
Warns starting in R2020a

In a future release, the psave and pload functions will be removed. To save and load data on the workers, in the form of Composite arrays or distributed arrays, use dsave and dload instead.

See Also

dload | labindex | load | numlabs | pmode | psave | save

Introduced in R2006b

pmode

(To be removed) Interactive Parallel Command Window

Note pmode will be removed in a future release. Use spmd instead. For more information, see "pmode will be removed".

Syntax

```
pmode start
pmode start numworkers
pmode start prof numworkers
pmode quit
pmode exit
pmode client2lab clientvar workers workervar
pmode lab2client workervar worker clientvar
pmode cleanup prof
```

Description

pmode allows the interactive parallel execution of MATLAB commands. pmode achieves this by defining and submitting a communicating job, and opening a Parallel Command Window connected to the workers running the job. The workers then receive commands entered in the Parallel Command Window, process them, and send the command output back to the Parallel Command Window. Variables can be transferred between the MATLAB client and the workers.

pmode start starts pmode, using the default profile to define the cluster and number of workers. (The initial default profile is local; you can change it by using the function parallel.defaultClusterProfile.) You can also specify the number of workers using pmode start numworkers.

pmode start prof numworkers starts pmode using the Parallel Computing Toolbox profile prof to locate the cluster, submits a communicating job with the number of workers identified by numworkers, and connects the Parallel Command Window with the workers. If the number of workers is specified, it overrides the minimum and maximum number of workers specified in the profile.

pmode quit or pmode exit stops the pmode job, deletes it, and closes the Parallel Command Window. You can enter this command at the MATLAB prompt or the pmode prompt.

pmode client2lab clientvar workers workervar copies the variable clientvar from the MATLAB client to the variable workervar on the workers identified by workers. If workervar is omitted, the copy is named clientvar. workers can be either a single index or a vector of indices. You can enter this command at the MATLAB prompt or the pmode prompt.

pmode lab2client workervar worker clientvar copies the variable workervar from the worker identified by worker, to the variable clientvar on the MATLAB client. If clientvar is omitted, the copy is named workervar. You can enter this command at the MATLAB prompt or the pmode prompt. Note: If you use this command in an attempt to transfer a codistributed array to the

client, you get a warning, and only the local portion of the array on the specified worker is transferred. To transfer an entire codistributed array, first use the `gather` function to assemble the whole array into the worker workspaces.

`pmode cleanup prof` deletes all communicating jobs created by `pmode` for the current user running on the cluster specified in the profile `prof`, including jobs that are currently running. The profile is optional; the default profile is used if none is specified. You can enter this command at the MATLAB prompt or the `pmode` prompt.

You can invoke `pmode` as either a command or a function, so the following are equivalent.

```
pmode start prof 4
pmode('start','prof',4)
```

Examples

In the following examples, the `pmode` prompt (P>>) indicates commands entered in the Parallel Command Window. Other commands are entered in the MATLAB Command Window.

Start `pmode` using the default profile to identify the cluster and number of workers.

```
pmode start
```

Start `pmode` using the `local` profile with four local workers.

```
pmode start local 4
```

Start `pmode` using the profile `myProfile` and eight workers on the cluster.

```
pmode start myProfile 8
```

Execute a command on all workers.

```
P>> x = 2*labindex;
```

Copy the variable `x` from worker 7 to the MATLAB client.

```
pmode lab2client x 7
```

Copy the variable `y` from the MATLAB client to workers 1 through 8.

```
pmode client2lab y 1:8
```

Display the current working directory of each worker.

```
P>> pwd
```

Compatibility Considerations

pmode will be removed
Warns starting in R2020a

In a future release, the `pmode` function will be removed. To execute commands interactively on multiple workers, use `spmd` instead.

See Also

createCommunicatingJob | parallel.defaultClusterProfile | parcluster | spmd

Introduced in R2006b

poll

Retrieve data sent from a worker

Syntax

```
poll(pollablequeue)
[data, OK] = poll(pollablequeue, timeout)
```

Description

`poll(pollablequeue)` retrieves the result of a message or data sent from a worker to the `parallel.pool.PollableDataQueue` specified by `pollablequeue`. You can use poll only in the process where you created the data queue.

`[data, OK] = poll(pollablequeue, timeout)` returns `data`, and `OK` as a boolean true to indicate that data has been returned. If there is no data in the queue, then an empty array is returned with a boolean false for `OK`. Specify `timeout` in seconds as an optional second parameter. In that case, the method might block for the time specified by `timeout` before returning. If any data arrives in the queue during that period, that data is returned.

Examples

Send a Message in a `parfor`-loop, and Poll for the Result

Construct a `PollableDataQueue`.

```
p = parallel.pool.PollableDataQueue;
```

Start a `parfor`-loop, and send a message, such as data with the value 1.

```
parfor i = 1
    send(p, i);
end
```

Poll for the result.

```
poll(p)
```

```
1
```

For more details on sending data using a `PollableDataQueue`, see `send`.

Send and Poll for Data while Using `parfeval`

This example shows how to return intermediate results from a worker to the client and to display the result on the client.

Construct a `PollableDataQueue`. A `PollableDataQueue` is most useful for sending and polling for data during asynchronous function evaluations using `parfeval` or `parfevalOnAll`.

```
q = parallel.pool.PollableDataQueue;
```

Start a timer and send the data queue as input to the function for `parfeval` execution on the pool. Display the time elapsed and the data returned.

```matlab
f = parfeval(@workerFcn, 0, q);
msgsReceived = 0;
starttime = tic;
while msgsReceived < 2
    [data, gotMsg] = poll(q, 1);
    if gotMsg
        fprintf('Got message: %s after %.3g seconds\n', ...
            data, toc(starttime));
        msgsReceived = msgsReceived + 1;
    else
        fprintf('No message available at %.3g seconds\n', ...
            toc(starttime));
    end
end

function workerFcn(q)
    send(q, 'start');
    pause(3);
    send(q, 'stop');
end
```

```
Got message: start after 0.39 seconds
No message available at 1.48 seconds
No message available at 2.56 seconds
Got message: stop after 3.35 seconds
```

The first message is returned in 0.39 s after you have executed `parfeval`. In that time the data and function for `parfeval` have been serialized, sent over to the workers, deserialized and set running. When you start the code, the worker sends some data, which is serialized, sent over the network back to the client and put on a data queue. `poll` notes this operation and returns the value to the client function. Then the time taken since `parfeval` was called is displayed. Note a delay of 3 s while the worker is computing something (in this case a long pause).

Input Arguments

pollablequeue — Pollable data queue
parallel.pool.PollableDataQueue

Pollable data queue, specified as a `parallel.pool.PollableDataQueue` object.

Example: `[data, OK] = poll(pollablequeue, optionalTimeout);`

timeout — Optional timeout
scalar

Optional timeout interval (in seconds) used to block `poll` before returning, specified as a scalar.

Example: `[data, OK] = poll(pollablequeue, timeout);`

Output Arguments

data — Message or data
scalar | vector | matrix | array | string | character vector

Message or data from workers to a data queue, specified as any serializable value.

Example: `[data, OK] = poll(pollablequeue, timeout);`

OK — Check if data has been returned
Boolean

Check if data has been returned, returned as a Boolean value. If data has been returned, then `OK` is assigned the value of a boolean `true`. If there is no data in the queue `pollablequeue`, then an empty array is returned and a boolean `false` for `OK`.

Example: `[data, OK] = poll(pollablequeue, timeout);`

See Also
DataQueue | afterEach | parallel.pool.PollableDataQueue | parfeval | parfevalOnAll | parfor | parpool | send

Introduced in R2017a

poolStartup

File for user-defined options to run on each worker when parallel pool starts

Syntax

```
poolStartup
```

Description

`poolStartup` runs automatically on a worker each time the worker forms part of a parallel pool. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the `poolStartup.m` file to define pool initialization on the worker. The worker looks for `poolStartup.m` in the following order, executing the one it finds first:

1   Included in the job's `AttachedFiles` property.
2   In a folder included in the job's `AdditionalPaths` property.
3   In the worker's MATLAB installation at the location

    matlabroot/toolbox/parallel/user/poolStartup.m

To create a version of `poolStartup.m` for `AttachedFiles` or `AdditionalPaths`, copy the provided file and modify it as required. .

`poolStartup` is the ideal location for startup code required for parallel execution on the parallel pool. For example, you might want to include code for using `mpiSettings`. Because `jobStartup` and `taskStartup` execute before `poolStartup`, they are not suited to pool-specific code. In other words, you should use `taskStartup` for setup code on your worker regardless of whether the task is from an independent job, communicating job, or using a parallel pool; while `poolStartup` is for setup code for pool usage only.

For further details on `poolStartup` and its implementation, see the text in the installed `poolStartup.m` file.

See Also
`jobStartup` | `taskFinish` | `taskStartup`

Introduced in R2010a

promote

Promote job in MATLAB Job Scheduler cluster queue

Syntax

```
promote(c,job)
```

Arguments

c	The MATLAB Job Scheduler cluster object that contains the job.
job	Job object promoted in the queue.

Description

`promote(c,job)` promotes the job object `job`, that is queued in the MATLAB Job Scheduler cluster c.

If `job` is not the first job in the queue, `promote` exchanges the position of `job` and the previous job.

Examples

Create and submit multiple jobs to the cluster identified by the default cluster profile, assuming that the default cluster profile uses a MATLAB Job Scheduler:

```
c = parcluster();
pause(c) % Prevent submissions from running.

j1 = createJob(c,'Name','Job A');
j2 = createJob(c,'Name','Job B');
j3 = createJob(c,'Name','Job C');
submit(j1);submit(j2);submit(j3);
```

Promote `Job C` by one position in its queue:

```
promote(c,j3)
```

Examine the new queue sequence:

```
[pjobs,qjobs,rjobs,fjobs] = findJob(c);
get(qjobs,'Name')

    'Job A'
    'Job C'
    'Job B'
```

Tips

After a call to `promote` or `demote`, there is no change in the order of job objects contained in the `Jobs` property of the MATLAB Job Scheduler cluster object. To see the scheduled order of execution

for jobs in the queue, use the `findJob` function in the form `[pending queued running finished] = findJob(c)`.

See Also

createJob | demote | findJob | submit

Introduced before R2006a

psave

(To be removed) Save data from communicating job session

Note psave will be removed in a future release. Use dsave instead. For more information, see "pload and psave will be removed".

Syntax

psave(fileroot)

Arguments

fileroot	Part of filename common to all saved files.

Description

psave(fileroot) saves the data from the workers' workspace into the files named [fileroot num2str(labindex)]. The files can be loaded by using the pload command with the same fileroot, which should point to a folder accessible to all the workers. If fileroot contains an extension, the character representation of the labindex is inserted before the extension. Thus, psave('abc') creates the files 'abc1.mat', 'abc2.mat', etc., one for each worker.

Examples

Create three arrays — one replicated, one variant, and one codistributed. Then save the data. (This example works in a communicating job or in pmode, but not in a parfor or spmd block.)

```
clear all;
rep = speye(numlabs);
var = magic(labindex);
D = eye(numlabs,codistributor());
psave('threeThings');
```

This creates three files (threeThings1.mat, threeThings2.mat, threeThings3.mat) in the current working folder.

Clear the workspace on all the workers and confirm there are no variables.

```
clear all
whos
```

Load the previously saved data into the workers. Confirm its presence.

```
pload('threeThings');
whos
isreplicated(rep)
iscodistributed(D)
```

Compatibility Considerations

`pload` and `psave` will be removed
Warns starting in R2020a

In a future release, the `psave` and `pload` functions will be removed. To save and load data on the workers, in the form of Composite arrays or distributed arrays, use `dsave` and `dload` instead.

See Also

dsave | labindex | load | numlabs | pload | pmode | save

Introduced in R2006b

rand

Array of rand values

Syntax

```
R = rand(sz,arraytype)
R = rand(sz,datatype,arraytype)

R = rand(sz,'like',P)
R = rand(sz,datatype,'like',P)

C = rand(sz,codist)
C = rand(sz,datatype,codist)
C = rand(sz, ___ ,codist,'noCommunication')
C = rand(sz, ___ ,codist,'like',P)
```

Description

`R = rand(sz,arraytype)` creates a matrix with underlying class of double, with `rand` values in all elements.

`R = rand(sz,datatype,arraytype)` creates a matrix with underlying class of *datatype*, with `rand` values in all elements.

The size and type of array are specified by the argument options according to the following table.

Argument	Values	Descriptions
sz	n	Specifies size as an n-by-n matrix.
	m,n or [m n]	Specifies size as an m-by-n matrix.
	m,n,...,k or [m n ... k]	Specifies size as an m-by-n-by-...-by-k array.
arraytype	'distributed'	Specifies distributed array.
	'codistributed'	Specifies codistributed array, using the default distribution scheme.
	'gpuArray'	Specifies gpuArray.
datatype	'double' (default), 'single'	Specifies underlying class of the array, i.e., the data type of its elements.

`R = rand(sz,'like',P)` creates an array of `rand` values with the same type and underlying class (data type) as array P.

`R = rand(sz,datatype,'like',P)` creates an array of `rand` values with the specified underlying class (*datatype*), and the same type as array P.

`C = rand(sz,codist)` or `C = rand(sz,datatype,codist)` creates a codistributed array of `rand` values with the specified size and underlying class (the default *datatype* is 'double'). The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For

information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = rand(8,codistributor1d());
end
```

`C = rand(sz, ___ ,codist,'noCommunication')` specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

`C = rand(sz, ___ ,codist,'like',P)` creates a codistributed array of `rand` values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array `P`.

Examples

Create Distributed Rand Matrix

Create a 1000-by-1000 distributed array of `rand`s with underlying class double:

```
D = rand(1000,'distributed');
```

Create Codistributed Rand Matrix

Create a 1000-by-1000 codistributed double matrix of `rand`s, distributed by its second dimension (columns).

```
spmd(4)
    C = rand(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of `C`.

Create a 1000-by-1000 codistributed `single` matrix of `rand`s, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = rand(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-`labindex` local piece of `C`.

Create gpuArray Rand Matrix

Create a 1000-by-1000 gpuArray of `rand`s with underlying class `double`:

```
G = rand(1000,'double','gpuArray');
```

See Also
`codistributed.sprand` | `distributed.sprand` | `rand` | `randi` | `randn`

Topics
"Control Random Number Streams on Workers" on page 5-29
"Random Number Streams on a GPU" on page 8-6

Introduced in R2006b

randi

Array of random integers

Syntax

```
R = randi(imax,n)
R = randi(imax,size1,...,sizeN)
R = randi(imax,size)

R = randi(imax,n,arraytype)
R = randi(imax,size1,...,sizeN,arraytype)
R = randi(imax,size,arraytype)

R = randi(imax,n,datatype,arraytype)
R = randi(imax,size1,...,sizeN,datatype,arraytype)
R = randi(imax,size,datatype,arraytype)

R = randi(imax,n,'like',P)
R = randi(imax,size1,...,sizeN,'like',P)
R = randi(imax,size,'like',P)

R = randi(imax,n,datatype,'like',P)
R = randi(imax,size1,...,sizeN,datatype,'like',P)
R = randi(imax,size,datatype,'like',P)

C = randi(imax,n,codist)
C = randi(imax,size1,...,sizeN,codist)
C = randi(imax,size,codist)

C = randi(imax,n,datatype,codist)
C = randi(imax,size1,...,sizeN,datatype,codist)
C = randi(imax,size,datatype,codist)

C = randi( ___ ,codist,'noCommunication')
C = randi( ___ ,codist,'like',P)

C = randi([imin imax], ___ )
```

Description

R = randi(imax,n) returns an *n*-by-*n* distributed matrix with random integer values in the range [1,imax].

R = randi(imax,size1,...,sizeN) returns a *size1*-by-...-by-*sizeN* distributed matrix with random integer values in the range [1,imax]. *size1*-by-...-by-*sizeN* are separated arguments that indicates the size of each dimension of the generated matrix.

R = randi(imax,size) returns an array which size is defined by the size vector *size*.

R = randi(imax,n,arraytype) returns an *n*-by-*n* matrix with underlying class of double, random integer values in the range [1,imax], and the type specified by arraytype.

R = randi(imax,size1,...,sizeN,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of double, random integer values in the range [1,imax], and the type specified by arraytype.

R = randi(imax,size,arraytype) returns an array which size is defined by the size vector *size* with underlying class of double, random integer values in the range [1,imax], and the type specified by arraytype.

R = randi(imax,n,datatype,arraytype) returns an *n*-by-*n* matrix with underlying class of datatype, random integer values in the range [1,imax], and the type specified by arraytype.

R = randi(imax,size1,...,sizeN,datatype,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of datatype, random integer values in the range [1,imax], and the type specified by arraytype.

R = randi(imax,size,datatype,arraytype) returns an array which size is defined by the size vector *size* with underlying class of datatype, random integer values in the range [1,imax], and the type specified by arraytype.

R = randi(imax,n,'like',P) returns an *n*-by-*n* array with random integer values in the range [1,imax] and the same type and underlying class (data type) as the prototype array, P.

R = randi(imax,size1,...,sizeN,'like',P) returns a *size1*-by-...-by-*sizeN* array with random integer values in the range [1,imax] and the same type and underlying class (data type) as the prototype array, P.

R = randi(imax,size,'like',P) returns an array which size is defined by the size vector *size* with random integer values in the range [1,imax] and the same type and underlying class (data type) as the prototype array, P.

R = randi(imax,n,datatype,'like',P) returns an *n*-by-*n* array with random integer values in the range [1,imax], the specified underlying class (datatype), and the same type as the prototype array, P.

R = randi(imax,size1,...,sizeN,datatype,'like',P) returns a *size1*-by-...-by-*sizeN* array with random integer values in the range [1,imax], the specified underlying class (datatype), and the same type as the prototype array, P.

R = randi(imax,size,datatype,'like',P) returns an array which size is defined by the size vector *size* with random integer values in the range [1,imax], the specified underlying class (datatype), and the same type as the prototype array, P.

C = randi(imax,n,codist) returns an *n*-by-*n* codistributed array with random integer values in the range [1,imax] and underlying class of double. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = randi(imax,size1,...,sizeN,codist) returns a *size1*-by-...-by-*sizeN* codistributed array with random integer values in the range [1,imax] and underlying class of double.

C = randi(imax,size,codist) returns a codistributed array which size is defined by the size vector *size* with random integer values in the range [1,imax] and underlying class of double.

C = randi(imax,n,datatype,codist) returns an *n*-by-*n* codistributed array with random integer values in the range [1,imax] and underlying class of datatype. The codistributor object

codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = randi(imax,size1,...,sizeN,datatype,codist) returns a *size1*-by-...-by-*sizeN* codistributed array with random integer values in the range [1,imax] and underlying class of datatype.

C = randi(imax,size,datatype,codist) returns a codistributed array which size is defined by the size vector *size* with random integer values in the range [1,imax] and underlying class of datatype.

C = randi(___ ,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = randi(___ ,codist,'like',P) creates a codistributed array of random integer values with the specified range, size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array, P.

C = randi([imin imax], ___) returns an array of random integers values in the range [imin,imax], using any of the above syntaxes.

Examples

Create Distributed Randi Matrix

Create a 1000-by-1000 distributed array of randi values from 1 to 100, with underlying class double.

```
D = randi(100,1000,'distributed');
```

Create Codistributed Randi Matrix

Create a 1000-by-1000 codistributed double matrix of randi values from 0 to 12, distributed by its second dimension (columns).

```
spmd(4)
    C = randi([0 12],1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed single matrix of randi values from 1 to 4, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = randi(4,1000,1000,'single',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray Rand Matrix

Create a 1000-by-1000 gpuArray of `randi` values from -50 to 50, with underlying class `double`:

```
G = randi([-50 50],1000,'double','gpuArray');
```

Input Arguments

`imax` — Maximum integer in range
integer value

Maximum integer in the range, specified as an integer value.

`imin` — Minimum integer in range
integer value

Minimum integer in the range, specified as an integer value.

`n` — Size of array
integer value

Size of the generated array, specified as an integer value.

`size1,...,sizeN` — Size of each dimension (as separate arguments)
integer values

Size of each dimension of the generated array, specified as separate arguments of two or more integer values.

`size` — Size of each dimension (as a row vector)
vector

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

`arraytype` — Array type
`'distributed'` | `'codistributed'` | `'gpuArray'`

Type of the generated array, specified as either `'distributed'`, `'codistributed'`, or `'gpuArray'`.

`datatype` — Array data type
`'double'` (default) | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'int64'` | `'uint64'`

Underlying class of the array, that is the data type of its elements, specified as one of these options:

- `'double'`
- `'single'`
- `'int8'`
- `'uint8'`
- `'int16'`
- `'uint16'`

- `'int32'`
- `'uint32'`
- `'int64'`
- `'uint64'`

codist — Codistributor
codistributor object

Distribution scheme for creating the codistributed array, specified as a codistributor object. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. To use the default distribution scheme, you can specify a codistributor constructor without arguments.

P — Prototype array
array

Prototype of array to create, specified as an array.

Output Arguments

R — Array
distributed array | codistributed array | gpuArray

Array of random integers, returned as either a distributed array, a codistributed array, or a gpuArray.

C — Codistributed array
codistributed array

Array of random integers, returned as a codistributed array.

See Also
rand | randi | randn

Topics
"Control Random Number Streams on Workers" on page 5-29
"Random Number Streams on a GPU" on page 8-6

Introduced in R2014a

randn

Array of randn values

Syntax

```
R = randn(n,arraytype)
R = randn(size1,...,sizeN,arraytype)
R = randn(size,arraytype)

R = randn(n,datatype,arraytype)
R = randn(size1,...,sizeN,datatype,arraytype)
R = randn(size,datatype,arraytype)

R = randn(n,'like',P)
R = randn(size1,...,sizeN,'like',P)
R = randn(size,'like',P)

R = randn(n,datatype,'like',P)
R = randn(size1,...,sizeN,datatype,'like',P)
R = randn(size,datatype,'like',P)

C = randn(n,codist)
C = randn(size1,...,sizeN,codist)
C = randn(size,codist)

C = randn(n,datatype,codist)
C = randn(size1,...,sizeN,datatype,codist)
C = randn(size,datatype,codist)

C = randn( ___ ,codist,'noCommunication')
C = randn( ___ ,codist,'like',P)
```

Description

R = randn(n,arraytype) returns an *n*-by-*n* matrix with underlying class of double, randn values in all elements, and the type specified by arraytype.

R = randn(size1,...,sizeN,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of double, randn values in all elements, and the type specified by arraytype.

R = randn(size,arraytype) returns an array which size is defined by the size vector *size* with underlying class of double, randn values in all elements, and the type specified by arraytype.

R = randn(n,datatype,arraytype) returns an *n*-by-*n* matrix with underlying class of datatype, randn values in all elements, and the type specified by arraytype.

R = randn(size1,...,sizeN,datatype,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of datatype, randn values in all elements, and the type specified by arraytype.

`R = randn(size,datatype,arraytype)` returns an array which size is defined by the size vector *size* with underlying class of `datatype`, `randn` values in all elements, and the type specified by `arraytype`.

`R = randn(n,'like',P)` returns an *n*-by-*n* array with `randn` values in all elements and the same type and underlying class (data type) as the prototype array, P.

`R = randn(size1,...,sizeN,'like',P)` returns a *size1*-by-...-by-*sizeN* array with `randn` values in all elements and the same type and underlying class (data type) as the prototype array, P.

`R = randn(size,'like',P)` returns an array which size is defined by the size vector *size* with `randn` values in all elements and the same type and underlying class (data type) as the prototype array, P.

`R = randn(n,datatype,'like',P)` returns an *n*-by-*n* array with `randn` values in all elements, the specified underlying class (`datatype`), and the same type as the prototype array, P.

`R = randn(size1,...,sizeN,datatype,'like',P)` returns a *size1*-by-...-by-*sizeN* array with `randn` values in all elements, the specified underlying class (`datatype`), and the same type as the prototype array, P.

`R = randn(size,datatype,'like',P)` returns an array which size is defined by the size vector *size* with `randn` values in all elements, the specified underlying class (`datatype`), and the same type as the prototype array, P.

`C = randn(n,codist)` returns an *n*-by-*n* codistributed array with `randn` values in all elements and underlying class of double. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = randn(size1,...,sizeN,codist)` returns a *size1*-by-...-by-*sizeN* codistributed array with `randn` values in all elements and underlying class of double.

`C = randn(size,codist)` returns a codistributed array which size is defined by the size vector *size* with `randn` values in all elements and underlying class of double.

`C = randn(n,datatype,codist)` returns an *n*-by-*n* codistributed array with `randn` values in all elements and underlying class of `datatype`. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = randn(size1,...,sizeN,datatype,codist)` returns a *size1*-by-...-by-*sizeN* codistributed array with `randn` values in all elements and underlying class of `datatype`.

`C = randn(size,datatype,codist)` returns a codistributed array which size is defined by the size vector *size* with `randn` values in all elements and underlying class of `datatype`.

`C = randn( ___ ,codist,'noCommunication')` specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

`C = randn( ___ ,codist,'like',P)` creates a codistributed array of random integer values with the specified range, size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array, P.

Examples

Create Distributed Randn Matrix

Create a 1000-by-1000 distributed array of `randn` values from 1 to 100, with underlying class double.

```
D = randn(1000,'distributed');
```

Create Codistributed Randn Matrix

Create a 1000-by-1000 codistributed double matrix of `randn` values, distributed by its second dimension (columns).

```
spmd(4)
    C = randn(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of `C`.

Create a 1000-by-1000 codistributed `single` matrix of `randn` values, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = randn(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-`labindex` local piece of `C`.

Create gpuArray Rand Matrix

Create a 1000-by-1000 gpuArray of `randn` values with underlying class `double`:

```
G = randn(1000,'double','gpuArray');
```

Input Arguments

n — Size of array
scalar

Size of the generated array, specified as a scalar.

size1,...,sizeN — Size of each dimension (as separate arguments)
scalars

Size of each dimension of the generated array, specified as separate arguments of two or more scalars.

size — Size of each dimension (as a row vector)
vector

Size of each dimension, specified as a row vector of scalars. Each element of this vector indicates the size of the corresponding dimension.

arraytype — Array type
'distributed' | 'codistributed' | 'gpuArray'

Type of the generated array, specified as either 'distributed', 'codistributed', or 'gpuArray'.

datatype — Array data type
'double' (default) |

Underlying class of the array, that is the data type of its elements, specified as either 'double' or 'single'.

codist — Codistributor
codistributor object

Distribution scheme for creating the codistributed array, specified as a codistributor object. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments.

P — Prototype array
array

Prototype of array to create, specified as an array.

Output Arguments

R — Array
distributed array | codistributed array | gpuArray

Array of random integers, returned as either a distributed array, a codistributed array, or a gpuArray.

C — Codistributed array
codistributed array

Array of random integers, returned as a codistributed array.

See Also
codistributed.sprandn | distributed.sprandn | rand | randi | randn

Topics
"Control Random Number Streams on Workers" on page 5-29
"Random Number Streams on a GPU" on page 8-6

Introduced in R2006b

recreate

Create new job from existing job

Syntax

```
newjob = recreate(oldjob)
newjob = recreate(oldjob,'Tasks',tasksToRecreate)
newjob = recreate(oldjob,'TaskState',states)
newjob = recreate(oldjob,'TaskID',ids)
```

Description

`newjob = recreate(oldjob)` creates a new job object based on an existing job, containing the same tasks and options as `oldjob`. The old job can be in any state; the new job state is pending. If `oldjob` was created using `batch`, then MATLAB automatically submits the new job.

`newjob = recreate(oldjob,'Tasks',tasksToRecreate)` creates a job object with tasks that correspond to `tasksToRecreate`. Because communicating jobs have only one task, this option only supports independent jobs.

`newjob = recreate(oldjob,'TaskState',states)` creates a job object with tasks that correspond to the tasks with `State` specified by `states`. Because communicating jobs have only one task, this option only supports independent jobs.

`newjob = recreate(oldjob,'TaskID',ids)` creates a job object containing the tasks from `oldjob` that correspond to the tasks with IDs specified by `ids`. Because communicating jobs have only one task, this option only supports independent jobs.

Examples

Use `recreate` to Resubmit Tasks with Errors

This approach is useful when tasks depend on a file that is not present anymore.

Create a new job using the default cluster profile. In this example, it is the local parallel pool.

```
cluster = parcluster;
job = createJob(cluster);
```

Create several tasks. In particular, create a task that depends on a MAT-file that does not exist.

```
createTask(job,@() 'Task1',1);
createTask(job,@() load('myData.mat'),1);
```

Submit the job, and wait for it to finish. Because the MAT-file in the second task does not exist, the job fails. If you call `fetchOutputs` on `job` to retrieve the results, you get an error. Check the error using the `Error` property of the corresponding task.

```
submit(job);
wait(job);
job.Tasks(2).Error

ans =
  ParallelException with properties:

     identifier: 'MATLAB:load:couldNotReadFile'
        message: 'Unable to read file 'myData.mat'. No such file or directory.'
          cause: {}
    remotecause: {[1×1 MException]}
          stack: [1×1 struct]
```

Create the MAT-file referenced from the second task using the `save` function. To create a new job with the tasks that resulted in an error, use the `'Tasks'` name-value pair in `recreate`, and provide the `hasError` function. If you want to select a different set of tasks, you can define your own function.

```
str = 'Task2';
save myData str
newjob = recreate(job,'Tasks',@hasError);
```

Submit the new job, wait for its completion, and fetch the outputs. Because the MAT-file now exists, the job does not fail.

```
submit(newjob);
wait(newjob);
out = fetchOutputs(newjob);
out{1}

ans = struct with fields:
    str: 'Task2'
```

Recreate an Entire Job

This example shows how to recreate the entire job `myJob`.

```
newJob = recreate(myJob)
```

Recreate a Job with Only Pending Tasks

This example shows how to recreate an independent job, which has only pending tasks from the job `oldIndependentJob`.

```
newJob = recreate(oldIndependentJob,'TaskState','pending');
```

Recreate a Job with Specified Tasks

This example shows how to recreate an independent job, which has only the tasks with IDs 21 to 32 from the job `oldIndependentJob`.

```
newJob = recreate(oldIndependentJob,'TaskID',[21:32]);
```

Recreate Jobs of a Specific User

This example shows how to find and recreate all failed jobs submitted by user Mary. Assume the default cluster is the one Mary had submitted her jobs to.

```
c = parcluster();
failedjobs = findJob(c,'Username','Mary','State','failed');
for m = 1:length(failedjobs)
    newJob(m) = recreate(failedjobs(m));
end
```

Input Arguments

oldjob — Original job
parallel.Job

Original job to be duplicated, specified as a `parallel.Job` object.

Example: `newJob = recreate(oldjob); submit(newJob);`

Data Types: `parallel.Job`

tasksToRecreate — Tasks to duplicate
`parallel.Task` array | logical array | function handle

Tasks to duplicate from `oldjob`, specified as:

- An array of `parallel.Task` belonging to `oldjob`.
- A `1 x N` logical array, where `N` is the size of `oldjob.Tasks`, indicating the tasks in `oldjob` to be recreated.
- A function handle that accepts `oldjob.Tasks` as an input argument. This function must return a `1 x N` logical array indicating the tasks in `oldjob` to be recreated, where `N` is the size of `oldjob.Tasks`.

To rerun tasks containing errors or warnings, use this syntax with the predefined functions `@hasError` and `hasWarnings`.

Example: `newJob = recreate(oldjob,'Tasks',@hasError | @hasWarnings);`

Data Types: `parallel.Task` | `logical` | `function_handle`

states — State of the tasks to duplicate
`'pending'` | `'running'` | `'finished'` | `'failed'` | cell array with any of the valid states

State of the tasks to duplicate, specified as a string or cell array of strings. `states` represents the state of the required tasks to recreate from `oldjob`. Valid states are `'pending'`, `'running'`, `'finished'`, and `'failed'`.

Example: `newJob = recreate(oldJob,'TaskState','failed');`

Data Types: `char` | `string` | `cell`

ids — IDs of the tasks to duplicate
vector of integers

IDs of the tasks to duplicate from `oldjob`, specified as a vector of integers.

Example: `newJob = recreate(oldIndependentJob,'TaskID',[1 5]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`createCommunicatingJob` | `createJob` | `createTask` | `findJob` | `submit`

Introduced in R2014a

redistribute

Redistribute codistributed array with another distribution scheme

Syntax

```
D2 = redistribute(D1,codist)
```

Description

`D2 = redistribute(D1,codist)` redistributes a codistributed array `D1` and returns `D2` using the distribution scheme defined by the `codistributor` object `codist`.

Examples

Redistribute an array according to the distribution scheme of another array.

```
spmd
  % First, create a magic square distributed by columns:
    M = codistributed(magic(10),codistributor1d(2,[1 2 3 4]));

  % Create a pascal matrix distributed by rows (first dimension):
    P = codistributed(pascal(10),codistributor1d(1));

  % Redistribute the pascal matrix according to the
  % distribution (partition) scheme of the magic square:
    R = redistribute(P,getCodistributor(M));
end
```

See Also
codistributed | codistributor | codistributor1d.defaultPartition

Introduced in R2006b

reset

Package: `parallel.gpu`

Reset GPU device and clear its memory

Syntax

```
reset(gpudev)
```

Description

`reset(gpudev)` resets the GPU device and clears its memory of gpuArray and CUDAKernel data. The GPU device identified by `gpudev` remains the selected device, but all gpuArray and CUDAKernel objects in MATLAB representing data on that device are invalid.

Examples

Reset GPU Device

Create a gpuArray on the selected GPU device.

```
g = gpuDevice(1);
M = gpuArray(magic(4));
M  % Display gpuArray

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Reset the device.

```
reset(g);
g   % Show that the device is still selected

g =

  CUDADevice with properties:

                      Name: 'Tesla K20c'
                     Index: 1
         ComputeCapability: '3.5'
            SupportsDouble: 1
             DriverVersion: 7.5
            ToolkitVersion: 7.5
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
              MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 5.0327e+09
```

```
        AvailableMemory: 4.9190e+09
     MultiprocessorCount: 13
            ClockRateKHz: 705500
             ComputeMode: 'Default'
    GPUOverlapsTransfers: 1
  KernelExecutionTimeout: 0
        CanMapHostMemory: 1
         DeviceSupported: 1
          DeviceSelected: 1
```

Show that the gpuArray variable name is still in the MATLAB workspace

```
whos
```

```
  Name       Size       Bytes  Class
  g          1x1          112  parallel.gpu.CUDADevice
  M          1x1          108  gpuArray
```

Try to display the gpuArray.

```
M
```

```
Data no longer exists on the GPU.
```

Clear the variable.

```
clear M
```

Input Arguments

gpudev — GPU device
GPU device

GPU device, specified as a GPUDevice object that represents the currently selected device

Tips

After you reset a GPU device, any variables representing arrays or kernels on the device are invalid; you should clear or redefine them.

See Also
gpuArray | gpuDevice | parallel.gpu.CUDAKernel

Introduced in R2012a

resume

Resume processing queue in MATLAB Job Scheduler

Syntax

```
resume(mjs)
```

Arguments

mjs   MATLAB Job Scheduler object whose queue is resumed.

Description

`resume(mjs)` resumes processing of the specified MATLAB Job Scheduler's queue so that jobs waiting in the queued state will be run. This call will do nothing if the MATLAB Job Scheduler is not paused.

See Also
pause | wait

Introduced before R2006a

saveAsProfile

Save cluster properties to specified profile

Description

saveAsProfile(cluster,profileName) saves the properties of the cluster object to the specified profile, and updates the cluster Profile property value to indicate the new profile name.

Examples

Create a cluster, then modify a property and save the properties to a new profile.

```
myCluster = parcluster('local');
myCluster.NumWorkers = 3;
saveAsProfile(myCluster,'local2'); New profile now specifies 3 workers
```

See Also
parallel.defaultClusterProfile | parcluster | saveProfile

Introduced in R2012a

saveProfile

Save modified cluster properties to its current profile

Description

saveProfile(cluster) saves the modified properties on the cluster object to the profile specified by the cluster's Profile property, and sets the Modified property to false. If the cluster's Profile property is empty, an error is thrown.

Examples

Create a cluster, then modify a property and save the change to the profile.

```
myCluster = parcluster('local')

myCluster =
 Local Cluster
    Properties:
                            Profile: local
                           Modified: false
                               Host: HOSTNAME
                         NumWorkers: 4

myCluster.NumWorkers = 3

myCluster =
 Local Cluster
    Properties:
                            Profile: local
                           Modified: true
                               Host: HOSTNAME
                         NumWorkers: 3
```

The myCluster.Modified property is now true.

```
saveProfile(myCluster);
myCluster

myCluster =
 Local Cluster
    Properties:
                            Profile: local
                           Modified: false
                               Host: HOSTNAME
                         NumWorkers: 3
```

After saving, the local profile now matches the current property settings, so the myCluster.Modified property is false.

See Also
parallel.defaultClusterProfile | parcluster | saveAsProfile

Introduced in R2012a

setConstantMemory

Set some constant memory on GPU

Syntax

```
setConstantMemory(kern,sym,val)
setConstantMemory(kern,sym1,val1,sym2,val2,...)
```

Description

`setConstantMemory(kern,sym,val)` sets the constant memory in the CUDA kernel `kern` with symbol name `sym` to contain the data in `val`. `val` can be any numeric array, including a gpuArray. The command errors if the named symbol does not exist or if it is not big enough to contain the specified data. Partially filling a constant is allowed.

There is no automatic data-type conversion for constant memory, so it is important to make sure that the supplied data is of the correct type for the constant memory symbol being filled.

`setConstantMemory(kern,sym1,val1,sym2,val2,...)` sets multiple constant symbols.

Examples

If KERN represents a CUDA kernel whose CU file contains the following includes and constant definitions:

```
#include "tmwtypes.h"
__constant__ int32_t N1;
__constant__ int N2; // Assume 'int' is 32 bits
__constant__ double CONST_DATA[256];
```

you can fill these with MATLAB data as follows:

```
KERN = parallel.gpu.CUDAKernel(ptxFile,cudaFile);

setConstantMemory(KERN,'N1',int32(10));
setConstantMemory(KERN,'N2',int32(10));
setConstantMemory(KERN,'CONST_DATA',1:10);
```

or

```
setConstantMemory(KERN,'N1',int32(10),'N2',int32(10),'CONST_DATA',1:10);
```

See Also
`gpuArray` | `parallel.gpu.CUDAKernel`

Introduced in R2012a

setJobClusterData

Set specific user data for job on generic cluster

Syntax

setJobClusterData(cluster,job,userdata)

Arguments

cluster	Cluster object identifying the generic third-party cluster running the job
job	Job object identifying the job for which to store data
userdata	Information to store for this job

Description

setJobClusterData(cluster,job,userdata) stores data for the job job that is running on the generic cluster cluster. You can later retrieve the information with the function getJobClusterData. For example, it might be useful to store the third-party scheduler's external ID for this job, so that the function specified in GetJobStateFcn can later query the scheduler about the state of the job. Or the stored data might be an array with the scheduler's ID for each task in the job.

For more information and examples on using these functions and properties, see "Plugin Scripts for Generic Schedulers" on page 6-17.

See Also
getJobClusterData

Introduced in R2012a

shutdown

Shut down cloud cluster

Syntax

```
shutdown(cluster)
shutdown(MJScluster,'At',D)
shutdown(MJScluster,'After',event)
shutdown(MJScluster,'After',numhours)
```

Description

`shutdown(cluster)` shuts down the cluster immediately.

`shutdown(MJScluster,'At',D)` shuts down the cluster at the time specified by the `datenum`, `datevec`, or `datetime` D. D is interpreted in the local time zone unless D is a `datetime` with `TimeZone` specified.

`shutdown(MJScluster,'After',event)` shuts down the cluster after the specified event `event` has occurred. `event` can be `'never'` or `'idle'`. A cluster is `'idle'` immediately when there are no running jobs, queued jobs, or running pools. The cluster is eligible for shutdown if `'idle'` for more than 5 minutes, and is guaranteed to shut down within 60 minutes.

`shutdown(MJScluster,'After',numhours)` shuts down the cluster after `numhours` hours, as measured from the time the method is called.

Examples

Shut Down a Cloud Cluster Immediately

```
shutdown(cluster);
```

Shut Down Cluster at Date and Time Specified

Specify date and time to terminate a cluster using a `datenum`, `datevec`, or `datetime`. The `datetime` can (optionally) have a `TimeZone` specified.

```
shutdown(MJSCluster,'At',datenum('2017-02-22 19:00'));
shutdown(MJSCluster,'At',datevec('2017-02-22 19:00'));
shutdown(MJSCluster,'At',datetime(2017, 2, 22, 19, 0, 0,'TimeZone','local');
```

Enable Cluster to Run Indefinitely

```
shutdown(MJSCluster,'After','never');
```

Shut Down When Cluster Is Idle

```
shutdown(MJSCluster,'After','idle');
```

Shut Down Cluster After a Number of Hours

```
shutdown(MJSCluster,'After',10);
```

Input Arguments

`cluster` — MATLAB Parallel Server for Amazon EC2 cloud cluster
cluster object (default)

MATLAB Parallel Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`.

Example: `shutdown(cluster);`

`MJScluster` — MATLAB Parallel Server for Amazon EC2 cloud cluster
cluster object (default)

MATLAB Parallel Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`.

Example: `shutdown(MJScluster);`

`D` — Date and time
datenum | datevec | datetime

Date and time, specified as a `datenum`, `datevec`, or `datetime`. `D` is interpreted in the local time zone unless `D` is a `datetime` with `TimeZone` specified.

Example: `shutdown(MJSCluster,'At',datenum('2017-02-22 19:00'));`

Example: `shutdown(MJSCluster,'At',datevec('2017-02-22 19:00'));`

Example: `shutdown(MJSCluster,'At',datetime(2017, 2, 22, 19, 0, 0,'TimeZone', 'local'));`

`event` — Event to shut down the cluster
'never' | 'idle'

Event to shut down the cluster, specified as `'never'` or `'idle'`. A cluster is `'idle'` immediately when there are no running jobs, queued jobs, or running pools. The cluster is eligible for shutdown if `'idle'` for more than 5 minutes, and is guaranteed to shut down within 60 minutes.

Example: `shutdown(MJSCluster,'After','idle');`

`numhours` — Number of hours
scalar

Number of hours after which the cluster shuts down, specified as scalar, measured from the time you call `shutdown`.

Example: `shutdown(MJSCluster,'After',10);`

See Also

datenum | datetime | datevec | parallel.Cluster | parcluster | parpool | start | wait
(cluster)

Introduced in R2017a

sparse

Create sparse distributed or codistributed matrix

Syntax

```
SD = sparse(FD)
SC = sparse(m,n,codist)
SC = sparse(m,n,codist,'noCommunication')
SC = sparse(i,j,v,m,n,nzmax)
SC = sparse(i,j,v,m,n)
SC = sparse(i,j,v)
```

Description

`SD = sparse(FD)` converts a full distributed or codistributed array `FD` to a sparse distributed or codistributed (respectively) array `SD`.

`SC = sparse(m,n,codist)` creates an m-by-n sparse codistributed array of underlying class double, distributed according to the scheme defined by the codistributor `codist`. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. This form of the syntax is most useful inside `spmd` or a communicating job.

`SC = sparse(m,n,codist,'noCommunication')` creates an m-by-n sparse codistributed array in the manner specified above, but does not perform any global communication for error checking when constructing the array. This form of the syntax is most useful inside `spmd` or a communicating job.

`SC = sparse(i,j,v,m,n,nzmax)` uses vectors `i` and `j` to specify indices, and `v` to specify element values, for generating an m-by-n sparse matrix such that `SC(i(k),j(k)) = v(k)`, with space allocated for `nzmax` nonzeros. If any of the input vectors `i`, `j`, or `v` is codistributed, the output sparse matrix `SC` is codistributed. Vectors `i`, `j`, and `v` must be the same length. Any elements of `v` that are zero are ignored, along with the corresponding values of `i` and `j`. Any elements of `v` that have duplicate values of `i` and `j` are added together.

To simplify this six-argument call, you can pass scalars for the argument `v` and one of the arguments `i` or `j`, in which case they are expanded so that `i`, `j`, and `v` all have the same length.

`SC = sparse(i,j,v,m,n)` uses `nzmax = max([length(i) length(j)])` .

`SC = sparse(i,j,v)` uses `m = max(i)` and `n = max(j)`. The maxima are computed before any zeros in `v` are removed, so one of the rows of `[i j v]` might be `[m n 0]`, assuring the matrix size satisfies the requirements of `m` and `n`.

Note To create a sparse codistributed array of underlying class logical, first create an array of underlying class double and then cast it using the `logical` function:

```
spmd
    SC = logical(sparse(m,n,codistributor1d()));
end
```

Examples

With four workers,

```
spmd(4)
    C = sparse(1000,1000,codistributor1d())
end
```

creates a 1000-by-1000 codistributed sparse double array C. C is distributed by its second dimension (columns), and each worker contains a 1000-by-250 local piece of C.

```
spmd(4)
    codist = codistributor1d(2,1:numlabs)
    C = sparse(10,10,codist);
end
```

creates a 10-by-10 codistributed sparse double array C, distributed by its columns. Each worker contains a 10-by-labindex local piece of C.

Convert a distributed array into a sparse distributed array:

```
R = rand(1000,'distributed');
D = floor(2*R); % D also is distributed
SD = sparse(D); % SD is sparse distributed
```

Create a sparse codistributed array from vectors of indices and a distributed array of element values:

```
r = [ 1  1  4  4 8];
c = [ 1  4  1  4 8];
v = [10 20 30 40 0];
V = distributed(v);
spmd
    SC = sparse(r,c,V);
end
```

In this example, even though the fifth element of the value array v is 0, the size of the result is an 8–by-8 matrix because of the corresponding maximum indices in r and c. Matrix SC is considered codistributed when viewed inside an spmd block, and distributed when viewed from the client workspace. To view a full version of the matrix, the full function converts this distributed sparse array to a full distributed array:

```
S = full(SC)

    10     0     0    20     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
    30     0     0    40     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

See Also

codistributed.spalloc | distributed.spalloc | sparse

Introduced in R2006b

spmd

Execute code in parallel on workers of parallel pool

Syntax

```
spmd
    statements
end
```

Description

spmd, *statements*, end defines an spmd statement on a single line. MATLAB executes the spmd body denoted by statements on several MATLAB workers simultaneously. Each worker can operate on a different data set or different portion of distributed data, and can communicate with other participating workers while performing the parallel computations. The spmd statement can be used only if you have Parallel Computing Toolbox. To execute the statements in parallel, you must first create a pool of MATLAB workers using parpool or have your parallel preferences allow the automatic start of a pool.

Inside the body of the spmd statement, each MATLAB worker has a unique value of labindex, while numlabs denotes the total number of workers executing the block in parallel. Within the body of the spmd statement, communication functions for communicating jobs (such as labSend and labReceive) can transfer data between the workers.

Values returning from the body of an spmd statement are converted to Composite objects on the MATLAB client. A Composite object contains references to the values stored on the remote MATLAB workers, and those values can be retrieved using cell-array indexing. The actual data on the workers remains available on the workers for subsequent spmd execution, so long as the Composite exists on the client and the parallel pool remains open.

By default, MATLAB uses all workers in the pool. When there is no pool active, MATLAB will create a pool and use all the workers from that pool. If your preferences do not allow automatic pool creation, MATLAB executes the block body locally and creates Composite objects as necessary. You cannot execute an spmd block if any worker is busy executing a parfeval request, unless you use spmd(0).

For more information about spmd and Composite objects, see "Distribute Arrays and Run SPMD" on page 1-12.

Note Use parfevalOnAll instead of parfor or spmd if you want to use clear. This preserves workspace transparency. See "Ensure Transparency in parfor-Loops or spmd Statements" on page 2-49.

---

spmd(*n*), *statements*, end uses n to specify the exact number of MATLAB workers to evaluate statements, provided that n workers are available from the parallel pool. If there are not enough workers available, an error is thrown. If n is zero, MATLAB executes the block body locally and creates Composite objects, the same as if there is no pool available.

spmd(*m*,*n*), statements, end uses a minimum of m and a maximum of n workers to evaluate statements. If there are not enough workers available, an error is thrown. m can be zero, which allows the block to run locally if no workers are available.

Examples

Execute Code in Parallel with spmd

Create a parallel pool, and perform a simple calculation in parallel using spmd. MATLAB executes the code inside the spmd on all workers in the parallel pool.
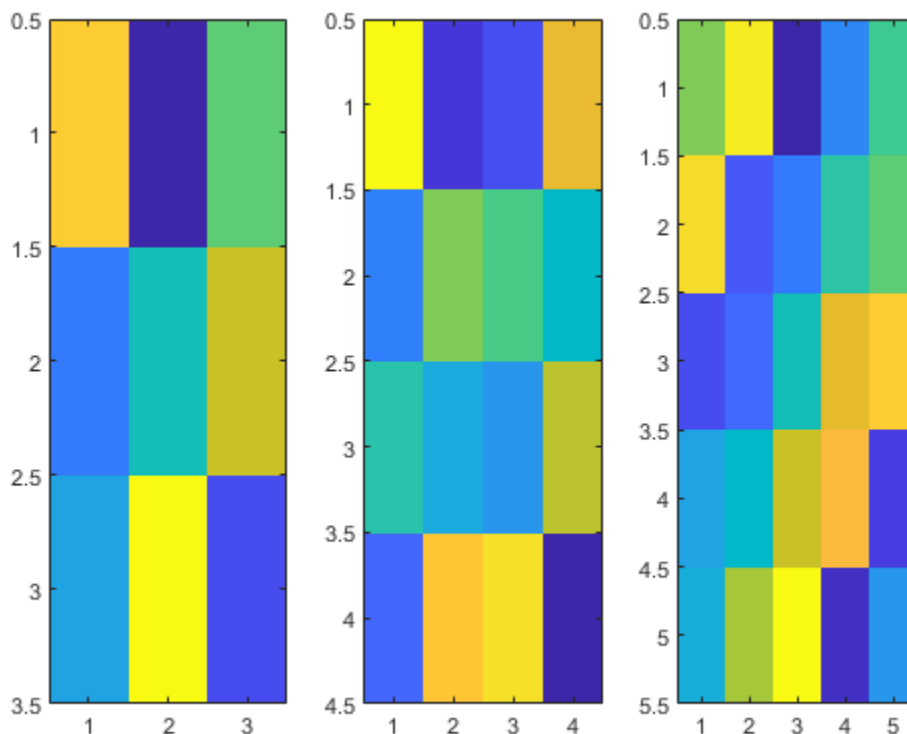
```
parpool(3);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 3).
```

```
spmd
  q = magic(labindex + 2);
end
```

Plot the results.

```
figure
subplot(1,3,1), imagesc(q{1});
subplot(1,3,2), imagesc(q{2});
subplot(1,3,3), imagesc(q{3});
```

When you are done with computations, you can delete the current parallel pool.

```
delete(gcp);
```

Use Multiple GPUs in a Parallel Pool

If you have access to several GPUs, you can perform your calculations on multiple GPUs in parallel using a parallel pool.

Start a parallel pool with as many workers as GPUs. To determine the number of GPUs available, use the `gpuDeviceCount` function. By default, MATLAB assigns a different GPU to each worker for best performance.

```
parpool('local',gpuDeviceCount);
```

To identify which GPU each worker is using, call `gpuDevice` inside an `spmd` block. The `spmd` block runs `gpuDevice` on every worker.

```
spmd
    gpuDevice
end
```

Use parallel language features, such as `parfor` or `parfeval`, to distribute your computations to workers in the parallel pool. If you use `gpuArray` enabled functions in your computations, these functions run on the GPU of the worker. For more information, see "Run MATLAB Functions on a GPU" on page 8-9. For an example, see "Run MATLAB Functions on Multiple GPUs" on page 9-82.

When you are done with your computations, shut down the parallel pool. You can use the `gcp` function to obtain the current parallel pool.

```
delete(gcp('nocreate'));
```

If you want to use a different choice of GPUs, then can use `gpuDevice` to select a particular GPU on each worker. Define an array, for example `gpuIndices`, that contains the indices of the GPUs to activate on each worker. Then, start a parallel pool with as many workers as GPUs to select, and use an `spmd` block to run `gpuDevice` on each worker. The `labindex` function identifies each worker. Use this function to associate a worker with a GPU index.

```
gpuIndices = [1 3];
parpool(numel(gpuIndices));
spmd
    gpuDevice(gpuIndices(labindex));
end
```

As a best practice, and for best performance, assign a different GPU to each worker.

Tips

- An `spmd` block runs on the workers of the existing parallel pool. If no pool exists, `spmd` will start a new parallel pool, unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `spmd` cannot start one, the code runs serially in the client session.

- If the `AutoAttachFiles` property in the cluster profile for the parallel pool is set to `true`, MATLAB performs an analysis on an `spmd` block to determine what code files are necessary for its execution, then automatically attaches those files to the parallel pool job so that the code is available to the workers.

- For information about restrictions and limitations when using `spmd`, see "Run Single Programs on Multiple Data Sets" on page 3-2.

- For information about the performance of `spmd` and other parallel programming constructs, see "Choose Between spmd, parfor, and parfeval" on page 3-15.

See Also

Composite | batch | gop | labindex | numlabs | parallel.pool.Constant | parpool

Introduced in R2008b

start

Start cloud cluster

Syntax

```
start(cluster)
```

Description

start(cluster) starts the specified MATLAB Parallel Server for Amazon EC2 cluster, if it is not already running. If the cluster is already running or in the process of shutting down, then start(cluster) returns immediately, and the state of the cluster is not changed.

Examples

Start Cloud Cluster

Obtain your cluster profile using one of the following ways:

- From the MATLAB **Parallel > Discover Clusters** user interface. For more information, see "Discover Clusters and Use Cluster Profiles" on page 5-11.
- By downloading it from Cloud Center. For more information, see MathWorks Cloud Center documentation.

Create a cluster using the default profile.

```
myCluster = parcluster;
```

Start the cluster.

```
start(myCluster);
```

Wait for the cluster to be ready to accept job submissions.

```
wait(myCluster);
```

Input Arguments

cluster — MATLAB Parallel Server for Amazon EC2 cluster
cluster object (default)

MATLAB Parallel Server for Amazon EC2 cluster, specified as cluster object created using parcluster.

Example: start(cluster)

See Also
parallel.Cluster | parcluster | parpool | shutdown | wait (cluster)

Introduced in R2017a

submit

Queue job in scheduler

Syntax

```
submit(j)
```

Description

submit(j) queues the job object j in its cluster queue. The cluster used for this job was determined when the job was created.

Examples

Create and Submit Job

Create a cluster object from a cluster profile.

```
c1 = parcluster('Profile1');
```

Create a job object in this cluster.

```
j1 = createJob(c1);
```

Add a task object to be evaluated for the job.

```
t1 = createTask(j1,@rand,1,{8,4});
```

Queue the job object in the cluster for execution.

```
submit(j1);
```

Input Arguments

j — Job to queue
job object

Job to queue, specified as a job object. To create a job object, use the createJob function.

Tips

When a job is submitted to a cluster queue, the job's State property is set to queued, and the job is added to the list of jobs waiting to be executed.

The jobs in the waiting list are executed in a first in, first out manner; that is, the order in which they were submitted, except when the sequence is altered by promote, demote, cancel, or delete.

See Also
createCommunicatingJob | createJob | findJob | parcluster | promote | recreate

Introduced before R2006a

subsasgn

Subscripted assignment for Composite

Syntax

```
C(i) = {B}
C(1:end) = {B}
C([i1,i2]) = {B1,B2}
C{i} = B
```

Description

`subsasgn` assigns remote values to Composite objects. The values reside on the workers in the current parallel pool.

`C(i) = {B}` sets the entry of C on worker `i` to the value B.

`C(1:end) = {B}` sets all entries of C to the value B.

`C([i1,i2]) = {B1,B2}` assigns different values on workers `i1` and `i2`.

`C{i} = B` sets the entry of C on worker `i` to the value B.

See Also

Composite | subsasgn | subsref

Introduced in R2008b

subsref

Subscripted reference for Composite

Syntax

```
B = C(i)
B = C([i1,i2,...])
B = C{i}
[B1,B2,...] = C{[i1,i2,...]}
```

Description

`subsref` retrieves remote values of a Composite object from the workers in the current parallel pool.

`B = C(i)` returns the entry of Composite `C` from worker `i` as a cell array.

`B = C([i1,i2,...])` returns multiple entries as a cell array.

`B = C{i}` returns the value of Composite `C` from worker `i` as a single entry.

`[B1,B2,...] = C{[i1,i2,...]}` returns multiple entries.

See Also
`Composite` | `subsasgn` | `subsref`

Introduced in R2008b

taskFinish

User-defined options to run on worker when task finishes

Syntax

```
taskFinish(task)
```

Arguments

task	The task being evaluated by the worker

Description

`taskFinish(task)` runs automatically on a worker each time the worker finishes evaluating a task for a particular job. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the `taskFinish.m` file to define anything you want executed on the worker when a task is finished. The worker looks for `taskFinish.m` in the following order, executing the one it finds first:

1   Included in the job's `AttachedFiles` property.
2   In a folder included in the job's `AdditionalPaths` property.
3   In the worker's MATLAB installation at the location

   matlabroot/toolbox/parallel/user/taskFinish.m

To create a version of `taskFinish.m` for `AttachedFiles` or `AdditionalPaths`, copy the provided file and modify it as required. For further details on `taskFinish` and its implementation, see the text in the installed `taskFinish.m` file.

See Also
jobStartup | poolStartup | taskStartup

Introduced before R2006a

taskStartup

User-defined options to run on worker when task starts

Syntax

taskStartup(task)

Arguments

task	The task being evaluated by the worker.

Description

taskStartup(task) runs automatically on a worker each time the worker evaluates a task for a particular job. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the taskStartup.m file to define task initialization on the worker. The worker looks for taskStartup.m in the following order, executing the one it finds first:

1 Included in the job's AttachedFiles property.
2 In a folder included in the job's AdditionalPaths property.
3 In the worker's MATLAB installation at the location

   matlabroot/toolbox/parallel/user/taskStartup.m

To create a version of taskStartup.m for AttachedFiles or AdditionalPaths, copy the provided file and modify it as required. For further details on taskStartup and its implementation, see the text in the installed taskStartup.m file.

See Also
jobStartup | poolStartup | taskFinish

Introduced before R2006a

send

Package: parallel.pool

Send data from worker to client using a data queue

Syntax

```
send(queue, data)
send(pollablequeue, data)
```

Description

send(queue, data) sends a message or data with the value data to the parallel.pool.DataQueue specified by queue. Call afterEach to pass each of the pending messages to the function specified by afterEach.

send(pollablequeue, data) sends a message or data with the value data to the parallel.pool.PollableDataQueue specified by pollablequeue. Retrieve the result using poll(pollablequeue), and return data as the answer.

Use the send and poll functions together using a pollable data queue to transfer and retrieve messages or data from different workers.

Examples

Send a Message in a parfor-Loop, and Dispatch the Message on the Queue

Construct a DataQueue, and call afterEach.

```
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```

Start a parfor-loop, and send a message. The pending message is passed to the afterEach function, in this example @disp.

```
parfor i = 1:3
    send(q, i);
end;

    1

    2

    3
```

For more details on listening for data using a DataQueue, see afterEach.

Send a Message in a `parfor`-loop, and Poll for the Result

Construct a `PollableDataQueue`.

```
p = parallel.pool.PollableDataQueue;
```

Start a `parfor`-loop, and send a message, such as data with the value 1.

```
parfor i = 1
    send(p, i);
end
```

Poll for the result.

```
poll(p)

    1
```

For more details on retrieving data using a `PollableDataQueue`, see `poll`.

Construct a Simple `parfor` Wait Bar Using a Data Queue

This example shows a function that creates a `parfor` wait bar. Create a `DataQueue`, and use `afterEach` to specify the function to execute each time the queue receives data. This example calls a subfunction that updates the wait bar.

Create a `parfor`-loop to carry out a computationally demanding task in MATLAB. Use `send` to send some dummy data on each iteration of the `parfor`-loop. When the queue receives the data, `afterEach` calls `nUpdateWaitbar` in the client MATLAB, and you can observe the wait bar progress.

```
function a = parforWaitbar

D = parallel.pool.DataQueue;
h = waitbar(0, 'Please wait ...');
afterEach(D, @nUpdateWaitbar);

N = 200;
p = 1;

parfor i = 1:N
    a(i) = max(abs(eig(rand(400))));
    send(D, i);
end

    function nUpdateWaitbar(~)
        waitbar(p/N, h);
        p = p + 1;
    end
end
```
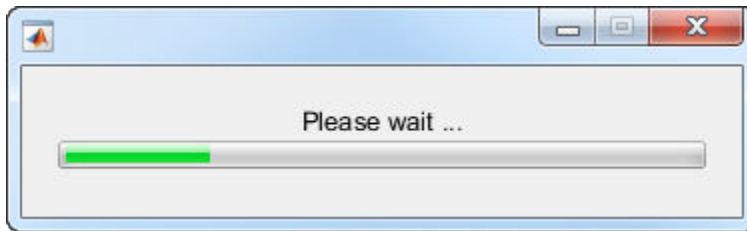
Input Arguments

queue — Data queue
parallel.pool.DataQueue

Data queue, specified as a `parallel.pool.DataQueue` object.

Example: q = parallel.pool.DataQueue;

data — Message or data
scalar | vector | matrix | array | string | character vector | serializable object

Message or data from workers to a data queue, specified as any data type that can be serialized.

Example: send(queue, data);

pollablequeue — Pollable data queue
parallel.pool.PollableDataQueue

Pollable data queue, specified as a `parallel.pool.PollableDataQueue` object.

Example: p = parallel.pool.PollableDataQueue;

See Also
DataQueue | afterEach | parallel.pool.PollableDataQueue | parfor | parpool | poll

Introduced in R2017a

ticBytes

Start counting bytes transferred within parallel pool

Syntax

```
ticBytes(pool)
startState = ticBytes(pool)
```

Description

ticBytes(pool) starts counting the number of bytes transferred to each worker in the pool, so that later tocBytes(pool) can measure the amount of data transferred to each worker between the two calls.

Use the ticBytes (pool) and tocBytes (pool) functions together to measure how much data is transferred to and from the workers in a parallel pool. You can use ticBytes and tocBytes while executing parallel language constructs and functions, such as parfor, spmd, or parfeval. Use ticBytes and tocBytes to pass around less data and optimize your code.

startState = ticBytes(pool) saves the state to an output argument, startState, so that you can simultaneously record the number of bytes transferred for multiple pairs of ticBytes and tocBytes calls. Use the value of startState as an input argument for a subsequent call to tocBytes.

Examples

Measure Amount of Data Transferred While Running a Simple parfor-loop

```
a = 0;
b = rand(100);
ticBytes(gcp);
parfor i = 1:100
    a = a + sum(b(:, i));
end
tocBytes(gcp)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

	BytesSentToWorkers	BytesReceivedFromWorkers
1	42948	7156
2	36548	7156
3	27500	4500
4	27500	4500
Total	1.345e+05	23312

Workers might transfer different numbers of bytes, because each worker might carry out different numbers of loop iterations.

Simultaneously Measure Multiple Amounts of Data Transferred,Using Two Pairs of `ticBytes` and `tocBytes` Calls

Measure the minimum and average number of bytes transferred while running a `parfor` loop nested in a `for` loop.

```matlab
REPS = 10;
minBytes = Inf;
ticBytes(gcp);  % ticBytes, pair 1

for ii=1:REPS
    a = 0;
    b = rand(100);
    startS = ticBytes(gcp)  % ticBytes, pair 2
    parfor i = 1:100
        a = a + sum(b(:, i));
    end
    bytes = tocBytes(gcp, startS)  % tocBytes, pair 2
    minBytes = min(bytes, minBytes)
end

averageBytes = tocBytes(gcp)/REPS  % tocBytes, pair 1
```

Note that nesting a `parfor`-loop in a `for`-loop can be slow due to overhead, see "Convert Nested for-Loops to parfor-Loops" on page 2-13.

Input Arguments

pool — parallel pool
parallel.Pool

Parallel pool, typically specified by `gcp`, if you want the current parallel pool. Otherwise, use `parpool` to create a new pool.

Example: `ticBytes(gcp);`

Output Arguments

startState — Starting state
TicBytesResult

Starting state returned as an input argument for a subsequent call to `tocBytes`.

Example: `startState = ticBytes(gcp);`

See Also

gcp | parfeval | parfor | parpool | spmd | tocBytes

Introduced in R2016b

tocBytes

Read how many bytes have been transferred since calling `ticBytes`

Syntax

```
tocBytes(pool)
bytes = tocBytes(pool)
tocBytes(pool,startState)
bytes = tocBytes(pool,startState)
```

Description

`tocBytes(pool)` reads how many bytes have been transferred since calling `ticBytes`. The function displays the total number of bytes transferred to and from each of the workers in a parallel `pool` after the most recent execution of `ticBytes`.

Use the `ticBytes (pool)` and `tocBytes (pool)` functions together to measure how much data is transferred to and from the workers in a parallel pool. You can use `ticBytes` and `tocBytes` while executing parallel language constructs and functions, such as `parfor`, `spmd`, or `parfeval`. Use `ticBytes` and `tocBytes` to pass around less data and optimize your code.

`bytes = tocBytes(pool)` returns the number of bytes transferred to and from each of the workers in the parallel `pool`.

`tocBytes(pool,startState)` displays the total number of bytes transferred in the parallel `pool` after the `ticBytes` command that generated `startState`.

`bytes = tocBytes(pool,startState)` returns the number of bytes transferred to and from each of the workers in the parallel `pool` after the `ticBytes` command that generated `startState`.

Examples

Measure Amount of Data Transferred While Running a Simple `parfor`-loop

Use `tocBytes(gcp,startS)` to measure the amount of data transferred.

```
a = 0;
b = rand(100);
startS = ticBytes(gcp);
parfor i = 1:100
    a = a + sum(b(:, i));
end
tocBytes(gcp,startS)

Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.

         BytesSentToWorkers    BytesReceivedFromWorkers
         _____    _____
```

```
1              42948               7156
2              36548               7156
3              27500               4500
4              27500               4500
Total     1.345e+05              23312
```

Workers might transfer different numbers of bytes, because each worker might carry out different numbers of loop iterations.

Measure Amount of Data Transferred While Running a Simple spmd Block

Use `bytes = tocBytes(gcp)` to measure the amount of data transferred.

```
ticBytes(gcp);
spmd
    rand(100);
end
bytes = tocBytes(gcp)

bytes =

        13448          1208
        13448          1208
        13448          1208
        13448          1208
```

Workers transfer the same number of bytes, because each worker carries out the same number of loop iterations.

Simultaneously Measure Multiple Amounts of Data Transferred, Using Two Pairs of ticBytes and tocBytes Calls

Measure the minimum and average number of bytes transferred while running a `parfor` loop nested in a `for` loop.

```
REPS = 10;
minBytes = Inf;
ticBytes(gcp);  % ticBytes, pair 1

for ii=1:REPS
    a = 0;
    b = rand(100);
    startS = ticBytes(gcp)  % ticBytes, pair 2
    parfor i = 1:100
        a = a + sum(b(:, i));
    end
    bytes = tocBytes(gcp, startS)  % tocBytes, pair 2
    minBytes = min(bytes, minBytes)
end

averageBytes = tocBytes(gcp)/REPS  % tocBytes, pair 1
```

Note that nesting a `parfor`-loop in a `for`-loop can be slow due to overhead, see "Convert Nested for-Loops to parfor-Loops" on page 2-13.

Input Arguments

pool — parallel pool
parallel.Pool

Parallel pool, typically specified by `gcp`, if you want the current parallel pool. Otherwise, use `parpool` to create a new pool.

Example: `tocBytes(gcp);`

startState — Starting state
TicBytesResult

Starting state returned by `ticBytes(pool)`.

Example: `startState = ticBytes(gcp);`

Output Arguments

bytes — Bytes transferred
tocBytes(pool)

Bytes transferred, returned as a matrix of size `numWorkers` x 2. This matrix contains the number of bytes transferred to and from each of the workers in the parallel pool. `bytes` returns values in bytes without headings. Use `tocBytes(pool)` without an output argument to get Sent and Received headings, worker numbers, and values in bytes in the Command Window output.

Example: `bytes = tocBytes(pool);`

See Also
gcp | parfeval | parfor | parpool | spmd | ticBytes

Introduced in R2016b

true

Array of logical 1 (true)

Syntax

```
T = true(sz,arraytype)

T = true(sz,'like',P)

C = true(sz,codist)
C = true(sz, ___ ,codist,'noCommunication')
C = true(sz, ___ ,codist,'like',P)
```

Description

`T = true(sz,arraytype)` creates a matrix with `true` values in all elements.

The size and type of array are specified by the argument options according to the following table.

Argument	Values	Descriptions
sz	n	Specifies size as an n-by-n matrix.
	m,n or [m n]	Specifies size as an m-by-n matrix.
	m,n,...,k or [m n ... k]	Specifies size as an m-by-n-by-...-by-k array.
arraytype	`'distributed'`	Specifies distributed array.
	`'codistributed'`	Specifies codistributed array, using the default distribution scheme.
	`'gpuArray'`	Specifies gpuArray.

`T = true(sz,'like',P)` creates an array of `true` values with the same type as array P.

`C = true(sz,codist)` creates a codistributed array of `true` values with the specified size. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = true(8,codistributor1d());
end
```

`C = true(sz, ___ ,codist,'noCommunication')` specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

`C = true(sz, ___ ,codist,'like',P)` creates a codistributed array of `true` values with the specified size and distribution scheme. If the codistributor argument is omitted, the distribution scheme is taken from the codistributed array P.

Examples

Create Distributed True Matrix

Create a 1000-by-1000 distributed array of `trues` with underlying class double:

```
D = true(1000,'distributed');
```

Create Codistributed True Matrix

Create a 1000-by-1000 codistributed matrix of `trues`, distributed by its second dimension (columns).

```
spmd(4)
    C = true(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of `C`.

Create a 1000-by-1000 codistributed matrix of `trues`, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = true(1000,1000,codist);
end
```

Each worker contains a 100-by-`labindex` local piece of `C`.

Create gpuArray True Matrix

Create a 1000-by-1000 gpuArray of `trues`:

```
G = true(1000,'gpuArray');
```

See Also
Inf | NaN | eye | false | ones | true | zeros

Introduced in R2006b

updateAttachedFiles

Update attached files or folders on parallel pool

Syntax

```
updateAttachedFiles(poolobj)
```

Description

`updateAttachedFiles(poolobj)` checks all the attached files of the specified parallel pool to see if they have changed, and replicates any changes to each of the workers in the pool. This checks files that were attached (by a profile or `parpool` argument) when the pool was started and those subsequently attached with the `addAttachedFiles` command.

Examples

Update Attached Files on Current Parallel Pool

Update all attached files on the current parallel pool.

```
poolobj = gcp;
updateAttachedFiles(poolobj)
```

Input Arguments

poolobj — Pool with attached files
pool object

Pool with attached files, specified as a pool object.

Example: `poolobj = gcp;`

See Also
`addAttachedFiles` | `gcp` | `listAutoAttachedFiles` | `parpool`

Topics
"Add and Modify Cluster Profiles" on page 5-14

Introduced in R2013b

wait

Package: `parallel`

Wait for job to change state

Syntax

```
wait(j)
wait(j,state)
OK = wait(j,state,timeout)
```

Arguments

j	Job object whose change in state to wait for.
state	Value of the job object's `State` property to wait for.
timeout	Maximum time to wait, in seconds.

Description

`wait(j)` blocks execution in the client session until the job identified by the object `j` reaches the `'finished'` state or fails. This occurs when all the job's tasks are finished processing on the workers.

`wait(j,state)` blocks execution in the client session until the specified job object changes state to the value of `state`. The valid states to wait for are `'queued'`, `'running'`, and `'finished'`.

If the object is currently or has already been in the specified state, a wait is not performed and execution returns immediately. For example, if you execute `wait(j,'queued')` for a job already in the `'finished'` state, the call returns immediately.

`OK = wait(j,state,timeout)` blocks execution until either the job reaches the specified `state`, or `timeout` seconds elapse, whichever happens first. `OK` is `false` if `timeout` is exceeded before `state` is reached.

Note Simulink models cannot run while a MATLAB session is blocked by `wait`. If you must run Simulink from the MATLAB client while also running jobs, you cannot use `wait`.

Examples

Submit a job to the queue, and wait for it to finish running before retrieving its results.

```
submit(j);
wait(j,'finished')
results = fetchOutputs(j)
```

Submit a batch job and wait for it to finish before retrieving its variables.

```
j = batch('myScript');
wait(j)
load(j)
```

See Also

pause | resume | wait | wait (GPUDevice)

Introduced in R2008a

wait (cluster)

Wait for cloud cluster to change state

Syntax

```
wait(cluster)
wait(cluster,state)
OK = wait(cluster,state,timeout)
```

Description

`wait(cluster)` blocks execution in the client MATLAB session until `cluster` reaches the `'online'` state. The `'online'` state indicates that the cluster is running and you can use all requested workers to run jobs.

`wait(cluster,state)` blocks execution in the client session until `cluster` changes state. For a cluster object, the valid states are:

- `'online'`: The cluster is running and you can use all requested workers to run jobs.
- `'waitingforworkers'`: The cluster is running, and you can use some but not all of the requested workers to run jobs. You can still use the cluster in this state with the workers that are available.
- `'offline'`: The cluster is not running, but you can restart using the `start()` command or via https://cloudcenter.mathworks.com. If the cluster has shared persisted storage, then any previous jobs in the queue are still present when you restart the cluster.

`OK = wait(cluster,state,timeout)` blocks execution in the client session until `cluster` changes state, or until `timeout` seconds have elapsed, whichever happens first. `OK` is `true` if `state` has been reached or a terminal state such as `'error'` occurs. `OK` is `false` in case of a timeout.

Examples

Wait Until the Cluster Is Running

In **Cluster Profile Manager**, select MATLAB Parallel Server for Amazon EC2 as your default cluster profile.

Create and start a cloud cluster using the default profile.

```
cluster = parcluster;
start(cluster);
```

Wait until the cluster is running. Use all requested workers to run jobs.

```
wait(cluster,'online');
```

Wait for Specified Time for Cluster to Start

In **Cluster Profile Manager**, select MATLAB Parallel Server for Amazon EC2 as your default cluster profile.

Create and start a cloud cluster using the default profile.

```
cluster = parcluster;
start(cluster);
```

Wait 100 seconds for the head node and all workers to start.

```
OK = wait(cluster,'online',100);
```

Input Arguments

`cluster` — MATLAB Parallel Server for Amazon EC2 cloud cluster
cluster object (default)

MATLAB Parallel Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`.

Example: `wait(cluster);`

`state` — cloud cluster state
`'online'` | `'waitingforworkers'` | `'offline'`

Cloud cluster state, specified as a cluster object, for which the valid states are `'online'`, `'waitingforworkers'`, and `'offline'`.

Example: `wait(cluster,'online');`

`timeout` — time elapsed before cloud cluster changes state
seconds

Time elapsed before cloud cluster changes state, specified in seconds.

Example: `wait(cluster,'online',100);`

Output Arguments

`OK` — check if `state` has been reached
Boolean

Check if `state` has been reached, specified as a Boolean. `OK` is `true` if `state` has been reached or a terminal state such as `'error'` occurs. `OK` is `false` in case of a timeout.

Example: `OK = wait(cluster,'waitingforworkers',10);`

See Also
`parallel.Cluster` | `parcluster` | `parpool` | `shutdown` | `start`

Introduced in R2017a

wait

Package: parallel

Wait for futures to complete

Syntax

```
wait(F)
wait(F,state)
OK = wait(F,state,timeout)
```

Description

`wait(F)` blocks execution until each element of the array of futures F has reached the `'finished'` state.

`wait(F,state)` blocks execution until each element of the array of futures F has reached the state `state`. Valid values for `state` are `'running'` or `'finished'`.

`OK = wait(F,state,timeout)` blocks execution for a maximum of `timeout` seconds. `OK` is `false` if `timeout` is exceeded before `state` is reached.

See Also

fetchNext | fetchOutputs | isequal | parfeval | parfevalOnAll

Introduced in R2013b

wait (GPUDevice)

Package: parallel.gpu

Wait for GPU calculation to complete

Syntax

wait(gpudev)

Description

wait(gpudev) blocks execution in MATLAB until the GPU device identified by the GPUDevice object gpudev completes its calculations. This can be used before calls to toc when timing GPU code that does not gather results back to the workspace. When gathering results from a GPU, MATLAB automatically waits until all GPU calculations are complete, so you do not need to explicitly call wait in that situation.

See Also
gather | gpuArray | gpuDevice | gputimeit

Topics
"Measure Performance on the GPU" on page 8-34

Introduced in R2014b

write

Write distributed data to an output location

Syntax

```
write(location,D)
write(filepattern,D)
write( ___ ,Name,Value)
```

Description

write(location,D) writes the values in the distributed array D to files in the folder location. The data is stored in an efficient binary format suitable for reading back using datastore(location). If not distributed along the first dimension, MATLAB redistributes the data before writing, so that the resulting files can be reread using datastore.

write(filepattern,D) uses the file extension from filepattern to determine the output format. filepattern must include a folder to write the files into followed by a file name that includes a wildcard *. The wildcard represents incremental numbers for generating unique file names, for example write('folder/myfile_*.csv',D).

write(___ ,Name,Value) specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify the file type with 'FileType' and a valid file type ('mat', 'seq', 'parquet', 'text', or 'spreadsheet'), or you can specify a custom write function to process the data with 'WriteFcn' and a function handle.

Examples

Write Distributed Arrays

This example shows how to write a distributed array to a file system, then read it back using a datastore.

Create a distributed array and write it to an output folder.

```
d = distributed.rand(5000,1);
location = 'hdfs://myHadoopCluster/some/output/folder';
write(location, d);
```

Recreate the distributed array from the written files.

```
ds = datastore(location);
d1 = distributed(ds);
```

Write Distributed Arrays Using File Patterns

This example shows how to write distributed arrays to different formats using a file pattern.

Create a distributed table and write it to a simple text-based format that many applications can read.

```
dt = distributed(array2table(rand(5000,3)));
location = "/tmp/CSVData/dt_*.csv";
write(location, dt);
```

Recreate the distributed table from the written files.

```
ds = datastore(location);
dt1 = distributed(ds);
```

Write and Read Back Tall and Distributed Data

You can write distributed data and read it back as tall data and vice versa.

Create a distributed timetable and write it to disk.

```
dt = distributed(array2table(rand(5000,3)));
location = "/tmp/CSVData/dt_*.csv";
write(location, dt);
```

Build a tall table from the written files.

```
 ds = datastore(location);
 tt = tall(ds);
```

Alternatively, you can read data written from tall data into distributed data. Create a tall timetable and write it to disk.

```
tt = tall(array2table(rand(5000,3)));
location = "/tmp/CSVData/dt_*.csv";
write(location, tt);
```

Read back into a distributed timetable.

```
 ds = datastore(location);
 dt = distributed(ds);
```

Write Distributed Arrays Using a Write Function

This example shows how to write distributed arrays to a file system using a custom write function.

Create a simple write function that writes out spreadsheet files.

```
function dataWriter(info, data)
       filename = info.SuggestedFilename;
       writetable(data, filename, "FileType", "spreadsheet");
end
```

Create a distributed table and write it to disk using the custom write function.

```
dt = distributed(array2table(rand(5000,3)));
location = "/tmp/MyData/tt_*.xlsx";
write(location, dt, "WriteFcn", @dataWriter);
```

Input Arguments

`location` — Folder location to write data
character vector | string

Folder location to write data, specified as a character vector or string. `location` can specify a full or relative path. The specified folder can be either of these options:

- Existing empty folder that contains no other files
- New folder that `write` creates

You can write data to local folders on your computer, folders on a shared network, or to remote locations, such as Amazon S3, Windows Azure® Storage Blob, or a Hadoop Distributed File System (HDFS). For more information about reading and writing data to remote locations, see "Work with Remote Data".

Example: `location = '../../dir/data'` specifies a relative file path.

Example: `location = 'C:\Users\MyName\Desktop\data'` specifies an absolute path to a Windows desktop folder.

Example: `location = 'file:///path/to/data'` specifies an absolute URI path to a folder.

Example: `location = 'hdfs://myHadoopCluster/some/output/folder'` specifies an HDFS URL.

Example: `location = 's3://bucketname/some/output/folder'` specifies an Amazon S3 location.

Data Types: `char` | `string`

`D` — Input array
distributed array

Input array, specified as a distributed array.

`filepattern` — File naming pattern
string | character vector

File naming pattern, specified as a string or a character vector. The file naming pattern must contain a folder to write the files into followed by a file name that includes a wildcard `*`. `write` replaces the wildcard with sequential numbers to ensure unique file names.

Example: `write('folder/data_*.txt',D)` writes the distributed array D as a series of `.txt` files in `folder` with the file names `data_1.txt`, `data_2.txt`, and so on.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `write('C:\myData', D, 'FileType', 'text', 'WriteVariableNames', false)` writes the distributed array `D` to `C:\myData` as a collection of text files that do not use variable names as column headings.

General Options

FileType — Type of file
`'auto'` (default) | `'mat'` | `'parquet'` | `'seq'` | `'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and one of the allowed file types: `'auto'`, `'mat'`, `'parquet'`, `'seq'`, `'text'`, or `'spreadsheet'`.

Use the `'FileType'` name-value pair with the `location` argument to specify what type of files to write. By default, `write` attempts to automatically detect the proper file type. You do not need to specify the `'FileType'` name-value pair argument if `write` can determine the file type from an extension in the `location` or `filepattern` arguments. `write` can determine the file type from these extensions:

- `.mat` for MATLAB data files
- `.parquet` or `.parq` for Parquet files
- `.seq` for sequence files
- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsx`, `.xlsb`, `.xlsm`, `.xltx`, or `.xltm` for spreadsheet files

Example: `write('C:\myData', D, 'FileType', 'text')`

WriteFcn — Custom writing function
function handle

Custom writing function, specified as the comma-separated pair consisting of `'WriteFcn'` and a function handle. The specified function receives blocks of data from `D` and is responsible for creating the output files. You can use the `'WriteFcn'` name-value pair argument to write data in a variety of formats, even if the output format is not directly supported by `write`.

Functional Signature

The custom writing function must accept two input arguments, `info` and `data`:

`function myWriter(info, data)`

- `data` contains a block of data from `D`.
- `info` is a structure with fields that contain information about the block of data. You can use the fields to build a new file name that is globally unique within the final location. The structure fields are:

Field	Description
RequiredLocation	Fully qualified path to a temporary output folder. All output files must be written to this folder.
RequiredFilePattern	The file pattern required for output file names. This field is empty if only a folder name is specified.

Field	Description
SuggestedFilename	A fully qualified, globally unique file name that meets the location and naming requirements.
PartitionIndex	Index of the distributed array partition being written.
NumPartitions	Total number of partitions in the distributed array.
BlockIndexInPartition	Position of current data block within the partition.
IsFinalBlock	`true` if current block is the final block of the partition.

File Naming

The file name used for the output files determines the order that the files are read back in later by `datastore`. If the order of the files matters, then the best practice is to use the `SuggestedFilename` field to name the files since the suggested name guarantees the file order. If you do not use the suggested file name, the custom writing function must create globally unique, correctly ordered file names. The file names should follow the naming pattern outlined in `RequiredFilePattern`. The file names must be unique and correctly ordered between workers, even though each worker writes to its own local folder.

Arrays with Multiple Partitions

A distributed array is divided into partitions to facilitate running calculations on the array in parallel with Parallel Computing Toolbox. When writing a distributed array, each of the partitions is divided in smaller blocks.

`info` contains several fields related to partitions: `PartitionIndex`, `NumPartitions`, `BlockIndexInPartition`, and `IsFinalBlock`. These fields are useful when you are writing out a single file and appending to it, which is a common task for arrays with large partitions that have been split into many blocks. The custom writing function is called once per block, and the blocks in one partition are always written in order on one worker. However, different partitions can be written by different workers.

Example Function

A simple writing function that writes out spreadsheet files is:

```
function dataWriter(info, data)
  filename = info.SuggestedFilename;
  writetable(data, filename, 'FileType', 'spreadsheet')
end
```

To invoke `dataWriter` as the writing function for some data D, use the commands:

```
D = distributed(array2table(rand(5000,3)));
location = '/tmp/MyData/D_*.xlsx';
write(location, D, 'WriteFcn', @dataWriter);
```

For each block, the `dataWriter` function uses the suggested file name in the `info` structure and calls `writetable` to write out a spreadsheet file. The suggested file name takes into account the file naming pattern that is specified in the `location` argument.

Data Types: `function_handle`

Text or Spreadsheet Files

WriteVariableNames — Indicator for writing variable names as column headings
`true` or `1` (default) | `false` or `0`

Indicator for writing variable names as column headings, specified as the comma-separated pair consisting of `'WriteVariableNames'` and a numeric or logical `1` (`true`) or `0` (`false`).

Indicator	Behavior
`true`	Variable names are included as the column headings of the output. This is the default behavior.
`false`	Variable names are not included in the output.

DateLocale — Locale for writing dates
character vector | string scalar

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values to the file, use `DateLocale` to specify the locale in which `write` should write month and day-of-week names and abbreviations. The character vector or string takes the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code indicating a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

For Excel® files, `write` writes variables containing `datetime` arrays as Excel dates and ignores the `'DateLocale'` parameter value. If the `datetime` variables contain years prior to either 1900 or 1904, then `write` writes the variables as text. For more information on Excel dates, see Differences between the 1900 and the 1904 date system in Excel.

Example: `'DateLocale','ja_JP'` or `'DateLocale',"ja_JP"`

Data Types: `char` | `string`

Text Files Only

Delimiter — Field delimiter character
`','` or `'comma'` | `' '` or `'space'` | …

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and one of these specifiers:

Specifier	Field Delimiter

`','`
 	
`'comma'`
	Comma. This is the default behavior.

`' '`
 	
`'space'`
	Space

`'\t'`
 	
`'tab'`
	Tab

Specifier	Field Delimiter

`';'`
 	
 `'semi'`
	Semicolon

`'|'`
 	
 `'bar'`
	Vertical bar

You can use the `'Delimiter'` name-value pair argument only for delimited text files.

Example: `'Delimiter'`,`'space'` or `'Delimiter'`,`"space"`

QuoteStrings — Indicator for writing quoted text
false (default) | true

Indicator for writing quoted text, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false` or `true`. If `'QuoteStrings'` is `true`, then `write` encloses the text in double quotation marks, and replaces any double-quote characters that appear as part of that text with two double-quote characters. For an example, see "Write Quoted Text to CSV File".

You can use the `'QuoteStrings'` name-value pair argument only with delimited text files.

Encoding — Character encoding scheme
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the `write` function uses your system default encoding to write the file.

`'Big5'`	`'ISO-8859-1'`	`'windows-874'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`
`'CP949'`	`'ISO-8859-3'`	`'windows-1250'`
`'EUC-KR'`	`'ISO-8859-4'`	`'windows-1251'`
`'EUC-JP'`	`'ISO-8859-5'`	`'windows-1252'`
`'EUC-TW'`	`'ISO-8859-6'`	`'windows-1253'`
`'GB18030'`	`'ISO-8859-7'`	`'windows-1254'`
`'GB2312'`	`'ISO-8859-8'`	`'windows-1255'`
`'GBK'`	`'ISO-8859-9'`	`'windows-1256'`
`'IBM866'`	`'ISO-8859-11'`	`'windows-1257'`
`'KOI8-R'`	`'ISO-8859-13'`	`'windows-1258'`
`'KOI8-U'`	`'ISO-8859-15'`	`'US-ASCII'`
	`'Macintosh'`	`'UTF-8'`
	`'Shift_JIS'`	

Example: `'Encoding'`,`'system'` or `'Encoding'`,`"system"` uses the system default encoding.

Spreadsheet Files Only

Sheet — Target worksheet
character vector | string scalar | positive integer

Target worksheet, specified as the comma-separated pair consisting of `'Sheet'` and a character vector or a string scalar containing the worksheet name or a positive integer indicating the worksheet index. The worksheet name cannot contain a colon (:). To determine the names of sheets in a spreadsheet file, use `[status,sheets] = xlsfinfo(filename)`.

If the sheet does not exist, then `write` adds a new sheet at the end of the worksheet collection. If the sheet is an index larger than the number of worksheets, then `write` appends empty sheets until the number of worksheets in the workbook equals the sheet index. In either case, `write` generates a warning indicating that it has added a new worksheet.

You can use the `'Sheet'` name-value pair argument only with spreadsheet files.

Example: `'Sheet',2`

Example: `'Sheet', 'MySheetName'`

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Parquet Files Only

VariableCompression — Parquet compression algorithm
`'snappy'` (default) | `'brotli'` | `'gzip'` | `'uncompressed'` | cell array of character vectors | string vector

Parquet compression algorithm, specified as one of these values.

- `'snappy'`, `'brotli'`, `'gzip'`, or `'uncompressed'`. If you specify one compression algorithm then `write` compresses all variables using the same algorithm.
- Alternatively, you can specify a cell array of character vectors or a string vector containing the names of the compression algorithms to use for each variable.

In general, `'snappy'` has better performance for reading and writing, `'gzip'` has a higher compression ratio at the cost of more CPU processing time, and `'brotli'` typically produces the smallest file size at the cost of compression speed.

Example: `write('C:\myData',D,'FileType','parquet','VariableCompression','brotli')`

Example: `write('C:\myData', D, 'FileType', 'parquet', 'VariableCompression', {'brotli' 'snappy' 'gzip'})`

VariableEncoding — Encoding scheme names
`'auto'` (default) | `'dictionary'` | `'plain'` | cell array of character vectors | string vector

Encoding scheme names, specified as one of these values:

- `'auto'` — `write` uses `'plain'` encoding for logical variables, and `'dictionary'` encoding for all others.
- `'dictionary'`, `'plain'` — If you specify one encoding scheme then `write` encodes all variables with that scheme.

- Alternatively, you can specify a cell array of character vectors or a string vector containing the names of the encoding scheme to use for each variable.

In general, `'dictionary'` encoding results in smaller file sizes, but `'plain'` encoding can be faster for variables that do not contain many repeated values. If the size of the dictionary or number of unique values grows to be too big, then the encoding automatically reverts to plain encoding. For more information on Parquet encodings, see Parquet encoding definitions.

Example: write('myData.parquet', D, 'FileType', 'parquet', 'VariableEncoding', 'plain')

Example: write('myData.parquet', D, 'FileType', 'parquet', 'VariableEncoding', {'plain' 'dictionary' 'plain'})

Version — Parquet version to use
'2.0' (default) | '1.0'

Parquet version to use, specified as either `'1.0'` or `'2.0'`. By default, `'2.0'` offers the most efficient storage, but you can select `'1.0'` for the broadest compatibility with external applications that support the Parquet format.

Limitations

In some cases, `write(location, D, 'FileType', type)` creates files that do not represent the original array D exactly. If you use `datastore(location)` to read the checkpoint files, then the result might not have the same format or contents as the original distributed table.

For the `'text'` and `'spreadsheet'` file types, `write` uses these rules:

- `write` outputs numeric variables using `longG` format, and categorical, character, or string variables as unquoted text.
- For non-text variables that have more than one column, `write` outputs multiple delimiter-separated fields on each line, and constructs suitable column headings for the first line of the file.
- `write` outputs variables with more than two dimensions as two-dimensional variables, with trailing dimensions collapsed.
- For cell-valued variables, `write` outputs the contents of each cell as a single row, in multiple delimiter-separated fields, when the contents are numeric, logical, character, or categorical, and outputs a single empty field otherwise.

Do not use the `'text'` or `'spreadsheet'` file types if you need to write an exact checkpoint of the distributed array.

Tips

- Use the `write` function to create *checkpoints* or *snapshots* of your data as you work. This practice allows you to reconstruct distributed arrays directly from files on disk rather than re-executing all of the commands that produced the distributed array.

See Also
datastore | distributed | tall

Topics
"Distributed Arrays"

Introduced in R2017a

zeros

Create array of all zeros for GPU or distributed arrays

Syntax

```
Z = zeros(n,arraytype)
Z = zeros(size1,...,sizeN,arraytype)
Z = zeros(size,arraytype)

Z = zeros(n,datatype,arraytype)
Z = zeros(size1,...,sizeN,datatype,arraytype)
Z = zeros(size,datatype,arraytype)

Z = zeros(n,'like',P)
Z = zeros(size1,...,sizeN,'like',P)
Z = zeros(size,'like',P)

Z = zeros(n,datatype,'like',P)
Z = zeros(size1,...,sizeN,datatype'like',P)
Z = zeros(size,datatype,'like',P)

C = zeros(n,codist)
C = zeros(size1,...,sizeN,codist)
C = zeros(size,codist)

C = zeros(n,datatype,codist)
C = zeros(size1,...,sizeN,datatype,codist)
C = zeros(size,datatype,codist)

C = zeros( ___ ,codist,'noCommunication')
C = zeros( ___ ,codist,'like',P)
```

Description

Z = zeros(n,arraytype) returns an *n*-by-*n* matrix with underlying class of double, zeros in all elements, and the type specified by arraytype.

Z = zeros(size1,...,sizeN,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of double, zeros in all elements, and the type specified by arraytype.

Z = zeros(size,arraytype) returns an array which size is defined by the size vector *size* with underlying class of double, zeros in all elements, and the type specified by arraytype.

Z = zeros(n,datatype,arraytype) returns an *n*-by-*n* matrix with underlying class of datatype, zeros in all elements, and the type specified by arraytype.

Z = zeros(size1,...,sizeN,datatype,arraytype) returns a *size1*-by-...-by-*sizeN* matrix with underlying class of datatype, zeros in all elements, and the type specified by arraytype.

Z = zeros(size,datatype,arraytype) returns an array which size is defined by the size vector *size* with underlying class of datatype, zeros in all elements, and the type specified by arraytype.

`Z = zeros(n,'like',P)` returns an *n*-by-*n* array of zeros with the same underlying class and type as the prototype array, P.

`Z = zeros(size1,...,sizeN,'like',P)` returns a *size1*-by-...-by-*sizeN* array of zeros with the same underlying class and type as the prototype array, P.

`Z = zeros(size,'like',P)` returns an array of zeros which size is defined by the size vector *size* with the same underlying class and type as the prototype array, P.

`Z = zeros(n,datatype,'like',P)` returns an *n*-by-*n* array with underlying class of `datatype`, zeros in all elements, and the same type as the prototype array, P.

`Z = zeros(size1,...,sizeN,datatype'like',P)` returns a *size1*-by-...-by-*sizeN* array with underlying class of `datatype`, zeros in all elements, and the same type as the prototype array, P.

`Z = zeros(size,datatype,'like',P)` returns an array which size is defined by the size vector *size* with underlying class of `datatype`, zeros in all elements, and the same type as the prototype array, P.

`C = zeros(n,codist)` returns an *n*-by-*n* codistributed array with underlying class of double and zeros in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = zeros(size1,...,sizeN,codist)` returns a *size1*-by-...-by-*sizeN* codistributed array with underlying class of double and zeros in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = zeros(size,codist)` returns a codistributed array which size is defined by the size vector *size* with underlying class of double and zeros in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = zeros(n,datatype,codist)` returns an *n*-by-*n* codistributed array with underlying class of `datatype` and zeros in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = zeros(size1,...,sizeN,datatype,codist)` returns a *size1*-by-...-by-*sizeN* codistributed array with underlying class of `datatype` and zeros in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = zeros(size,datatype,codist)` returns a codistributed array which size is defined by the size vector *size* with underlying class of `datatype` and zeros in all elements. The codistributor object `codist` specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

`C = zeros( ___ ,codist,'noCommunication')` specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = zeros(___ ,codist,'like',P) creates a codistributed array of zeros with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

Examples

Create Distributed Zeros Matrix

Create a 1000-by-1000 distributed array of zeros with underlying class double.

```
D = zeros(1000,'distributed');
```

Create Codistributed Zeros Matrix

Create a 1000-by-1000 codistributed double matrix of zeros, distributed by its second dimension (columns).

```
spmd(4)
    C = zeros(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed uint16 matrix of zeros, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = zeros(1000,1000,'uint16',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray Zeros Matrix

Create a 1000-by-1000 gpuArray of zeros, with underlying class uint32:

```
G = zeros(1000,'uint32','gpuArray');
```

Input Arguments

n — Size of array
integer value

Size of the generated array, specified as an integer value.

size1,...,sizeN — Size of each dimension (as separate arguments)
integer values

Size of each dimension of the generated array, specified as separate arguments of two or more integer values.

size — Size of each dimension (as a row vector)
vector

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

arraytype — Array type
'distributed' | 'codistributed' | 'gpuArray'

Type of the generated array, specified as either 'distributed', 'codistributed', or 'gpuArray'.

datatype — Array data type
'double' (default) | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64'

Underlying class of the array, that is the data type of its elements, specified as one of these options:

- 'double'
- 'single'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'

codist — Codistributor
codistributor object

Distribution scheme for creating the codistributed array, specified as a codistributor object. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments.

P — Prototype array
array

Prototype of array to create, specified as an array.

Output Arguments

Z — Array
distributed array | codistributed array | gpuArray

Array of zeros, returned as either a distributed array, a codistributed array, or a gpuArray.

C — Codistributed array
codistributed array

Array of zeros, returned as a codistributed array.

See Also
Inf | NaN | eye | false | ones | true

Introduced in R2006b