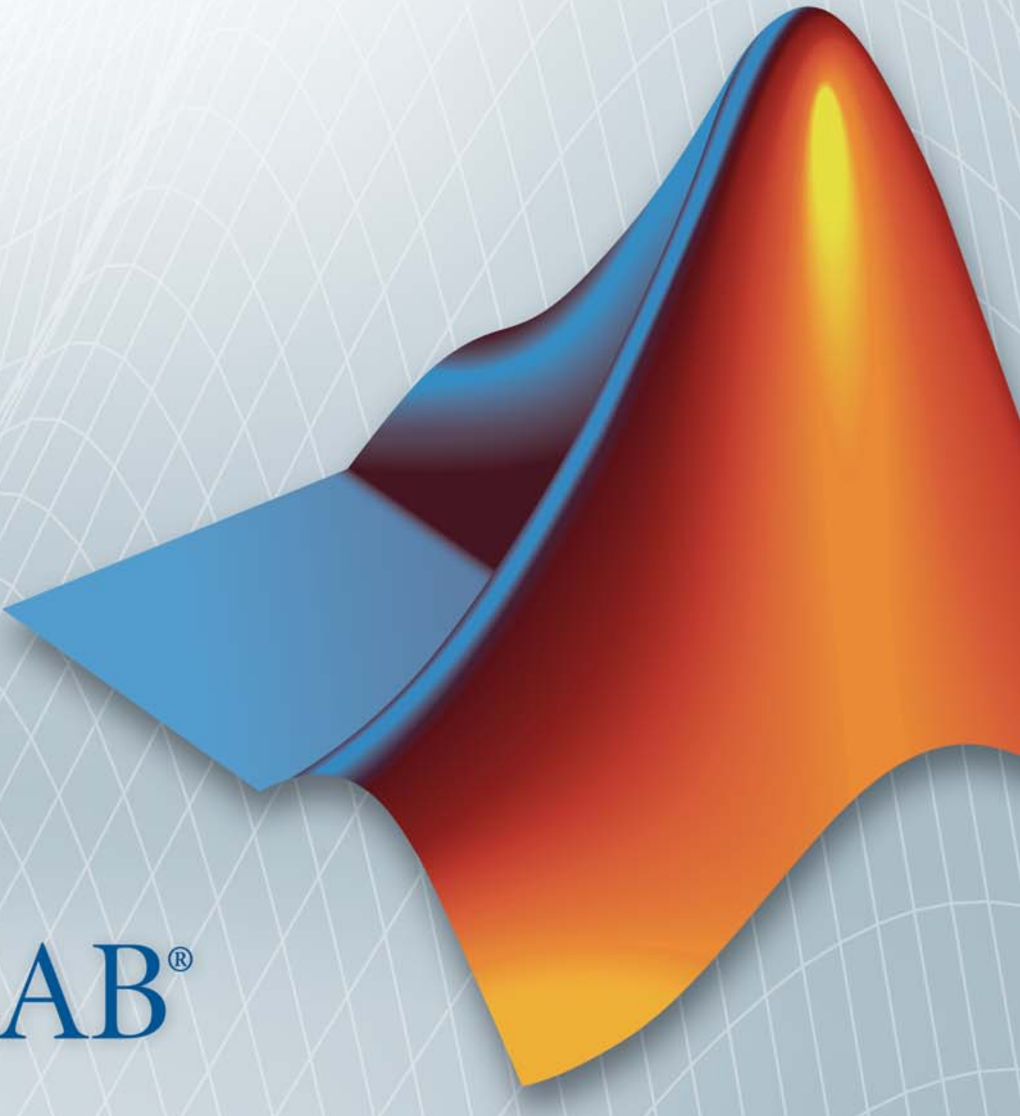
Parallel Computing Toolbox™

User's Guide

R2012a



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Parallel Computing Toolbox™ User's Guide

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)

Contents

8 Program Independent Jobs

9 Program Communicating Jobs

Getting Started

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—let you parallelize MATLAB® applications without CUDA or MPI programming. You can use the toolbox with Simulink® to run multiple simulations of a model in parallel.

The toolbox provides twelve workers (MATLAB computational engines) to execute applications locally on a multicore desktop. Without changing the code, you can run the same application on a computer cluster or a grid computing service (using MATLAB Distributed Computing Server™). You can run parallel applications interactively or in batch.

Key Features

- Parallel for-loops (parfor) for running task-parallel algorithms on multiple processors
- Support for CUDA-enabled NVIDIA GPUs
- Ability to run twelve workers locally on a multicore desktop
- Computer cluster and grid support (with MATLAB Distributed Computing Server)
- Interactive and batch execution of parallel applications
- Distributed arrays and spmd (single-program-multiple-data) for large dataset handling and data-parallel algorithms

Parallel Computing with MathWorks Products

In addition to Parallel Computing Toolbox, MATLAB Distributed Computing Server software allows you to run as many MATLAB workers on a remote cluster of computers as your licensing allows. You can also use MATLAB Distributed Computing Server to run workers on your client machine if you want to run more than twelve local workers.

Most MathWorks products let you code in such a way as to run applications in parallel. For example, Simulink models can run simultaneously in parallel, as described in "Running Parallel Simulations". MATLAB Compiler™ software lets you build and deploy parallel applications, as shown in "Deploying Applications Created Using Parallel Computing Toolbox".

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:

```
http://www.mathworks.com/products/parallel-computing/builtin-parallel-support.html
```

Key Problems Addressed by Parallel Computing

In this section...

Running Parallel for-Loops (parfor)

Many applications involve multiple segments of code, some of which are repetitive. Often you can use for-loops to solve these cases. The ability to execute code in parallel, on one computer or on a cluster of computers, can significantly improve performance in many cases:

- Parameter sweep applications
  - Many iterations — A sweep might take a long time because it comprises many iterations. Each iteration by itself might not take long to execute, but to complete thousands or millions of iterations in serial could take a long time.
  - Long iterations — A sweep might not have a lot of iterations, but each iteration could take a long time to run.

  Typically, the only difference between iterations is defined by different input data. In these cases, the ability to run separate sweep iterations simultaneously can improve performance. Evaluating such iterations in parallel is an ideal way to sweep through large or multiple data sets. The only restriction on parallel loops is that no iterations be allowed to depend on any other iterations.
- Test suites with independent segments — For applications that run a series of unrelated tasks, you can run these tasks simultaneously on separate resources. You might not have used a `for`-loop for a case such as this comprising distinctly different tasks, but a `parfor`-loop could offer an appropriate solution.

Parallel Computing Toolbox software improves the performance of such loop execution by allowing several MATLAB workers to execute individual loop iterations simultaneously. For example, a loop of 100 iterations could run on a cluster of 20 MATLAB workers, so that simultaneously, the workers each execute only five iterations of the loop. You might not get quite 20 times improvement in speed because of communications overhead and network traffic, but the speedup should be significant. Even running local workers all on the same machine as the client, you might see significant performance improvement on a multicore/multiprocessor machine. So whether your loop takes a long time to run because it has many iterations or because each iteration takes a long time, you can improve your loop speed by distributing iterations to MATLAB workers.

Executing Batch Jobs in Parallel

When working interactively in a MATLAB session, you can offload work to a MATLAB worker session to run as a batch job. The command to perform this job is asynchronous, which means that your client MATLAB session is not blocked, and you can continue your own interactive session while the MATLAB worker is busy evaluating your code. The MATLAB worker can run either on the same machine as the client, or if using MATLAB Distributed Computing Server, on a remote cluster machine.

Partitioning Large Data Sets

If you have an array that is too large for your computer's memory, it cannot be easily handled in a single MATLAB session. Parallel Computing Toolbox software allows you to distribute that array among multiple MATLAB workers, so that each worker contains only a part of the array. Yet you can operate on the entire array as a single entity. Each worker operates only on its part of the array, and workers automatically transfer data between themselves when necessary, as, for example, in matrix multiplication. A large number of matrix operations and functions have been enhanced to work directly with these arrays without further modification; see "Using MATLAB Functions on Codistributed Arrays" on page 5-25 and "Using MATLAB Constructor Functions" on page 5-10.

Introduction to Parallel Solutions

In this section...

Interactively Run a Loop in Parallel

This section shows how to modify a simple `for`-loop so that it runs in parallel. This loop does not have a lot of iterations, and it does not take long to execute, but you can apply the principles to larger loops. For these simple examples, you might not notice an increase in execution speed.

1 Suppose your code includes a loop to create a sine wave and plot the waveform:

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

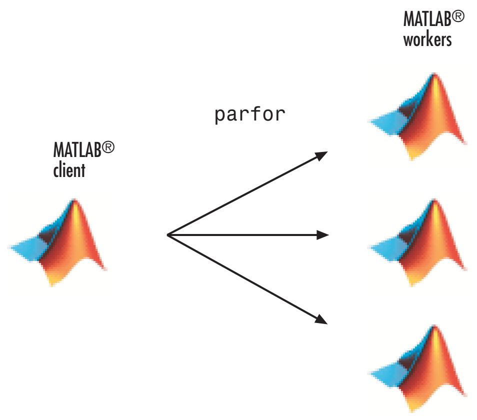
2 To interactively run code that contains a parallel loop, you first open a MATLAB pool. This reserves a collection of MATLAB worker sessions to run your loop iterations. The MATLAB pool can consist of MATLAB sessions running on your local machine or on a remote cluster:

```
matlabpool open local 3
```

3 With the MATLAB pool reserved, you can modify your code to run your loop in parallel by using a `parfor` statement:

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

The only difference in this loop is the keyword `parfor` instead of `for`. After the loop runs, the results look the same as those generated from the previous `for`-loop.



Because the iterations run in parallel in other MATLAB sessions, each iteration must be completely independent of all other iterations. The worker calculating the value for `A(100)` might not be the same worker calculating `A(500)`. There is no guarantee of sequence, so `A(900)` might be calculated before `A(400)`. (The MATLAB Editor can help identify some problems with `parfor` code that might not contain independent iterations.) The only place where the values of all the elements of the array `A` are available is in the MATLAB client, after the data returns from the MATLAB workers and the loop completes.

4 When you are finished with your code, close the MATLAB pool and release the workers:

```
matlabpool close
```

For more information on `parfor`-loops, see Chapter 2, "Parallel for-Loops (parfor)".

The examples in this section run on three local workers. With cluster profiles, you can control how many workers run your loops, and whether the workers are local or on a cluster. For more information on profiles, see "Cluster Profiles" on page 6-12.

You can run Simulink models in parallel loop iterations with the `sim` command inside your loop. For more information and examples of using Simulink with `parfor`, see “Running Parallel Simulations” in the Simulink documentation.

Run a Batch Job

To offload work from your MATLAB session to another session, you can use the `batch` command. This example uses the `for`-loop from the last section inside a script.

1. To create the script, type:

   ```
   edit mywave
   ```

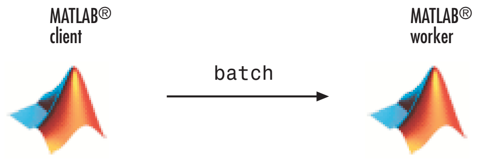
2. In the MATLAB Editor, enter the text of the `for`-loop:

   ```
   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. The `batch` command does not block MATLAB, so you must wait for the job to finish before you can retrieve and view its results:

   ```
   wait(job)
   ```

6. The `load` command transfers variables from the workspace of the worker to the workspace of the client, where you can view the results:

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

```
plot(A)
```

7. When the job is complete, permanently remove its data:

   ```
   destroy(job)
   ```

Run a Batch Parallel Loop

You can combine the abilities to offload a job and run a parallel loop. In the previous two examples, you modified a `for`-loop to make a `parfor`-loop, and you submitted a script with a `for`-loop as a batch job. This example combines the two to create a batch `parfor`-loop.

1. Open your script in the MATLAB Editor:

   ```
   edit mywave
   ```

2. Modify the script so that the `for` statement is a `parfor` statement:

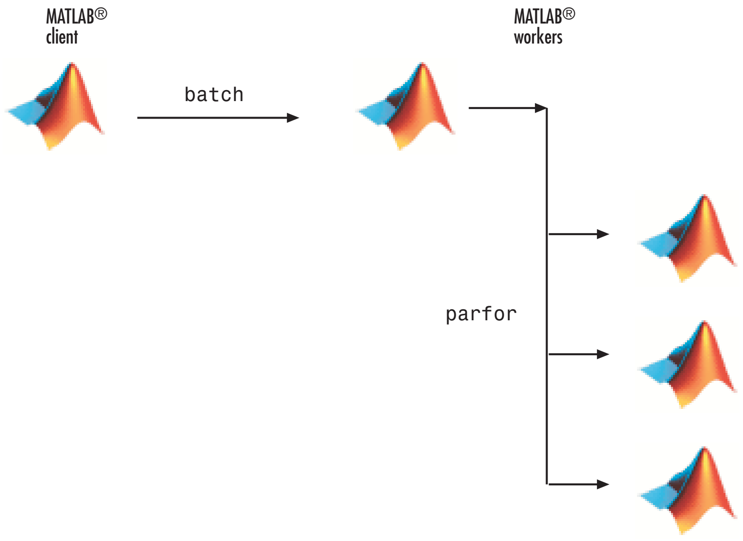
   ```
   parfor i=1:1024
     A(i) = sin(i*2*pi/1024);
   end
   ```

3. Save the file and close the Editor.

4. Run the script in MATLAB with the `batch` command as before, but indicate that the script should use a MATLAB pool for the parallel loop:

   ```
   job = batch('mywave', 'matlabpool', 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.



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:

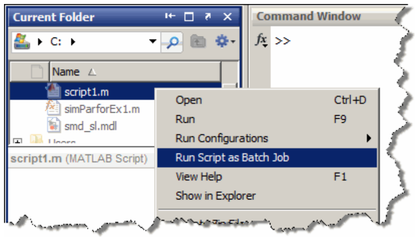
- The work of defining the `parfor`-loop and accumulating its results are offloaded to another MATLAB session (`batch`).
- The loop iterations are distributed from one MATLAB worker to another set of workers running simultaneously (`matlabpool` and `parfor`), so the loop might run faster than having only one worker execute it.

6 When the job is complete, permanently remove its data:

```
destroy(job)
```

Run Script as Batch Jobs 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 current 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 Parallel Loop" 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 6-23.

Distributing Arrays and Running SPMD

Distributed Arrays

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

First, open the MATLAB pool:

```
matlabpool open  % Use default parallel profile
```

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
```

When you are finished and have no further need of data from the workers, you can close the MATLAB pool. Data on the workers does not persist from one instance of a MATLAB pool to another.

```
matlabpool close
```

Single Program Multiple Data

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

```
matlabpool    % Use default parallel profile
spmd          % By default uses all workers in the pool
    R = rand(4);
end
```

This code creates an individual 4-by-4 matrix, `R`, of random numbers on each worker (or lab) 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 MATLAB 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`.

When you are finished with all `spmd` execution and have no further need of data from the workers, you can close the MATLAB pool.

```
matlabpool close
```

Although data persists on the workers from one `spmd` block to another as long as the MATLAB pool remains open, data does not persist from one instance of a MATLAB pool to another.

For more information about using distributed arrays, `spmd`, and Composites, see Chapter 3, "Single Program Multiple Data (spmd)".

Determining 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 Distributed Computing Server installed on your cluster.

Parallel for-Loops (parfor)

Getting Started with parfor

In this section...

parfor-Loops in MATLAB

The basic concept of a `parfor`-loop in MATLAB software is the same as the standard MATLAB `for`-loop: MATLAB executes a series of statements (the loop body) over a range of values. Part of the `parfor` body is executed on the MATLAB client (where the `parfor` is issued) and part is executed in parallel on MATLAB workers. The necessary data on which `parfor` operates is sent from the client to workers, where most of the computation happens, and the results are sent back to the client and pieced together.

Because several MATLAB workers can be computing concurrently on the same loop, a `parfor`-loop can provide significantly better performance than its analogous `for`-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 is useful in situations where you need many loop iterations of a simple calculation, such as a Monte Carlo simulation. `parfor` divides the loop iterations into groups so that each worker executes some portion of the total number of iterations. `parfor`-loops are also useful when you have loop iterations that take a long time to execute, because the workers can execute iterations simultaneously.

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. Since there is a communications cost involved in a `parfor`-loop, there might be no advantage to using one when you have only a small number of simple calculations. The example of this section are only to illustrate the behavior of `parfor`-loops, not necessarily to demonstrate the applications best suited to them.

Creating a parfor-Loop

Set Up MATLAB Resources Using matlabpool

You use the function `matlabpool` to reserve a number of MATLAB workers for executing a subsequent `parfor`-loop. Depending on your scheduler, the workers might be running remotely on a cluster, or they might run locally on your MATLAB client machine. You identify a cluster by selecting a cluster profile. For a description of how to manage and use profiles, see "Cluster Profiles" on page 6-12.

To begin the examples of this section, allocate local MATLAB workers for the evaluation of your loop iterations:

```
matlabpool
```

This command starts the number of MATLAB worker sessions defined by the default cluster. If the local profile is your default and does not specify the number of workers, this starts one worker per core (maximum of twelve) on your local MATLAB client machine.

Note If `matlabpool` is not running, a `parfor`-loop runs serially on the client without regard for iteration sequence.

Program the Loop

The safest assumption about a parfor-loop is that each iteration of the loop is evaluated by a different MATLAB worker. If you have a for-loop in which all iterations are completely independent of each other, this loop is a good candidate for a parfor-loop. Basically, if one iteration depends on the results of another iteration, these iterations are not independent and cannot be evaluated in parallel, so the loop does not lend itself easily to conversion to a parfor-loop.

The following examples produce equivalent results, with a for-loop on the left, and a parfor-loop on the right. Try typing each 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
```

Notice that each element of A is equal to its index. The parfor-loop works because each element depends only upon its iteration of the loop, and upon no other iterations. for-loops that merely repeat such independent tasks are ideally suited candidates for parfor-loops.

Differences Between for-Loops and parfor-Loops

Because `parfor`-loops are not quite the same as `for`-loops, there are special behaviors to be aware of. As seen from the preceding example, when you assign to an array variable (such as `A` in that example) inside the loop by indexing with the loop variable, the elements of that array are available to you after the loop, much the same as with a `for`-loop.

However, suppose you use a nonindexed variable inside the loop, or a variable whose indexing does not depend on the loop variable `i`. Try these examples and notice 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
```

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

Although the elements of `A` come out the same in both of these examples, the value of `d` does not. In the `for`-loop above on the left, the iterations execute in sequence, so afterward `d` has the value it held in the last iteration of the loop. In the `parfor`-loop on the right, the iterations execute in parallel, not in sequence, so it would be impossible to assign `d` a definitive value at the end of the loop. This 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 at all, and their values remain the same before and after the loop.
So, a `parfor`-loop requires that each iteration be independent of the other iterations, and that all code that follows the `parfor`-loop not depend on the loop iteration sequence.

Reduction Assignments: Values Updated by Each Iteration

The next 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, the client properly accumulates or assembles the final loop result.

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

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

If the loop iterations operate in random 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 close or release your pool of workers:

```
clear
matlabpool close
```

The following sections provide further information regarding programming considerations and limitations for `parfor`-loops.

Displaying Output

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

Programming Considerations

In this section...

MATLAB Path

All workers executing a `parfor`-loop must have the same MATLAB search path as the client, so that they can execute any functions called in the body of the loop. 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 `matlabpool` reference page. When the workers are running on a different platform than the client, use the function `pctRunOnAll` to properly set the MATLAB search path on all workers.

Functions files that contain `parfor`-loops must be available on the search path of the workers in the pool running the `parfor`, or made available to the workers by the `AttachedFiles` or `AdditionalPaths` setting of the MATLAB pool.

Error Handling

When an error occurs during the execution of a `parfor`-loop, all iterations that are in progress are terminated, new ones are not initiated, and the loop terminates.

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

The behavior of lastwarn is unspecified at the end of the parfor if used within the loop body.

Limitations

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" in the MATLAB documentation.) 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.

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

Transparency

The body of a parfor-loop must be *transparent*, meaning that all references to variables must be "visible" (i.e., they occur in the text of the program).

In the following example, because X is not visible as an input variable in the parfor body (only the string 'X' is passed to eval), it does not get transferred to the workers. As a result, MATLAB issues an error at run time:

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

Similarly, you cannot clear variables from a worker's workspace by executing clear inside a parfor statement:

```
parfor ii= 1:4
```

```
    <statements...>
    clear('X')  % cannot clear: transparency violation
    <statements...>
end
```

As a workaround, you can free up most of the memory used by a variable by setting its value to empty, presumably when it is no longer needed in your `parfor` statement:

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

Examples of some other functions that violate transparency are `evalc`, `evalin`, and `assignin` with the `workspace` argument specified as `'caller'`; `save` and `load`, unless the output of `load` is assigned to a variable. Running a script from within a `parfor`-loop can cause a transparency violation if the script attempts to access (read or write) variables of the parent workspace; to avoid this issue, convert the script to a function and call it with the necessary variables as input or output arguments.

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

Sliced Variables Referencing Function Handles

Because of the way sliced input variables are segmented and distributed to the workers in the pool, you cannot use a sliced input variable to reference a function handle. If you need to call a function handle with the `parfor` index variable as an argument, use `feval`.

For example, suppose you had a `for`-loop that performs:

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

A corresponding `parfor`-loop does not allow `B` to reference a function handle. So you can work around the problem with `feval`:

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

Nondistributable Functions

If you use a function that is not strictly computational in nature (e.g., `input`, `plot`, `keyboard`) in a `parfor`-loop or in any function called by a `parfor`-loop, the behavior of that function occurs on the worker. The results might include hanging the worker process or having no visible effect at all.

Nested Functions

The body of a `parfor`-loop cannot make reference to a nested function. However, it can call a nested function by means of a function handle.

Nested Loops

The body of a `parfor`-loop cannot contain another `parfor`-loop. But it can call a function that contains another `parfor`-loop.

However, because a worker cannot open a MATLAB pool, a worker cannot run the inner nested `parfor`-loop in parallel. This means that only one level of nested `parfor`-loops can run in parallel. If the outer loop runs in parallel on a MATLAB pool, the inner loop runs serially on each worker. If the outer loop runs serially in the client (e.g., `parfor` specifying zero workers), the function that contains the inner loop can run the inner loop in parallel on workers in a pool.

The body of a `parfor`-loop can contain `for`-loops. You can use the inner loop variable for indexing the sliced array, but only if you use the variable in plain form, not part of an expression. For example:

```
A = zeros(4,5);
parfor j = 1:4
    for k = 1:5
        A(j,k) = j + k;
```

```
   end
end
A
```

Further nesting of `for`-loops with a `parfor` is also allowed.

Limitations of Nested for-Loops. For proper variable classification, the range of a `for`-loop nested in a `parfor` must be defined by constant numbers or variables. In the following example, the code on the left does not work because the `for`-loop upper limit is defined by a function call. The code on the right works around this by defining a broadcast or constant variable outside the `parfor` first:

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

When using the nested `for`-loop variable for indexing the 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:

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

If you use a nested `for`-loop to index into a sliced array, you cannot use that array elsewhere in the `parfor`-loop. For example, 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 the nested loop:

```
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
```

Inside a `parfor`, if you use multiple `for`-loops (not nested inside each other) to index into a single sliced array, they must loop over the same range of values. In the following example, the code on the left does not work because `j` and `k` loop over different values; the code on the right works to index different portions of the sliced array `A`:

```
A = zeros(4, 10);
parfor i = 1:4
   for j = 1:5
      A(i, j) = i + j;
   end
   for k = 6:10
      A(i, k) = pi;
   end
end
```

```
A = zeros(4, 10);
parfor i = 1:4
   for j = 1:10
      if j < 6
         A(i, j) = i + j;
      else
         A(i, j) = pi;
      end
   end
end
```

Nested spmd Statements

The body of a `parfor`-loop cannot contain an `spmd` statement, and an `spmd` statement cannot contain a `parfor`-loop.

Break and Return Statements

The body of a `parfor`-loop cannot contain `break` or `return` statements.

Global and Persistent Variables

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

Handle Classes

Changes made to handle classes on the workers during loop iterations are not automatically propagated to the client.

P-Code Scripts

You can call P-code script files from within a `parfor`-loop, but P-code script cannot contain a `parfor`-loop.

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 “Saving and Loading Objects”.

Performance Considerations

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. For more information, see “Where to Create Arrays” on page 2-32.

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. For more details, see “Optimizing on Local vs. Cluster Workers” on page 2-33.

Compatibility with Earlier Versions of MATLAB Software

In versions of MATLAB prior to 7.5 (R2007b), the keyword `parfor` designated a more limited style of `parfor`-loop than what is available in MATLAB 7.5 and later. This old style was intended for use with codistributed arrays (such as inside an `spmd` statement or a parallel job), and has been replaced by a `for`-loop that uses `drange` to define its range; see “Using a for-Loop Over a Distributed Range (for-drange)” on page 5-21.

The past and current functionality of the `parfor` keyword is outlined in the following table:

Functionality	Syntax Prior to MATLAB 7.5	Current Syntax

Parallel loop for codistributed arrays	`parfor i = range`
	`  loop body`
	`    .`
	`    .`
	`end`
		`for i = drange(range)`
		`  loop body`
		`    .`
		`    .`
		`end`

Parallel loop for implicit distribution of work	Not Implemented	`parfor i = range`
		`  loop body`
		`    .`
		`    .`
		`end`

Advanced Topics

In this section...

About Programming Notes

This section presents guidelines and restrictions in shaded boxes like the one shown below. Those labeled as **Required** result in an error if your `parfor` code does not adhere to them. MATLAB software catches some of these errors at the time it reads the code, and others when it executes the code. These are referred to here as *static* and *dynamic* errors, respectively, and are labeled as **Required (static)** or **Required (dynamic)**. Guidelines that do not cause errors are labeled as **Recommended**. You can use MATLAB Code Analyzer to help make your `parfor`-loops comply with these guidelines.

Required (static): Description of the guideline or restriction

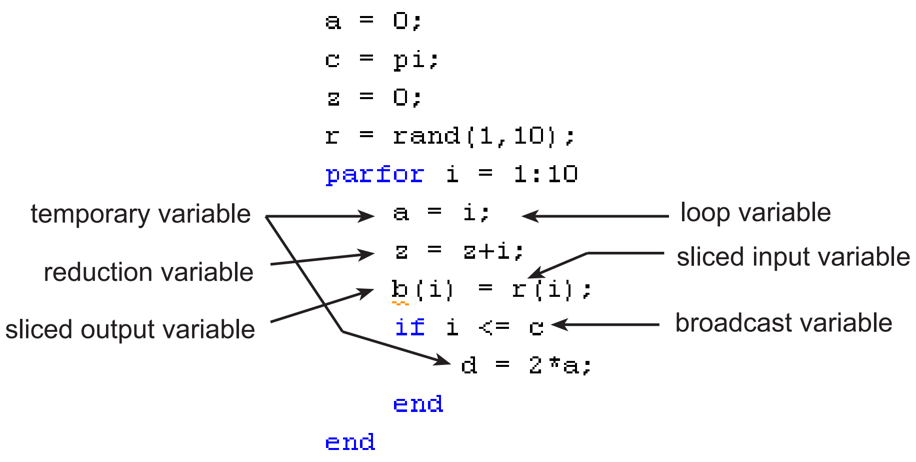
Classification of Variables

Overview

When a name in a `parfor`-loop is recognized as referring to a variable, it is classified into one of the following categories. A `parfor`-loop generates an error if it contains any variables that cannot be uniquely categorized or if any variables violate their category restrictions.

Classification	Description
Loop	Serves as a loop index for arrays
Sliced	An array whose segments are operated on by different iterations of the loop
Broadcast	A variable defined before the loop whose value is used inside the loop, but never assigned inside the loop
Reduction	Accumulates a value across iterations of the loop, regardless of iteration order
Temporary	Variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop

Each of these variable classifications appears in this code fragment:



```
a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;
    z = z+i;
    b(i) = r(i);
    if i <= c
        d = 2*a;
    end
end
```

Loop Variable

The following restriction is required, because changing `i` in the `parfor` body invalidates the assumptions MATLAB makes about communication between the client and workers.

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

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

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

Sliced Variables

A *sliced variable* is one whose value can be broken up into segments, or *slices*, which are then operated on separately by workers and by the MATLAB client. Each iteration of the loop works on a different slice of the array. Using sliced variables is important because this type of variable can reduce communication between the client and workers. Only those slices needed by a worker are sent to it, and only when it starts working on a particular range of indices.

In the next example, a slice of `A` consists of a single element of that array:

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

Characteristics of a Sliced Variable. A variable in a `parfor`-loop is sliced if it has all of the following characteristics. A description of each characteristic follows the list:

- Type of First-Level Indexing — The first level of indexing is either parentheses, `()`, or braces, `{}`.
- Fixed Index Listing — Within the first-level parenthesis 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 — In assigning to a sliced variable, the right-hand side of the assignment is not `[]` or `''` (these operators indicate deletion of elements).

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

This table lists the forms for the first level of indexing for arrays sliced and not sliced.

Reference for Variable Not Sliced	Reference for Sliced Variable
`A.x`	`A(...)`
`A.(...)`	`A{...}`

After the first level, you can use any type of valid MATLAB indexing in the second and further 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 parentheses or braces of a sliced variable's indexing, the list of indices is the same for all occurrences of a given variable.

The variable `A` shown here on the left is not sliced because A is indexed by `i` and `i+1` in different places; that shown on the right is sliced:

```
parfor i = 1:k                        parfor i = 1:k
   B(:) = h(A(i), A(i+1));               B(:) = f(A(i));
end                                      C(:) = g(A{i});
                                      end
```

The example above on the right shows some occurrences of a sliced variable with first-level parenthesis indexing and with first-level brace indexing in the same loop. This is acceptable.

Form of Indexing. Within the list of indices for a sliced variable, one of these indices is of the form `i`, `i+k`, `i-k`, `k+i`, or `k-i`, where `i` is the loop variable and k is a constant or a simple (nonindexed) broadcast variable; and every other index is a constant, a simple broadcast variable, colon, or end.

With i as the loop variable, the A variables shown here on the left are not sliced; those on the right are sliced:

```
A(i+f(k),j,:,3)                    A(i+k,j,:,3)
A(i,20:30,end)                     A(i,:,end)
A(i,:,s.field1)                    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 on either line is not sliced:

```
A(i,:) = [];
A(end + 1) = i;
```

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

Sliced Input and Output Variables. All sliced variables have the characteristics of being input or output. A sliced variable can sometimes have both characteristics. 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, `r` is a sliced input variable and `b` is a sliced output variable:

```
a = 0;
z = 0;
r = rand(1,10);
parfor ii = 1:10
   a = ii;
   z = z + ii;
   b(ii) = r(ii);
end
```

However, if it is clear that in every iteration, every reference to an array element is set before it is used, the variable is not a sliced input variable. In this example, all the elements of `A` are set, and then only those fixed values are used:

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

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

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

Broadcast Variables

A *broadcast variable* is any variable other than the loop variable or a sliced variable that is not affected by an assignment inside the loop. At the start of a `parfor`-loop, the values of any broadcast variables are sent to all workers. Although this type of variable can be useful or even essential, broadcast variables that are large can cause a lot of communication between client and workers. In some cases it might be more efficient to use temporary variables for this purpose, creating and assigning them inside the loop.

Reduction Variables

MATLAB supports an important exception, called reductions, 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 side 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 "Further Considerations with Reduction Variables" on page 2-25
`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 = {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*, and, by definition, a reduction variable can appear only in assignments of this type.

The following example shows a typical usage of a reduction variable `X`:

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

This loop is equivalent to the following, where each `d(i)` is calculated by a different iteration:

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

If the loop were a regular `for`-loop, the variable `X` in each iteration 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 workers' partial sums into `X`. Thus, workers do some of the additions, and the client does the rest.

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:

```
parfor i = 1:n
   if testLevel(k)
      A = A + i;
   else
      A = [A, 4+i];
   end
   % loop body continued
end
```

```
parfor i = 1:n
   if testLevel(k)
      A = A + i;
   else
      A = A + i + 5*k;
   end
   % loop body continued
end
```

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

The `parfor`-loop on the left below 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:

```
parfor i = 1:n
   if testLevel(k)
      A = [A, 4+i];
   else
      A = [r(i), A];
   end
   % loop body continued
end
```

```
parfor i = 1:n
   if testLevel(k)
      A = [A, 4+i];
   else
      A = [A, r(i)];
   end
   % loop body continued
end
```

Further Considerations with Reduction Variables. This section provide more detail about reduction assignments, associativity, commutativity, and overloading of reduction functions.

Reduction Assignments. In addition to the specific forms of reduction assignment listed in the table in "Reduction Variables" on page 2-23, 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 it is a variable, it must not be affected by 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, this is not strictly required; as long as the right-hand side can be evaluated, the resulting value is stored in `X`.

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

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

Note that the operators `&&` and `||` are not listed in the table in “Reduction Variables” on page 2-23. 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`, but `f(u,v)` *always* evaluates `v` before calling `f`. This is why `&&` 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. Concerning the function `f` as used in the definition of a reduction variable, the following practice is recommended, but 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, different executions of the loop might result in different answers.

Note While the addition of mathematical real numbers is associative, addition of floating-point numbers is only approximately associative, and different executions of this `parfor` statement might produce values of `X` with different round-off errors. This is an unavoidable 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)
```

With the exception of the minus operator (`-`), all the special cases listed in the table in "Reduction Variables" on page 2-23 have a corresponding (perhaps 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)
```

Examples of noncommutative functions are `*` (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 should be commutative. If `f` is not commutative, different executions of the loop might result in different answers.

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

Overloading in Reduction Assignments. Most associative functions `f` have an identity element `e`, so that for any `a`, the following holds true:

```
f(e,a) = a = f(a,e)
```

Examples of identity elements for some functions are listed in this table.

Function	Identity Element
`+`	`0`
`*` and `.*`	`1`
`min`	`Inf`
`max`	`-Inf`
`[,]`, `[;]`, and `union`	`[]`

MATLAB uses the identity elements of reduction functions when it knows them. So, in addition to associativity and commutativity, you should also keep identity elements in mind when overloading these functions.

Recommended: An overload of `+`, `*`, `.*`, `min`, `max`, `union`, `[,]`, or `[;]` should be associative if it is used in a reduction assignment in a `parfor`. The overload must treat the respective identity element given above (all with class `double`) as an identity element.

Recommended: An overload of `+`, `.*`, `min`, `max`, `union`, or `intersect` should be commutative.

There is no way to specify the identity element for a function. In these cases, the behavior of `parfor` is a little less efficient than it is for functions with a known identity element, but the results are correct.

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

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

Example: Using a Custom Reduction Function. Suppose each iteration of a loop performs some calculation, and you are interested in finding which iteration of a loop produces the maximum value. This is a reduction exercise that makes an accumulation across multiple iterations of a loop. Your reduction function must compare iteration results, until finally the maximum value can be determined after all iterations are compared.

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

```
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
```

Inside the loop, each iteration calls the reduction function (`comparemax`), passing in a pair of 2-element vectors:

- The accumulated maximum and its iteration index (this is the reduction variable, `cummax`)

- The iteration's own calculation 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.

The code for the loop looks like the following, with each iteration calling the reduction function `comparemax` to compare its own data `[dat i]` to that already accumulated in `cummax`.

```
% 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);
```

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 effectively 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 the context of the `parfor` statement has no effect on a variable with the same name that exists outside the loop, again in contrast to ordinary `for`-loops.

Uninitialized Temporaries. Because temporary variables are cleared at the beginning of every iteration, MATLAB can detect certain cases in which any iteration through the loop uses the temporary variable before it is set in that iteration. In this case, MATLAB issues a static error rather than a run-time error, because 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, suppose you write

```
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, but 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, so that `do_something(i)` is executed for only 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, but you use it elsewhere in the loop, causing it technically to be 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` were the two in the first statement of the body, it 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, so MATLAB issues an error about this fact, but points out the possible connection with reduction.

Note that if you change `parfor` to `for`, the use of `s` outside the reduction assignment relies on the iterations being performed in a particular order. The point here is that 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.

Improving Performance

Where to Create Arrays

With a `parfor`-loop, it might be faster to have each MATLAB worker create its own arrays or portions of them in parallel, rather than to create a large array in the client before the loop and send it out to all the workers separately. Having each worker create its own copy of these arrays inside the loop saves the time of transferring the data from client to workers, because all the workers can be creating it at the same time. This might challenge your usual practice to do as much variable initialization before a `for`-loop as possible, so that you do not needlessly repeat it inside the loop.

Whether to create arrays before the `parfor`-loop or inside the `parfor`-loop depends on the size of the arrays, the time needed to create them, whether the workers need all or part of the arrays, the number of loop iterations that each worker performs, and other factors. While many `for`-loops can be directly converted to `parfor`-loops, even in these cases there might be other issues involved in optimizing your code.

Optimizing on Local vs. Cluster Workers

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. 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 matlabpool locally, and notice the difference in time execution for each loop. First open a local matlabpool:

```
matlabpool
```

Then enter the following examples. (If you are viewing this documentation in the MATLAB help browser, highlight each segment of code below, right-click, and select **Evaluate Selection** in the context menu to execute the block in MATLAB. That way the time measurement will not include the time required to paste or type.)

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

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

Single Program Multiple Data (spmd)

Executing Simultaneously on Multiple Data Sets

In this section...

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.

Setting Up MATLAB Resources Using matlabpool

You use the function `matlabpool` to reserve a number of MATLAB labs (workers) for executing a subsequent `spmd` statement or `parfor`-loop. Depending on your scheduler, the workers might be running remotely on a cluster, or they might run locally on your MATLAB client machine. You identify a cluster by selecting a cluster profile. For a description of how to manage and use profiles, see "Cluster Profiles" on page 6-12.

To begin the examples of this section, allocate local MATLAB workers for the evaluation of your `spmd` statement:

```
matlabpool
```

This command starts the number of MATLAB worker sessions defined by the default cluster profile. If the local profile is your default and does not specify the number of workers, this starts one worker per core (maximum of twelve) on your local MATLAB client machine.

If you do not want to use default settings, you can specify in the `matlabpool` statement which profile or how many workers to use. For example, to use only three workers with your default profile, type:

```
matlabpool 3
```

To use a different profile, type:

```
matlabpool MyProfileName
```

To inquire whether you currently have a MATLAB pool open, type:

```
matlabpool size
```

This command returns a value indicating the number of workers in the current pool. If the command returns 0, there is currently no pool open.

Note If there is no MATLAB pool open, an spmd statement runs locally in the MATLAB client without any parallel execution, provided you have Parallel Computing Toolbox software installed. In other words, it runs in your client session as though it were a single worker.

When you are finished using a MATLAB pool, close it with the command:

```
matlabpool close
```

Defining an spmd Statement

The general form of an spmd statement is:

```
spmd
    <statements>
end
```

The block of code represented by <statements> executes in parallel simultaneously on all workers in the MATLAB 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 MATLAB 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 open.

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:

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

Note All subsequent examples in this chapter assume that a MATLAB 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 parallel job, you are allowed to directly control communications between the workers, transfer data between them, and use codistributed arrays among them. For a list of toolbox functions that facilitate these capabilities, see the Function Reference sections "Interlab Communication Within a Communicating Job" on page 13-11 and "Distributed and Codistributed Arrays" on page 13-4.

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 Chapter 5, "Math with Codistributed Arrays".

Displaying Output

When running an spmd statement on a MATLAB 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.

Accessing Data with Composites

In this section...

Introduction

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:

- Using the `Composite` function on the client. Values assigned to the Composite elements are stored on the workers.
- Defining variables on workers inside an `spmd` statement. After the `spmd` statement, the stored values are accessible on the client as Composites.

Creating 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 open a MATLAB pool of three local workers and run an `spmd` statement on that pool:

```
matlabpool open 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 close the pool:

```
matlabpool close
```

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 MATLAB pool is closed. The following example illustrates data value lifespan with spmd blocks, using a pool of four workers:

```
matlabpool open 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

matlabpool close
```

Creating 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 MATLAB pool is already open:

```
PP = Composite()
```

By default, this creates a Composite with an element for each worker in the MATLAB 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:

```
for ii = 1:numel(PP)
```

```
    PP{ii} = ii;
end
```

Distributing Arrays

In this section...

Distributed Versus Codistributed Arrays

You can create a distributed array in the MATLAB client, and its data is stored on the workers of the open MATLAB pool. A distributed array is distributed in one dimension, along the last nonsingleton 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 codistributed array by executing on the workers themselves, either inside an `spmd` statement, in pmode, or inside a parallel 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. Distributed arrays are partitioned among workers from the client with the open MATLAB pool. 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 is as a distributed array in the client. Only `spmd` statements let you access the same array data from two different perspectives.

Creating Distributed Arrays

You can create a distributed array in any of several ways:

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

- Use any of the overloaded distributed object methods 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. These overloaded functions include `distributed.eye`, `distributed.rand`, etc. For a full list, see the `distributed` object reference page.
- Create a codistributed array inside an `spmd` statement, then access it as a distributed array outside the `spmd` statement. This lets you use distribution schemes other than the default.

The first two of these techniques do not involve `spmd` in creating the array, but you can see how `spmd` might be used to manipulate arrays created this way. For example:

Create an array in the client workspace, then make it a distributed array:

```
matlabpool open local 2
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.
matlabpool close
```

Creating Codistributed Arrays

You can create a codistributed array in any of several ways:

- Use the `codistributed` function inside an `spmd` statement, a parallel job, or pmode to codistribute data already existing on the workers running that job.
- Use any of the overloaded codistributed object methods to directly construct a codistributed array on the workers. This technique does not require that the array already exists in the workers. These overloaded functions include `codistributed.eye`, `codistributed.rand`, etc. 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 MATLAB pool.

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.

```
matlabpool open local 2
spmd
    codist = codistributor1d(3, [4, 12]);
    Z = codistributed.zeros(3, 3, 16, codist);
    Z = Z + labindex;
end
Z  % View results in client.
   % Z is a distributed array here.
matlabpool close
```

For more details on codistributed arrays, see Chapter 5, “Math with Codistributed Arrays”, and Chapter 4, “Interactive Parallel Computation with pmode”.

Programming Tips

In this section...

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 `matlabpool` 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 lab (worker) ID 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.

Limitations

Transparency

The body of an `spmd` statement must be *transparent*, meaning that all references to variables must be "visible" (i.e., they occur in the text of the program).

In the following example, because X is not visible as an input variable in the spmd body (only the string 'X' is passed to eval), it does not get transferred to the workers. As a result, MATLAB issues an error at run time:

```
X = 5;
spmd
    eval('X');
end
```

Similarly, you cannot clear variables from a worker's workspace by executing clear inside an spmd statement:

```
spmd; clear('X'); end
```

To clear a specific variable from a worker, clear its Composite from the client workspace. Alternatively, you can free up most of the memory used by a variable by setting its value to empty, presumably when it is no longer needed in your spmd statement:

```
spmd
    <statements....>
    X = [];
end
```

Examples of some other functions that violate transparency are evalc, evalin, and assignin with the workspace argument specified as 'caller'; save and load, unless the output of load is assigned to a variable.

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

Nested Functions

Inside a function, the body of an spmd statement cannot make any direct reference to 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 do not get updated in the workspace of the outer function.

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.

Nested spmd Statements

The body of an `spmd` statement cannot contain another `spmd`. However, it can call a function that contains another `spmd` statement. Be sure that your MATLAB pool has enough workers to accommodate such expansion.

Nested parfor-Loops

The body of a `parfor`-loop cannot contain an `spmd` statement, and an `spmd` statement cannot contain a `parfor`-loop.

Break and Return Statements

The body of an `spmd` statement cannot contain `break` or `return` statements.

Global and Persistent Variables

The body of an `spmd` statement cannot contain `global` or `persistent` variable declarations.

Interactive Parallel Computation with pmode

This chapter describes interactive pmode in the following sections:

pmode Versus spmd

pmode lets you work interactively with a parallel job running simultaneously on several workers. Commands you type at the pmode prompt in the Parallel Command Window are executed on all workers at the same time. Each (worker) lab executes the commands in its own workspace on its own variables.

The way the workers remain synchronized is that each worker becomes idle when it completes a command or statement, waiting until all the workers working on this job have completed the same statement. Only when all the workers are idle, do they then proceed together to the next pmode command.

In contrast to `spmd`, pmode provides a desktop with a display for each worker running the job, where you can enter commands, see results, access each worker's workspace, etc. What pmode does not let you do is to freely interleave serial and parallel work, like `spmd` does. When you exit your pmode session, its job is effectively destroyed, and all information and data on the workers is lost. Starting another pmode session always begins from a clean state.

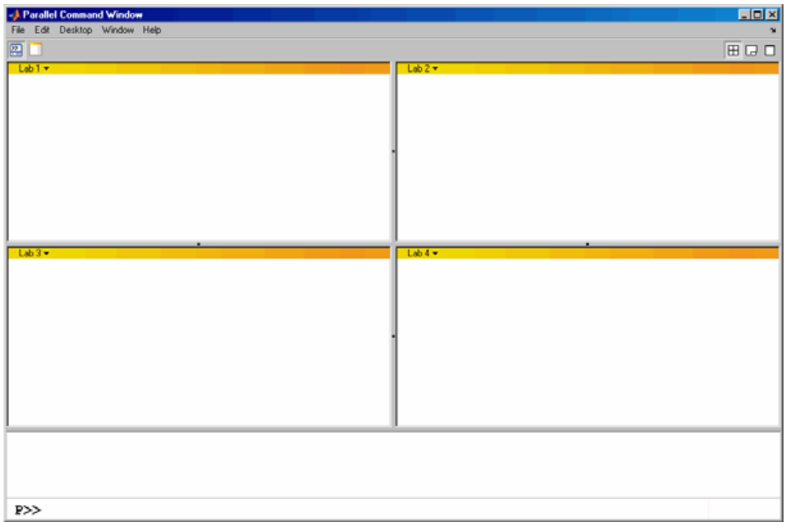
Run Parallel Jobs Interactively Using pmode

This example uses a local scheduler and runs the workers on your local MATLAB client machine. It does not require an external cluster or scheduler. The steps include the pmode prompt (`P>>`) for commands that you type in the Parallel Command Window.

1. Start the pmode with the `pmode` command.

   ```
   pmode start local 4
   ```

   This starts four local workers, creates a parallel job to run on those workers, and opens the Parallel Command Window.

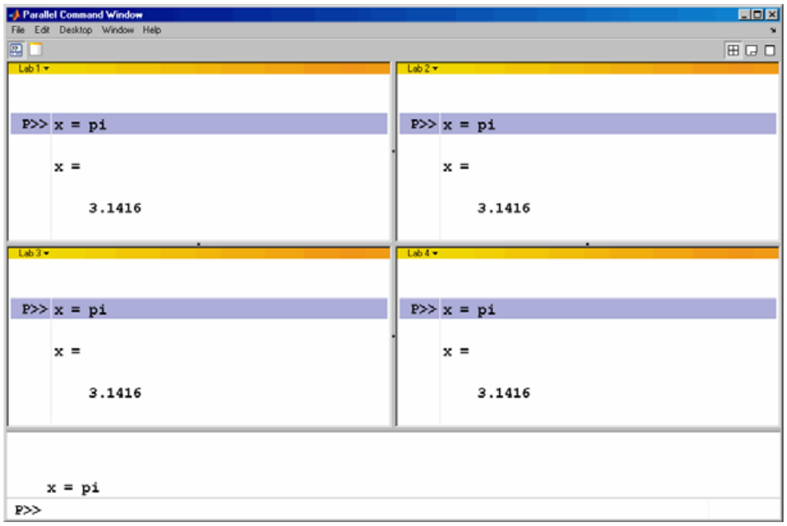
   You can control where the command history appears. For this exercise, the position is set by clicking **Window > History Position > Above Prompt**, but you can set it according to your own preference.

2. To illustrate that commands at the pmode prompt are executed on all workers, ask for help on a function.

   ```
   P>> help magic
   ```

3 Set a variable at the pmode prompt. Notice that the value is set on all the workers.

```
P>> x = pi
```



```
P>> x = pi

x =

    3.1416
```

```
x = pi
P>>
```

4 A variable does not necessarily have the same value on every worker. The `labindex` function returns the ID particular to each worker working on this parallel job. In this example, the variable `x` exists with a different value in the workspace of each worker.

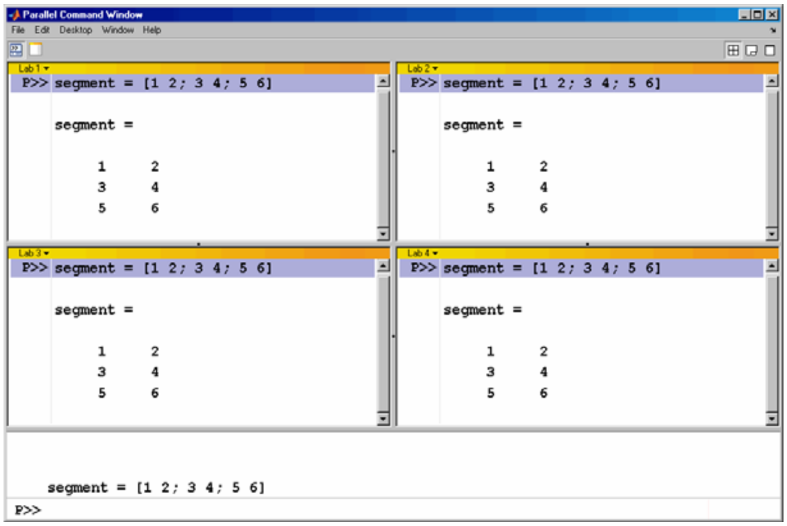
```
P>> x = labindex
```

5 Return the total number of workers working on the current parallel job with the `numlabs` function.

```
P>> all = numlabs
```

6 Create a replicated array on all the workers.

```
P>> segment = [1 2; 3 4; 5 6]
```

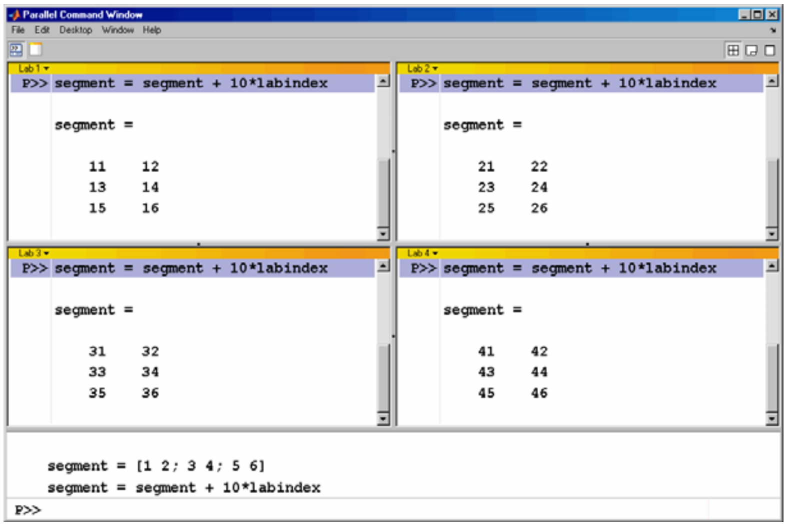


```
P>> segment = [1 2; 3 4; 5 6]

segment =

     1     2
     3     4
     5     6
```

```
segment = [1 2; 3 4; 5 6]
P>>
```

7 Assign a unique value to the array on each worker, dependent on the worker number (`labindex`). With a different value on each worker, this is a variant array.

```
P>> segment = segment + 10*labindex
```



```
P>> segment = segment + 10*labindex

    segment =

        11    12
        13    14
        15    16
```

```
P>> segment = segment + 10*labindex

    segment =

        21    22
        23    24
        25    26
```

```
P>> segment = segment + 10*labindex

    segment =

        31    32
        33    34
        35    36
```

```
P>> segment = segment + 10*labindex

    segment =

        41    42
        43    44
        45    46
```

```
    segment = [1 2; 3 4; 5 6]
    segment = segment + 10*labindex
P>>
```

8 Until this point in the example, the variant arrays are independent, other than having the same name. Use the `codistributed.build` function to aggregate the array segments into a coherent array, distributed among the workers.

```
P>> codist = codistributor1d(2, [2 2 2 2], [3 8])
P>> whole = codistributed.build(segment, codist)
```

This combines four separate 3-by-2 arrays into one 3-by-8 codistributed array. The `codistributor1d` object indicates that the array is distributed along its second dimension (columns), with 2 columns on each of the four workers. On each worker, `segment` provided the data for the local portion of the `whole` array.

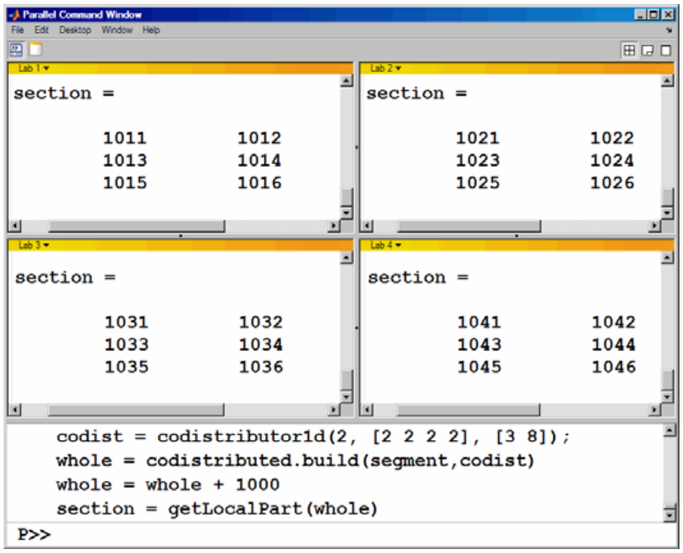
9 Now, when you operate on the codistributed array `whole`, each worker handles the calculations on only its portion, or segment, of the array, not the whole array.

```
P>> whole = whole + 1000
```

10 Although the codistributed array allows for operations on its entirety, you can use the getLocalPart function to access the portion of a codistributed array on a particular worker.

```
P>> section = getLocalPart(whole)
```

Thus, section is now a variant array because it is different on each worker.



```
section =

        1011        1012
        1013        1014
        1015        1016
```

```
section =

        1021        1022
        1023        1024
        1025        1026
```

```
section =

        1031        1032
        1033        1034
        1035        1036
```

```
section =

        1041        1042
        1043        1044
        1045        1046
```

```
    codist = codistributor1d(2, [2 2 2 2], [3 8]);
    whole = codistributed.build(segment,codist)
    whole = whole + 1000
    section = getLocalPart(whole)
P>>
```

11 If you need the entire array in one workspace, use the gather function.

```
P>> combined = gather(whole)
```

Notice, however, that this gathers the entire array into the workspaces of all the workers. See the gather reference page for the syntax to gather the array into the workspace of only one worker.

12 Because the workers ordinarily do not have displays, if you want to perform any graphical tasks involving your data, such as plotting, you must do this from the client workspace. Copy the array to the client workspace by typing the following commands in the MATLAB (client) Command Window.

```
pmode lab2client combined 1
```

Notice that `combined` is now a 3-by-8 array in the client workspace.

```
whos combined
```

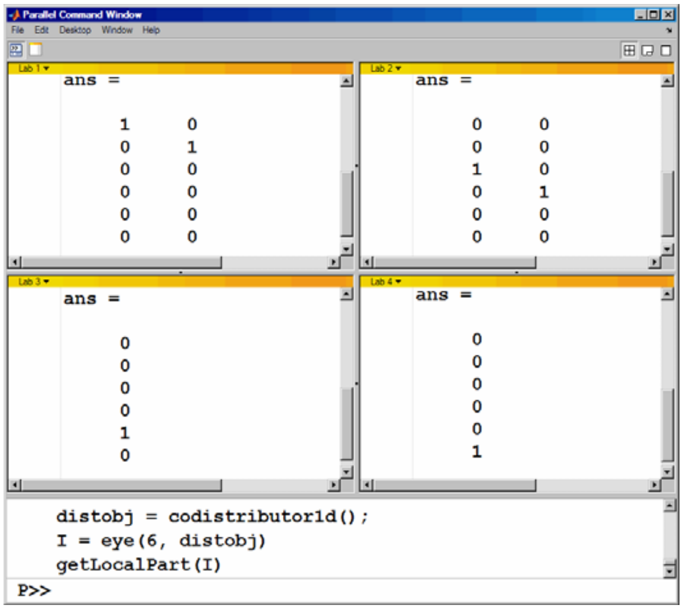
To see the array, type its name.

```
combined
```

13 Many matrix functions that might be familiar can operate on codistributed arrays. For example, the `eye` function creates an identity matrix. Now you can create a codistributed identity matrix with the following commands in the Parallel Command Window.

```
P>> distobj = codistributor1d();
P>> I = eye(6, distobj)
P>> getLocalPart(I)
```

Calling the `codistributor1d` function without arguments specifies the default distribution, which is by columns in this case, distributed as evenly as possible.



```
ans =

     1     0
     0     1
     0     0
     0     0
     0     0
     0     0
```

```
ans =

     0     0
     0     0
     1     0
     0     1
     0     0
     0     0
```

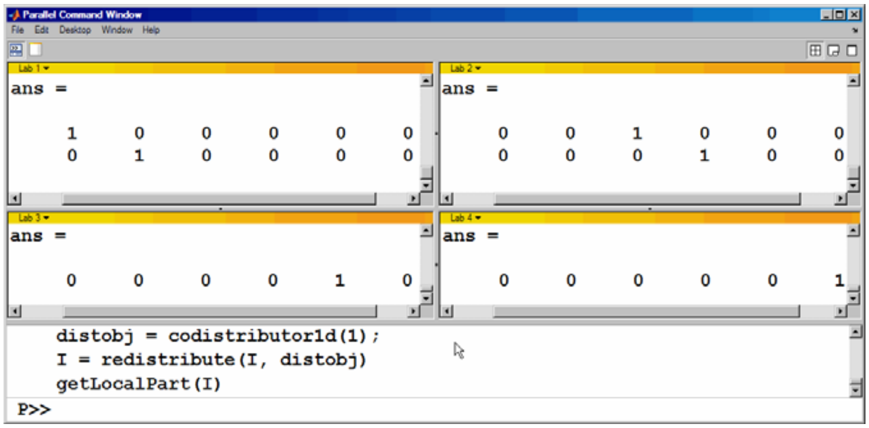
```
ans =

     0
     0
     0
     0
     1
     0
```

```
ans =

     0
     0
     0
     0
     0
     1
```

```
distobj = codistributor1d();
I = eye(6, distobj)
getLocalPart(I)
P>>
```

14 If you require distribution along a different dimension, you can use the `redistribute` function. In this example, the argument `1` to `codistributor1d` specifies distribution of the array along the first dimension (rows).

```
P>> distobj = codistributor1d(1);
P>> I = redistribute(I, distobj)
P>> getLocalPart(I)
```



```
ans =

     1     0     0     0     0     0
     0     1     0     0     0     0
```

```
ans =

     0     0     1     0     0     0
     0     0     0     1     0     0
```

```
ans =

     0     0     0     0     1     0
```

```
ans =

     0     0     0     0     0     1
```

```
    distobj = codistributor1d(1);
    I = redistribute(I, distobj)
    getLocalPart(I)
 P>>
```

15 Exit pmode and return to the regular MATLAB desktop.

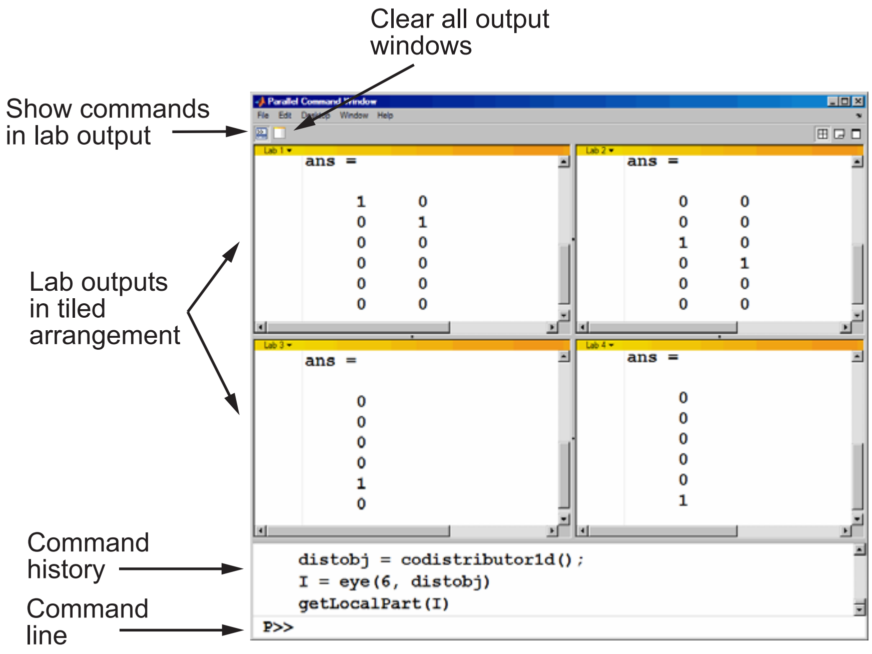
```
P>> pmode exit
```

Parallel Command Window

When you start pmode on your local client machine with the command

```
pmode start local 4
```

four workers start on your local machine and a parallel job is created to run on them. The first time you run pmode with these options, you get a tiled display of the four workers.



```
ans =

     1     0
     0     1
     0     0
     0     0
     0     0
     0     0
```

```
ans =

     0     0
     0     0
     1     0
     0     1
     0     0
     0     0
```

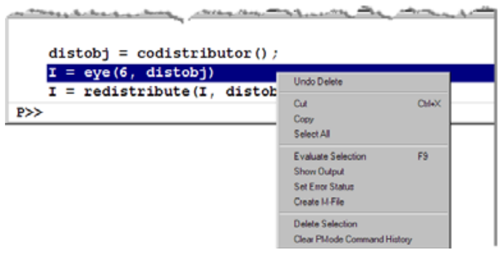
```
ans =

     0
     0
     0
     0
     1
     0
```

```
ans =

     0
     0
     0
     0
     0
     1
```

```
distobj = codistributor1d();
I = eye(6, distobj)
getLocalPart(I)
P>>
```

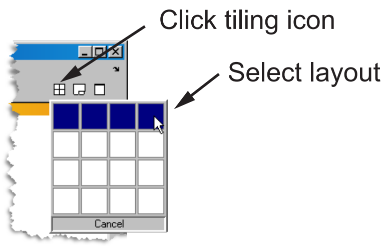
The Parallel Command Window offers much of the same functionality as the MATLAB desktop, including command line, output, and command history.

When you select one or more lines in the command history and right-click, you see the following context menu.

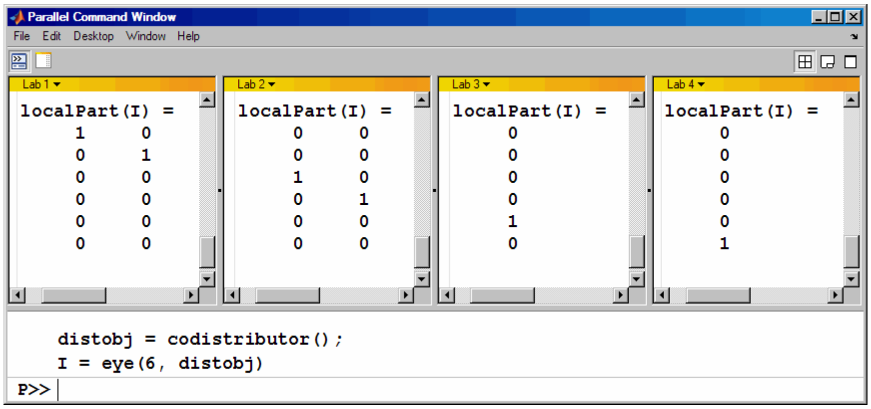


```
distobj = codistributor();
I = eye(6, distobj)
I = redistribute(I, distob
P>>
```

You have several options for how to arrange the tiles showing your worker outputs. Usually, you will choose an arrangement that depends on the format of your data. For example, the data displayed until this point in this section, as in the previous figure, is distributed by columns. It might be convenient to arrange the tiles side by side.



This arrangement results in the following figure, which might be more convenient for viewing data distributed by columns.



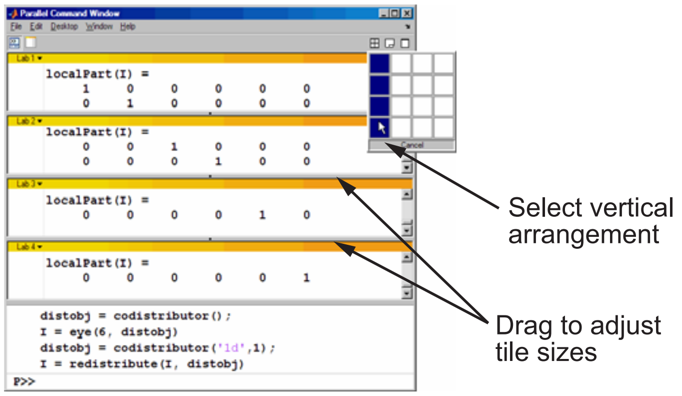
```
Lab 1                Lab 2                Lab 3                Lab 4
localPart(I) =       localPart(I) =       localPart(I) =       localPart(I) =
     1     0              0     0              0                    0
     0     1              0     0              0                    0
     0     0              1     0              0                    0
     0     0              0     1              0                    0
     0     0              0     0              1                    0
     0     0              0     0              0                    1
```

```
    distobj = codistributor();
    I = eye(6, distobj)
P>>
```

Alternatively, if the data is distributed by rows, you might want to stack the worker tiles vertically. For the following figure, the data is reformatted with the command

```
P>> distobj = codistributor('1d',1);
P>> I = redistribute(I, distobj)
```

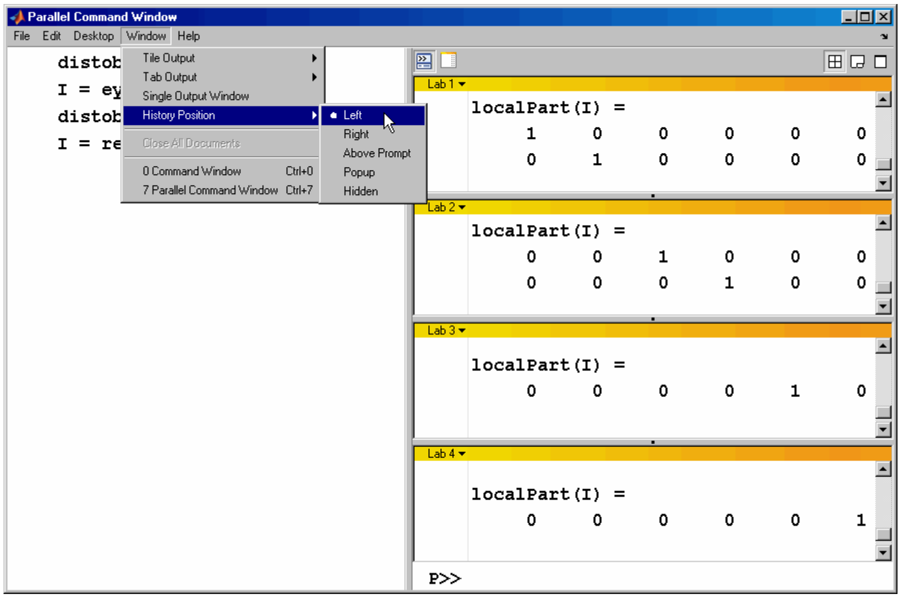
When you rearrange the tiles, you see the following.


Parallel Command Window
File Edit Desktop Window Help
Lab 1
Lab 2
Lab 3
Lab 4
Cancel
Select vertical arrangement
Drag to adjust tile sizes

```
Lab 1
    localPart(I) =
         1     0     0     0     0     0
         0     1     0     0     0     0
Lab 2
    localPart(I) =
         0     0     1     0     0     0
         0     0     0     1     0     0
Lab 3
    localPart(I) =
         0     0     0     0     1     0
Lab 4
    localPart(I) =
         0     0     0     0     0     1
```

```
    distobj = codistributor();
    I = eye(6, distobj)
    distobj = codistributor('1d',1);
    I = redistribute(I, distobj)
P>>
```

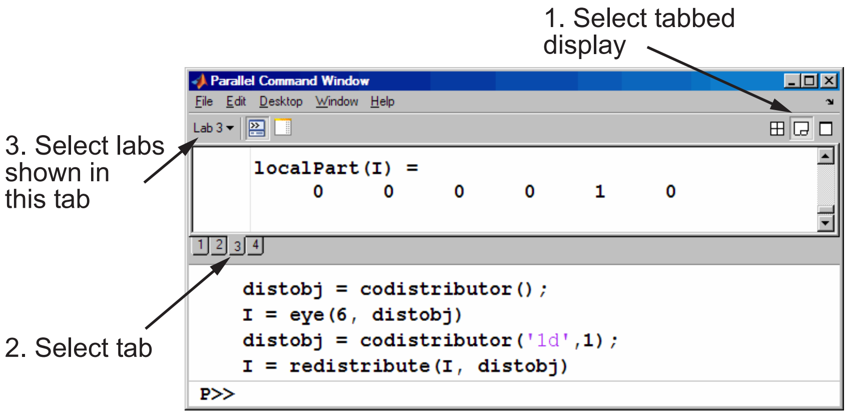
You can control the relative positions of the command window and the worker output. The following figure shows how to set the output to display beside the input, rather than above it.



```
distob
I = ey
distob
I = re
```

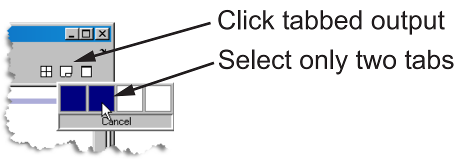
```
localPart(I) =
     1     0     0     0     0     0
     0     1     0     0     0     0

localPart(I) =
     0     0     1     0     0     0
     0     0     0     1     0     0

localPart(I) =
     0     0     0     0     1     0

localPart(I) =
     0     0     0     0     0     1
P>>
```

You can choose to view the worker outputs by tabs.

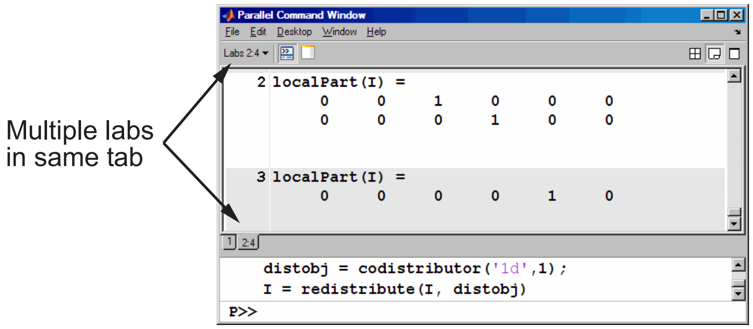


```
localPart(I) =
     0     0     0     0     1     0

distobj = codistributor();
I = eye(6, distobj)
distobj = codistributor('1d',1);
I = redistribute(I, distobj)
P>>
```

You can have multiple workers send their output to the same tile or tab. This allows you to have fewer tiles or tabs than workers.



In this case, the window provides shading to help distinguish the outputs from the various workers.



```
  2 localPart(I) =
         0     0     1     0     0     0
         0     0     0     1     0     0

  3 localPart(I) =
         0     0     0     0     1     0
```

```
    distobj = codistributor('1d',1);
    I = redistribute(I, distobj)
P>>
```

Running pmode Interactive Jobs on a Cluster

When you run pmode on a cluster of workers, you are running a job that is much like any other parallel job, except it is interactive. The cluster can be heterogeneous, but with certain limitations described at `http://www.mathworks.com/products/parallel-computing/requirements.html`; carefully locate your scheduler on that page and note that pmode sessions run as jobs described as "parallel applications that use inter-worker communication."

Many of the job's properties are determined by the cluster profile. For more details about creating and using profilies, see "Cluster Profiles" on page 6-12.

The general form of the command to start a pmode session is

```
pmode start <profile-name> <num-workers>
```

where `<profile-name>` is the name of the cluster prifile you want to use, and `<num-workers>` is the number of workers you want to run the pmode job on. If `<num-workers>` is omitted, the number of workers is determined by the profile. Coordinate with your system administrator when creating or using a profile.

If you omit `<profile-name>`, pmode uses the default profile (see the `parallel.defaultClusterProfile` reference page).

For details on all the command options, see the `pmode` reference page.

Plotting Distributed Data Using pmode

Because the workers running a job in pmode are MATLAB sessions without displays, they cannot create plots or other graphic outputs on your desktop.

When working in pmode with codistributed arrays, one way to plot a codistributed array is to follow these basic steps:

1. Use the `gather` function to collect the entire array into the workspace of one worker.
2. Transfer the whole array from any worker to the MATLAB client with `pmode lab2client`.
3. Plot the data from the client workspace.

The following example illustrates this technique.

Create a 1-by-100 codistributed array of 0s. With four labs (workers), each has a 1-by-25 segment of the whole array.

```
P>> D = zeros(1,100,codistributor1d())

  Lab 1: This lab stores D(1:25).
  Lab 2: This lab stores D(26:50).
  Lab 3: This lab stores D(51:75).
  Lab 4: This lab stores D(76:100).
```

Use a `for`-loop over the distributed range to populate the array so that it contains a sine wave. Each worker does one-fourth of the array.

```
P>> for i = drange(1:100)
D(i) = sin(i*2*pi/100);
end;
```

Gather the array so that the whole array is contained in the workspace of lab 1.

```
P>> P = gather(D, 1);
```

Transfer the array from the workspace of worker 1 to the MATLAB client workspace, then plot the array from the client. Note that both commands are entered in the MATLAB (client) Command Window.

```
pmode lab2client P 1
plot(P)
```

This is not the only way to plot codistributed data. One alternative method, especially useful when running noninteractive parallel jobs, is to plot the data to a file, then view it from a later MATLAB session.

pmode Limitations and Unexpected Results

Using Graphics in pmode

Displaying a GUI

The workers that run the tasks of a parallel job are MATLAB sessions without displays. As a result, these workers cannot display graphical tools and so you cannot do things like plotting from within pmode. The general approach to accomplish something graphical is to transfer the data into the workspace of the MATLAB client using

```
pmode lab2client var labindex
```

Then use the graphical tool on the MATLAB client.

Using Simulink Software

Because the workers running a pmode job do not have displays, you cannot use Simulink software to edit diagrams or to perform interactive simulation from within pmode. If you type `simulink` at the pmode prompt, the Simulink Library Browser opens in the background on the workers and is not visible.

You can use the `sim` command to perform noninteractive simulations in parallel. If you edit your model in the MATLAB client outside of pmode, you must save the model before accessing it in the workers via pmode; also, if the workers had accessed the model previously, they must close and open the model again to see the latest saved changes.

pmode Troubleshooting

In this section...

Connectivity Testing

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 "Admin Center" in the MATLAB Distributed Computing Server documentation.

Hostname Resolution

If a worker cannot resolve the hostname of the computer running the MATLAB client, use `pctconfig` to change the hostname by which the client machine advertises itself.

Socket Connections

If a worker cannot open a socket connection to the MATLAB client, try the following:

- Use `pctconfig` to change the hostname by which the client machine advertises itself.
- Make sure that firewalls are not preventing communication between the worker and client machines.
- Use `pctconfig` to change the client's `pmodeport` property. This determines the port that the workers will use to contact the client in the next pmode session.

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, parallel jobs, and pmode. The sections are as follows.

Nondistributed Versus Distributed Arrays

In this section...

Introduction

All 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; but not function handles or user-defined objects. 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 (lab) 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 5-2
- "Variant Arrays" on page 5-3
- "Private Arrays" on page 5-4

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

  LAB 1         LAB 2         LAB 3         LAB 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

  LAB 1         LAB 2         LAB 3         LAB 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
  LAB 1          LAB 2           LAB 3          LAB 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 (labs). 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

   LAB 1          LAB 2       LAB 3      LAB 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 5-5.

Working with Codistributed Arrays

In this section...

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 (lab). 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);
    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).
end
```

Each worker (lab) 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 = codistributed.eye(8)
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 a 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 5-15 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 5-8 — 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 5-9 — 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 requiremenets as it lets you build a codistributed array from smaller pieces.
- "Using MATLAB Constructor Functions" on page 5-10 — Use any of the MATLAB constructor functions like `rand` or `zeros` with the 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 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 (labs) 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

     LAB 1       |       LAB 2                LAB 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.

```
codistributed.cell(m, n, ..., codist)
codistributed.colon(a, d, b)
codistributed.eye(m, ..., classname, codist)
codistributed.false(m, n, ..., codist)
codistributed.Inf(m, n, ..., classname, codist)
codistributed.linspace(m, n, ..., codist)
codistributed.logspace(m, n, ..., codist)
codistributed.NaN(m, n, ..., classname, codist)
codistributed.ones(m, n, ..., classname, codist)
codistributed.rand(m, n, ..., codist)
codistributed.randn(m, n, ..., codist)
sparse(m, n, codist)
```

```
codistributed.speye(m, ..., codist)
codistributed.sprand(m, n, density, codist)
codistributed.sprandn(m, n, density, codist)
codistributed.true(m, n, ..., codist)
codistributed.zeros(m, n, ..., classname, 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` function to perform the reverse operation. This function, described in “Building from Smaller Arrays” on page 5-9, 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
        Lab 1      |     Lab 2      |   Lab 3    |   Lab 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 = distributed.rand(1,1e6); %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, parallel job, or pmode. This example uses pmode for a visual display.

```
P>> A = reshape(1:64, 8, 8)
```

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
```

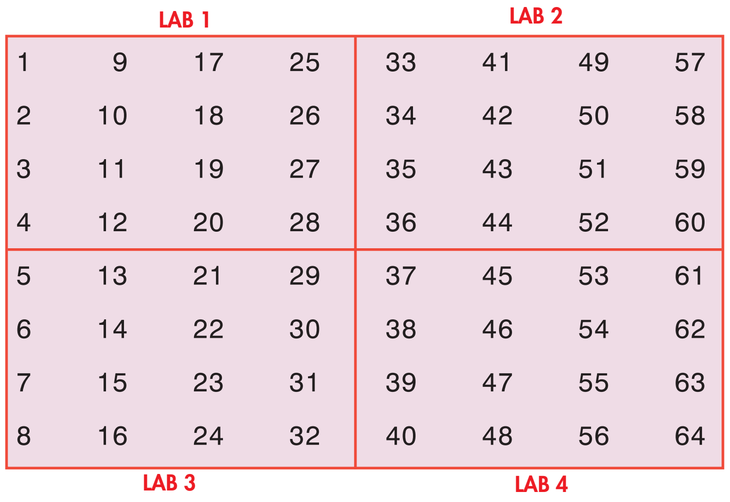
Suppose you want to distribute this array among four workers, with a 4-by-4 block as the local part on each worker. In this case, the lab grid is a 2-by-2 arrangement of the workers, and the block size is a square of four elements on a side (i.e., each block is a 4-by-4 square). With this information, you can define the codistributor object:

```
P>> DIST = codistributor2dbc([2 2], 4)
```

Now you can use this codistributor object to distribute the original matrix:

```
P>> AA = codistributed(A, DIST)
```

This distributes the array among the workers according to this scheme:



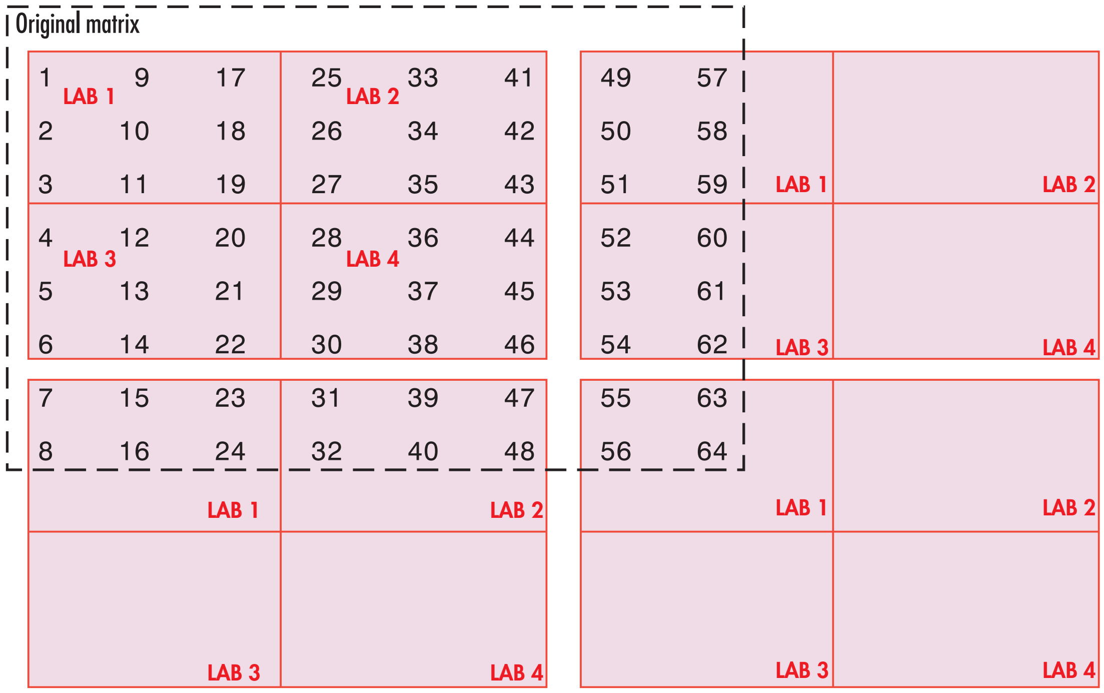
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

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.

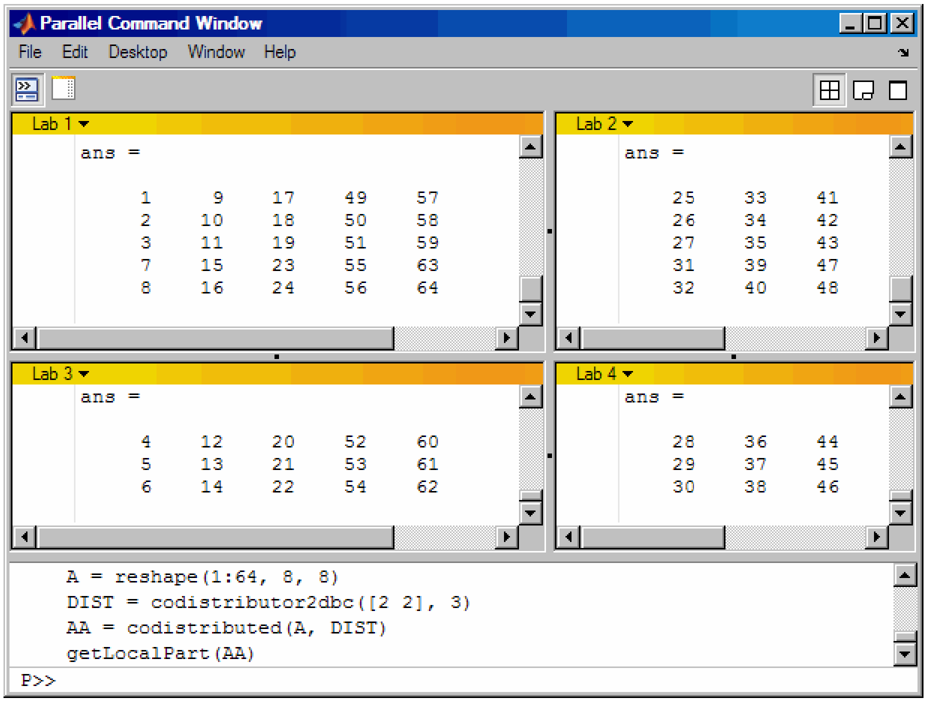
Using the same original 8-by-8 matrix and 2-by-2 lab grid, consider a block size of 3 instead of 4, so that 3-by-3 square blocks are distributed among the workers. The code looks like this:

```
P>> DIST = codistributor2dbc([2 2], 3)
P>> AA = codistributed(A, DIST)
```

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 pmode session shows the code and resulting distribution of data to each of the workers:



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
```

```
    A = reshape(1:64, 8, 8)
    DIST = codistributor2dbc([2 2], 3)
    AA = codistributed(A, DIST)
    getLocalPart(AA)
P>>
```

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.

Using a for-Loop Over a Distributed Range (for-drange)

In this section...

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 parallel job). For most applications involving parallel `for`-loops you should first try using `parfor` loops. See Chapter 2, "Parallel for-Loops (parfor)".

Parallelizing a for-Loop

If you already have a coarse-grained application to perform, but you do not want to bother with the overhead of defining jobs and tasks, you can take advantage of the ease-of-use that pmode 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 pmode, or in a parallel 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 parallel 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.

At the pmode prompt, 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 `1`s 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 MATLAB pool of 4 workers:

```
spmd
  PP = codistributed.zeros(6,8,12);
  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
```

Using MATLAB Functions on Codistributed Arrays

Many functions in MATLAB software are enhanced or overloaded so that they operate on codistributed arrays in much the same way that they operate on arrays contained in a single workspace.

In most cases, if any of the input arguments to these functions is a distributed or codistributed array, their output arrays are distributed or codistributed, respectively. If the output is always scalar, it is replicated on each worker. All these overloaded functions with codistributed array inputs must reference the same inputs at the same time on all workers; therefore, you cannot use variant arrays for input arguments.

A few of these functions might exhibit certain limitations when operating on a codistributed array. To see if any function has different behavior when used with a codistributed array, type

```
help codistributed/functionname
```

For example,

```
help codistributed/normest
```

The following table lists the enhanced MATLAB functions that operate on codistributed arrays.

Type of Function	Function Names
Data functions	`cumprod`, `cumsum`, `fft`, `max`, `min`, `prod`, `sum`
Data type functions	`arrayfun`, `cast`, `cell2mat`, `cell2struct`, `celldisp`, `cellfun`, `char`, `double`, `fieldnames`, `int16`, `int32`, `int64`, `int8`, `logical`, `num2cell`, `rmfield`, `single`, `struct2cell`, `swapbytes`, `typecast`, `uint16`, `uint32`, `uint64`, `uint8`

Type of Function	Function Names
Elementary and trigonometric functions	abs, acos, acosd, acosh, acot, acotd, acoth, acsc, acscd, acsch, angle, asec, asecd, asech, asin, asind, asinh, atan, atan2, atand, atanh, ceil, complex, conj, cos, cosd, cosh, cot, cotd, coth, csc, cscd, csch, exp, expm1, fix, floor, hypot, imag, isreal, log, log10, log1p, log2, mod, nextpow2, nthroot, pow2, real, reallog, realpow, realsqrt, rem, round, sec, secd, sech, sign, sin, sind, sinh, sqrt, tan, tand, tanh
Elementary matrices	cat, diag, eps, find, isempty, isequal, isequaln, isfinite, isinf, isnan, length, meshgrid, ndgrid, ndims, numel, reshape, size, sort, tril, triu
Matrix functions	chol, eig, inv, lu, norm, normest, qr, svd
Array operations	all, and (&), any, bitand, bitor, bitxor, ctranspose ('), end, eq (==), ge (>=), gt (>), horzcat ([]), ldivide (.\), le (<=), lt (<), minus (-), mldivide (\), mrdivide (/), mtimes (*), ne (~=), not (~), or (|), plus (+), power (.^), rdivide (./), subsasgn, subsindex, subsref, times (.*), transpose (.'), uminus (-), uplus (+), vertcat ([;]), xor
Sparse matrix functions	full, issparse, nnz, nonzeros, nzmax, sparse, spfun, spones
Special functions	dot

Programming Overview

This chapter provides information you need for programming with Parallel Computing Toolbox software. Further details of evaluating functions in a cluster, programming distributed jobs, and programming parallel 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

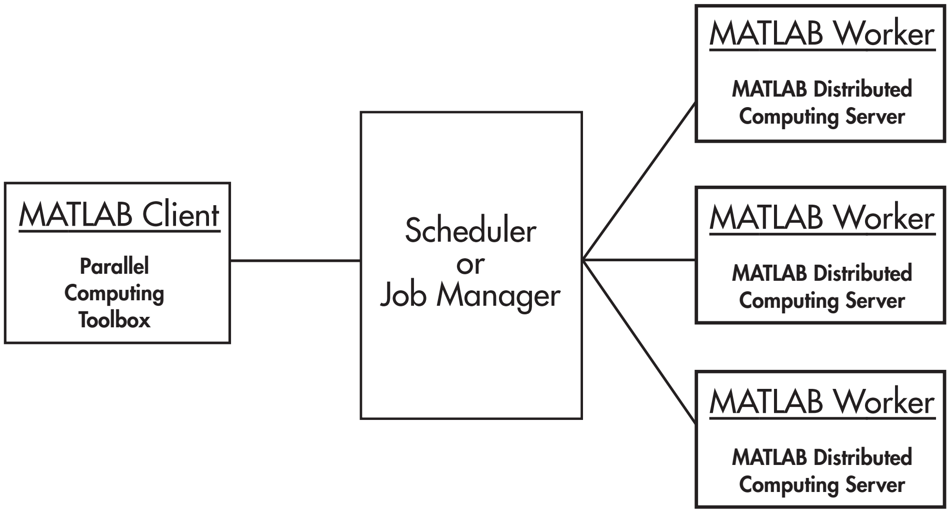
Parallel Computing Toolbox and MATLAB Distributed Computing Server software let you solve computationally and data-intensive problems using MATLAB and Simulink on multicore and multiprocessor computers. Parallel processing constructs such as parallel for-loops and code blocks, distributed arrays, parallel 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 Distributed Computing Server software is the product that performs the execution of your job on a cluster of machines.

The *MATLAB job scheduler* (MJS) is the process that coordinates the execution of jobs and the evaluation of their tasks. The MJS distributes the tasks for evaluation to the server's individual MATLAB sessions called *workers*. Use of the MJS 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®.

See the “Glossary” on page Glossary-1 for definitions of the parallel computing terms used in this manual.


MATLAB Worker
MATLAB Distributed Computing Server
MATLAB Client
Parallel Computing Toolbox
Scheduler or Job Manager
MATLAB Worker
MATLAB Distributed Computing Server
MATLAB Worker
MATLAB Distributed Computing Server

Basic Parallel Computing Setup

Toolbox and Server Components

- “MJS, Workers, and Clients” on page 6-3
- “Local Scheduler” on page 6-5
- “Third-Party Schedulers” on page 6-5
- “Components on Mixed Platforms or Heterogeneous Clusters” on page 6-7
- “mdce Service” on page 6-7
- “Components Represented in the Client” on page 6-7

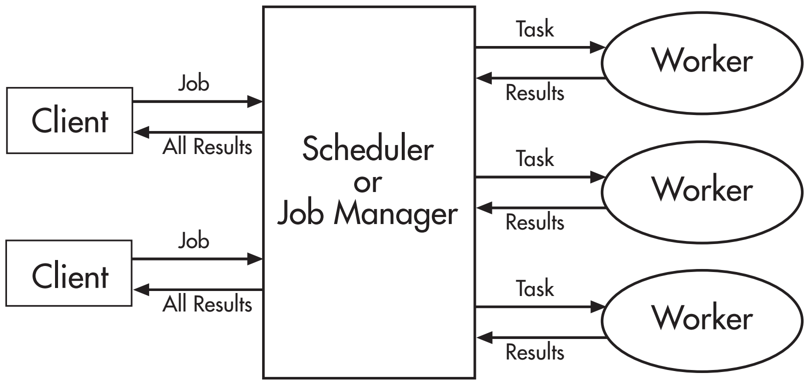
MJS, Workers, and Clients

The MJS can be run on any machine on the network. The MJS 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 MJS, executes the task, returns the result to the MJS, and then is given another task. When all tasks for a running job have been assigned to workers, the MJS starts running the next job on the next available worker.

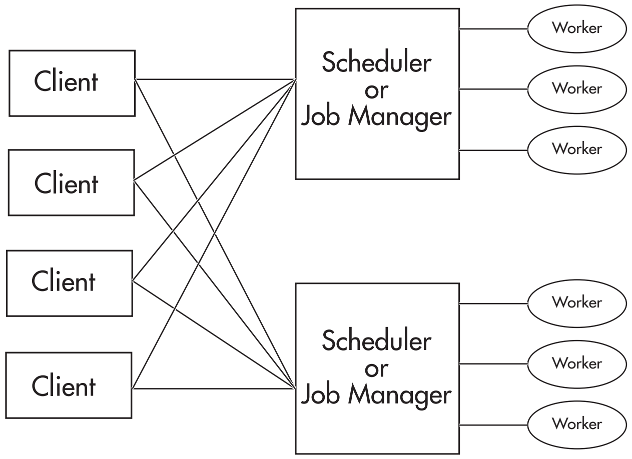
A MATLAB Distributed Computing Server software setup usually includes many workers that can all execute tasks simultaneously, speeding up execution of large MATLAB jobs. It is generally not important which worker executes a specific task. In an independent job, the workers evaluate tasks one at a time as available, perhaps simultaneously, perhaps not, returning the results to the MJS. In a communicating job, the workers evaluate tasks simultaneously. The MJS 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 MJS host. You can also have more than one worker session or more than one MJS session on a machine.



Interactions of Parallel Computing Sessions

A large network might include several MJSs as well as several client sessions. Any client session can create, run, and access jobs on any MJS, but a worker session is registered with and dedicated to only one MJS at a time. The following figure shows a configuration with multiple MJSs.



Cluster with Multiple Clients and MJSs

Local Scheduler

A feature of Parallel Computing Toolbox software is the ability to run a local scheduler and a cluster of up to twelve workers on the client machine, so that you can run distributed and parallel jobs without requiring a remote cluster or MATLAB Distributed Computing 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 distributed or parallel applications before running them on your cluster.

Third-Party Schedulers

As an alternative to using the MJS, 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 an MJS. You should consider the following when deciding to use a third-party scheduler or the MATLAB job scheduler (MJS) 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 an MJS, 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 MJS 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 MJS 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 an MJS, worker processes usually remain running at all times, dedicated to their MJS. 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 MJS.

- 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 MJS 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 Distributed Computing Server software are supported on Windows®, UNIX®, and Macintosh® operating systems. Mixed platforms are supported, so that the clients, MJS, and workers do not have to be on the same platform. The cluster can also be comprised of both 32-bit and 64-bit machines, so long as your data does not exceed the limitations posed by the 32-bit systems. Other limitations are described at `http://www.mathworks.com/products/parallel-computing/requirements.html`.

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.

mdce Service

If you are using the MJS, every machine that hosts a worker or MJS session must also run the mdce service.

The mdce service controls the worker and MJS sessions and recovers them when their host machines crash. If a worker or MJS machine crashes, when the mdce service starts up again (usually configured to start at machine boot time), it automatically restarts the MJS and worker sessions to resume their sessions from before the system crash. These processes are covered more fully in the MATLAB Distributed Computing Server System Administrator's Guide.

Components Represented in the Client

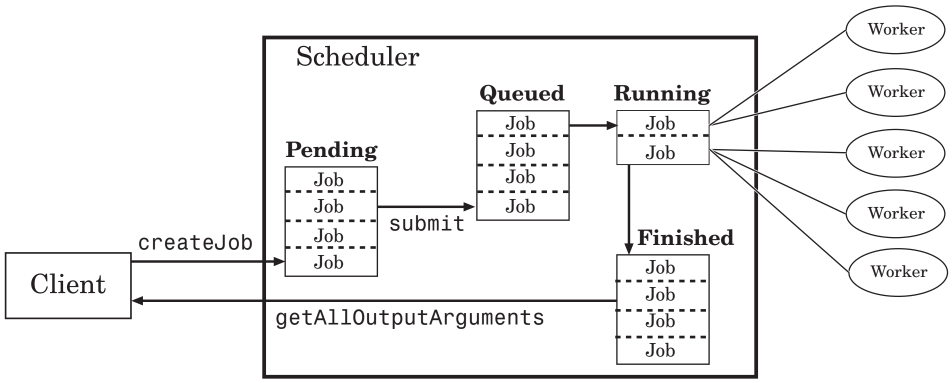
A client session communicates with the MJS by calling methods and configuring properties of an *MJS cluster object*. Though not often necessary, the client session can also access information about a worker session through a *worker object*.

When you create a job in the client session, the job actually exists in the MJS 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 MJS 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 MJS (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 `getAllOutputArguments`.



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 MJS 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.

Job Stage	**Description**
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 `getAllOutputArguments`.
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.
Destroyed	When a job's data has been removed from its data location or from the MJS, the state of the job in the client is `destroyed`. 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 MJS's `JobStorageLocation` folder, even if you clear all the objects from the client session. The MJS or scheduler keeps all the jobs it has executed, until you restart the MJS in a clean state. Therefore, you can retrieve information from a job later or in another client session, so long as the MJS has not been restarted with the `-clean` option.

To permanently remove completed jobs from the MJS or scheduler's storage location, use the `delete` function or the Job Monitor GUI.

Create Simple Independent Jobs

Program a Basic Job with 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 8-3.)

   ```
   parallel.defaultClusterProfile('local');
   c = parcluster();
   ```

2 Create a job. Create job `j` on the cluster. (For more information, see "Create a Job" on page 8-3.)

   ```
   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 8-5.)

   ```
   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 scheduler actually starts a MATLAB worker session for each task, up to twelve at one time. (For more information, see "Submit a Job to the Cluster" on page 8-5.)

```
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's Results" on page 8-6.)

   ```
   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)
   ```

Cluster Profiles

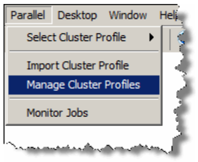
In this section...

Define Cluster Profiles

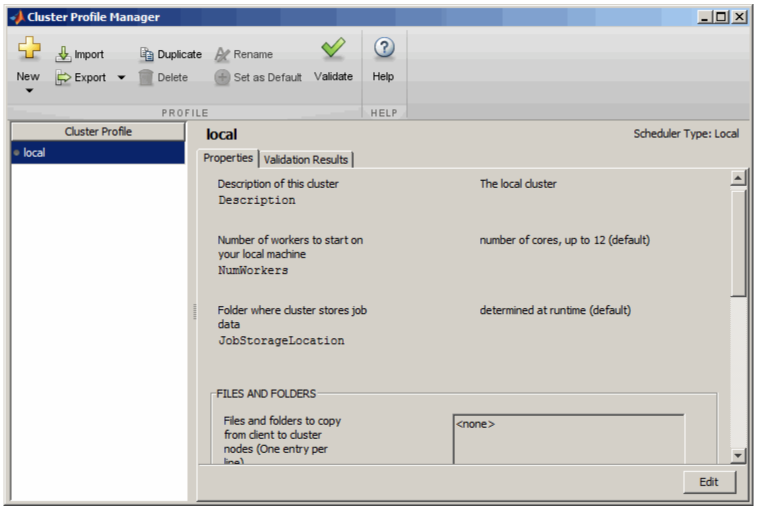
Cluster Profiles let you define certain properties for your cluster, then have these properties applied when you create cluster, job, and task objects in the MATLAB client. Some of the functions that support the use of cluster profiles are

- `batch`
- `matlabpool`
- `parcluster`
- `pmode`

You create and modify profiles through the Cluster Profile Manager. You access the Cluster Profile Manager using the **Parallel** pull-down menu on the MATLAB desktop. To open the Cluster Profile Manager, select **Parallel > Manage Cluster Profiles**.



The first time you open the Cluster Profile Manager, it lists only one profile called `local`, which at first is the default profile and has only default settings.



local

Scheduler Type: Local

Description of this cluster
Description
The local cluster

Number of workers to start on your local machine
NumWorkers
number of cores, up to 12 (default)

Folder where cluster stores job data
JobStorageLocation
determined at runtime (default)

FILES AND FOLDERS

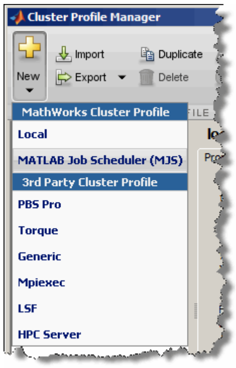
Files and folders to copy from client to cluster nodes (One entry per line)
<none>

The following example provides instructions on how to create and modify profiles using the Cluster Profile Manager.

Create and Modify Cluster Profiles

Suppose you want to create a profile to set several properties for some jobs being run in an MJS cluster. The following example illustrates a possible workflow, where you create two profiles differentiated only by the number of workers.

1 In the Cluster Profile Manager, select **New > MATLAB Job Scheduler (MJS)**. This specifies that you want a new profile whose type of cluster is an MJS.



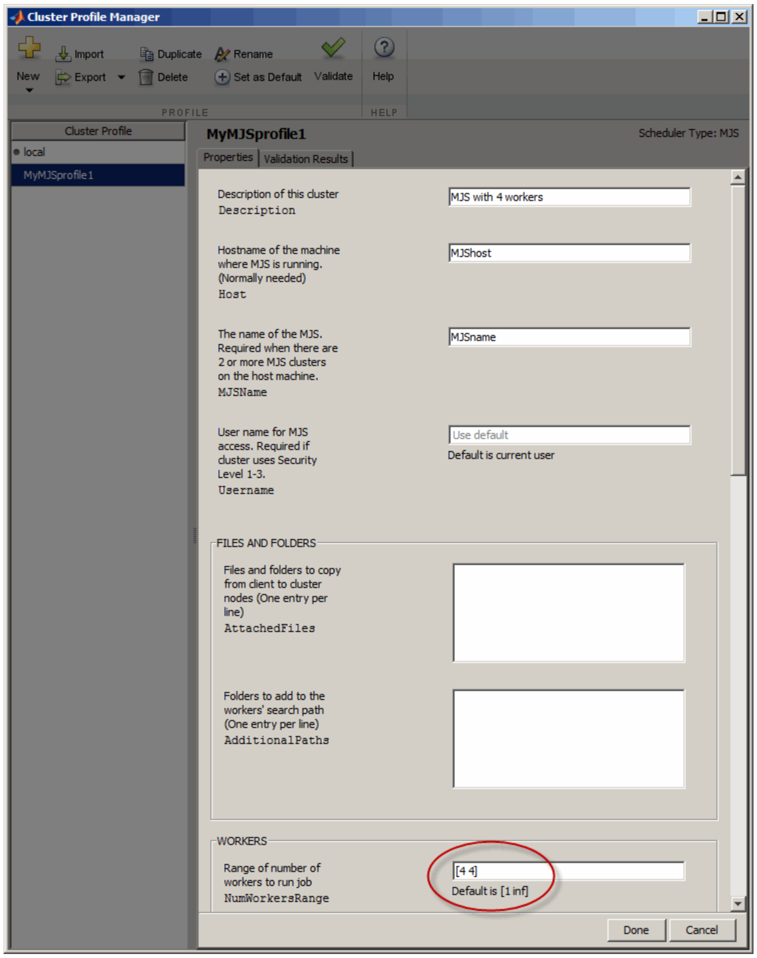
This creates and displays a new profile, called Profile1.

2 Double-click the profile name in the listing, and enter a profile name `MyMJSprofile1`.

3 Click **Edit** in the lower-right corner 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 MJS is running and the name of the MJS. If you are entering information for an actual MJS already running on your network, enter the appropriate text. If you are unsure about MJS (also known as a job manager) names and locations on your network, ask your system administrator for help.

4 Scroll down to the Workers section, and for the Range of number of workers, enter `[4 4]`. This specifies that jobs using this profile require at least four workers and no more than four workers. Therefore, the job runs on exactly four workers, even if it has to wait until four workers are available before starting.



MyMJSprofile1

Scheduler Type: MJS

Description of this cluster `Description`	MJS with 4 workers
Hostname of the machine where MJS is running. (Normally needed) `Host`	MJShost
The name of the MJS. Required when there are 2 or more MJS clusters on the host machine. `MJSName`	MJSname
User name for MJS access. Required if cluster uses Security Level 1-3. `Username`	Use default Default is current user

FILES AND FOLDERS

Files and folders to copy from client to cluster nodes (One entry per line) `AttachedFiles`	
Folders to add to the workers' search path (One entry per line) `AdditionalPaths`	

WORKERS

Range of number of workers to run job `NumWorkersRange`	[4 4] Default is [1 inf]

You might want to edit other properties depending on your particular situation.

5 Click **Done** to save the profile.

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:

1. In the Cluster Profile Manager, right-click the profile `MyMJSprofile1` in the list and select **Duplicate**.

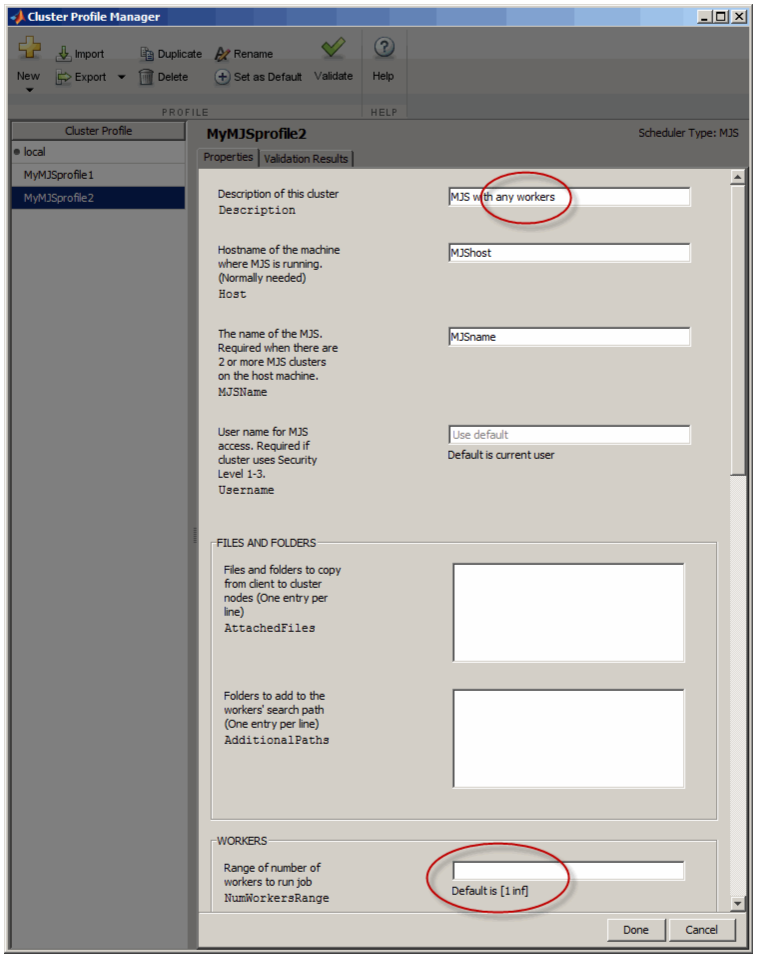
   The duplicate profile is created with a name based on the original profile name prefixed with `Copy_Of_`.

2. Double-click the new profile name and edit its name to be `MyMJSprofile2`.

3. Click **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.



MyMJSprofile2

Scheduler Type: MJS

Description of this cluster
`Description`: MJS with any workers

Hostname of the machine where MJS is running. (Normally needed)
`Host`: MJShost

The name of the MJS. Required when there are 2 or more MJS clusters on the host machine.
`MJSName`: MJSname

User name for MJS access. Required if cluster uses Security Level 1-3.
`Username`: Use default
Default is current user

FILES AND FOLDERS

Files and folders to copy from client to cluster nodes (One entry per line)
`AttachedFiles`

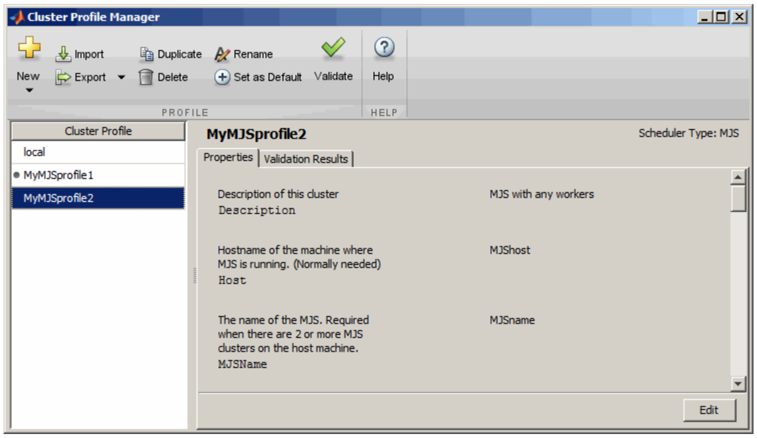
Folders to add to the workers' search path (One entry per line)
`AdditionalPaths`

WORKERS

Range of number of workers to run job
`NumWorkersRange`
Default is [1 inf]

6 Click **Done** to save the profile and close the properties editor.

You now have two profiles that differ only in the number of workers required for running a job.



MyMJSprofile2 Scheduler Type: MJS

Description of this cluster `Description`	MJS with any workers
Hostname of the machine where MJS is running. (Normally needed) `Host`	MJShost
The name of the MJS. Required when there are 2 or more MJS clusters on the host machine. `MJSName`	MJSname

When creating a job, you can apply either profile to that job as a way of specifying how many workers it should run on.

For examples of profiles for different kinds of supported schedulers, see the MATLAB Distributed Computing Server installation instructions on the web at mathworks.com/distconfig.

Validate Profiles

The Cluster Profile Manager includes a tool for validating profiles.

To validate a profile, follow these steps:

1. Open the Cluster Profile Manager by selecting on the desktop **Parallel > Manage Cluster Profiles**.
2. In the Cluster Profile Manager, click the name of the profile you want to test. Note that you can highlight a profile this way without changing the selected default profile. So a profile selected for validation does not need to be your default profile.
3. Click **Validate**.

Profile validation includes five stages:

1. Connects to the cluster (`parcluster`)

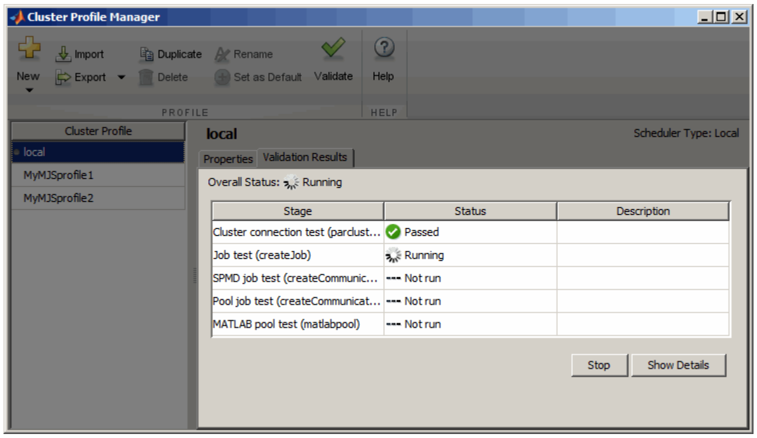
2 Runs an independent job (`createJob`) on the cluster using the profile

3 Runs an SPMD-type communicating job on the cluster using the profile

4 Runs a pool-type communicating job on the cluster using the profile

5 Runs a MATLAB pool job on the cluster using the profile

While the tests are running, the Cluster Profile Manager displays their progress as shown here:



Overall Status: Running

Stage	Status	Description
Cluster connection test (parclust…	✓ Passed	
Job test (createJob)	Running	
SPMD job test (createCommunic…	--- Not run	
Pool job test (createCommunicat…	--- Not run	
MATLAB pool test (matlabpool)	--- Not run	

Note You cannot run a profile validation if you already have a MATLAB pool open.

Note When using an mpiexec scheduler, a failure is expected for the `Independent Job` stage. It is normal for the test then to proceed to the `Communicating Job` and `Matlabpool` stages.

Click **Show Details** to get more information about test results. This information includes any error messages, debug logs, and other data that might be useful in diagnosing problems or helping to determine proper network settings.

The Validation Results tab keeps the test results available until the current MATLAB session closes.

Apply Cluster Profiles in Client Code

In the MATLAB client where you create and define your parallel computing objects, you can use cluster profiles when creating the objects.

Select a Default Cluster Profile

Some functions support default profiles, so that if you do not specify a profile for them to use, they automatically apply the default. There are several ways to specify which of your profiles should be used as the default profile:

- In the MATLAB desktop, click **Parallel > Select Cluster Profile**, and from there, all your profiles are available. The current default profile appears with a dot next to it. You can select any profile in the list as the default.
- In the Cluster Profile Manager, a dot indicates which is currently the default profile. You can select any profile in the list, then click **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.defaultProfile('MyMJSprofile1')
  matlabpool open
  ```

  or

  ```
  matlabpool open MyMJSprofile1
  ```

Create Cluster Object

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('myMJSprofile')
```

This command finds the cluster defined by the settings of the profile named `myMJSprofile` and sets property values on the cluster object based on settings in the profile. By applying different profiles, you can alter your cluster choices without changing your MATLAB application code.

Create Jobs and Tasks

Because the properties of cluster, job, and task objects can be defined in a profile, you do not have to explicitly define them in your application. Therefore, your code can accommodate any type of cluster without being modified. For example, the following code uses one profile to set properties on cluster, job, and task objects:

```
c = parcluster('myProfile1);
job1 = createJob(c); % Uses profile of cluster c.
createTask(job1,@rand,1,{3}) % Uses profile of c.
```

Export and Import 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 `.settings` 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 Profile Clusters Manager, select (highlight) the profile you want to export.

2. Click **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, click **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 `.settings` extension appended; you can alter the names if you want to.

Profiles saved in this way can then be imported by other MATLAB users:

1 In the Cluster Profile Manager, click **Import**.

2 In the Import profiles from file dialog box, browse to find the `.settings` file for the profile you want to import. Select the file and click **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 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 "Deploying Applications Created Using Parallel Computing Toolbox" in the MATLAB Compiler documentation.

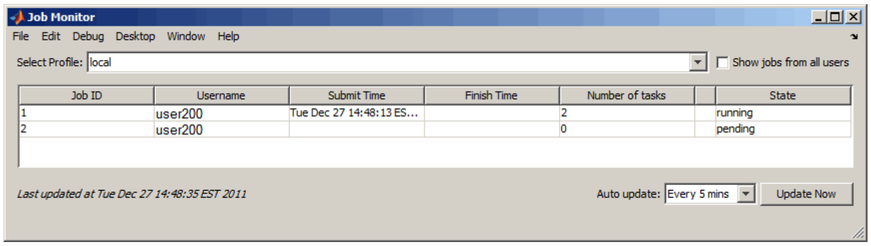
A compiled application has the same default profile and the same list of alternative profiles that the compiling user had when the application was compiled. This means that in many cases the profile file is not needed, as might be the case when using the `local` profile for local workers. If an exported 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).

Job Monitor

In this section...

Job Monitor GUI

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 by selecting **Parallel > Monitor Jobs**.



Select Profile: local

Show jobs from all users

Job ID	Username	Submit Time	Finish Time	Number of tasks		State
1	user200	Tue Dec 27 14:48:13 ES...		2		running
2	user200			0		pending

Last updated at Tue Dec 27 14:48:35 EST 2011

Auto update: Every 5 mins

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.

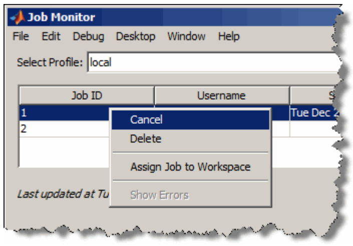
Typical Use Cases

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
- Clean up old jobs you no longer need
- Create a job object in MATLAB for access to a 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:



Select Profile: local

Job ID	Username	S
1		Tue Dec 2
2		

Last updated at Tu

- **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 jobs data and removes it from the queue. This is the same as the command-line `delete` function for the job.
- **Assign Job to Workspace** — This creates a job object in the MATLAB workspace so that you can access the job and its properties from the command line. This is accomplished by the `findJob` command, which is reflected in the command window.
- **Show Errors** — This displays all the tasks that generated an error in that job, with their error properties.

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 for Improving Performance" in the MATLAB documentation.

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 pmode to develop parallel functionality.** Use pmode with the local scheduler to develop your functions on several workers in parallel. As you progress and use pmode 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 (labs) 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 `mdce_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\MDCE\Checkpoint`, the starting current directory for that worker session is

```
C:\TEMP\MDCE\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 an MJS, 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 MATLAB 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 MJS. For information on recreating these objects in the client session, see "Recover Objects" on page 8-18.

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 "Object Data Size Limitations" on page 6-44.

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.

Profiling Parallel Code

In this section...

Introduction

The parallel profiler provides an extension of the `profile` command and the profile viewer specifically for communicating jobs, 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 (labs). Before using the parallel profiler, familiarize yourself with the standard profiler and its views, as described in "Profiling for Improving Performance".

Note The parallel profiler works on communicating jobs, including inside pmode. It does not work on `parfor`-loops.

Collecting Parallel Profile Data

For parallel profiling, you use the `mpiprofile` command within your communicating job (often within pmode) in a similar way to how you use `profile`.

To turn on the parallel profiler to start collecting data, enter the following line in your communicating job task code file, or type at the pmode prompt in the Parallel Command Window:

```
mpiprofile on
```

Now the profiler is collecting 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

With the parallel profiler on, you can proceed to execute your code while the profiler collects the data.

In the pmode Parallel Command Window, to find out if the profiler is on, type:

```
P>> mpiprofile status
```

For a complete list of options regarding profiler data details, clearing data, etc., see the `mpiprofile` reference page.

Viewing Parallel Profile Data

To open the parallel profile viewer from pmode, type in the Parallel Command Window:

```
P>> mpiprofile viewer
```

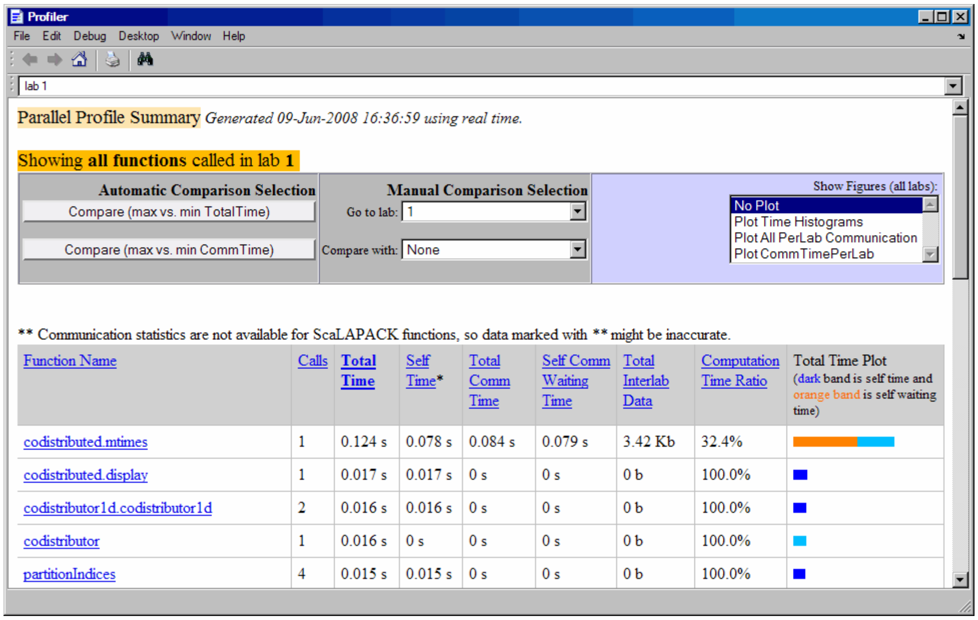
The remainder of this section is an example that illustrates some of the features of the parallel profile viewer. This example executes in a pmode session running on four local workers. Initiate pmode by typing in the MATLAB Command Window:

```
pmode start local 4
```

When the Parallel Command Window (pmode) starts, type the following code at the pmode prompt:

```
P>> R1 = rand(16, codistributor())
P>> R2 = rand(16, codistributor())
P>> mpiprofile on
P>> P = R1*R2
P>> mpiprofile off
P>> mpiprofile viewer
```

The last command opens the Profiler window, first showing the Parallel Profile Summary (or function summary report) for worker (lab) 1.


Profiler
File Edit Debug Desktop Window Help
lab 1
Compare (max vs. min TotalTime)
Compare (max vs. min CommTime)
No Plot
Plot Time Histograms
Plot All PerLab Communication
Plot CommTimePerLab

Parallel Profile Summary *Generated 09-Jun-2008 16:36:59 using real time.*

Showing **all functions** called in lab **1**

Automatic Comparison Selection | **Manual Comparison Selection** Go to lab: 1 Compare with: None | Show Figures (all labs): No Plot

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

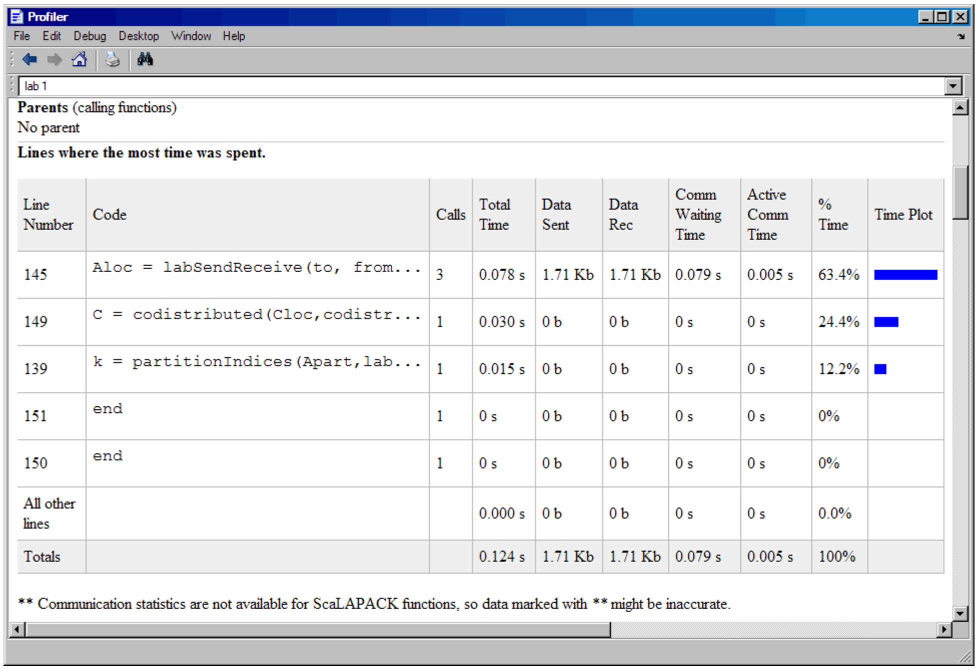
Function Name	Calls	**Total Time**	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Interlab Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
codistributed.mtimes	1	0.124 s	0.078 s	0.084 s	0.079 s	3.42 Kb	32.4%	🟧🟦
codistributed.display	1	0.017 s	0.017 s	0 s	0 s	0 b	100.0%	🟦
codistributor1d.codistributor1d	2	0.016 s	0.016 s	0 s	0 s	0 b	100.0%	🟦
codistributor	1	0.016 s	0 s	0 s	0 s	0 b	100.0%	🟦
partitionIndices	4	0.015 s	0.015 s	0 s	0 s	0 b	100.0%	🟦

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 subfunctions
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

Column Header	Description
Total Interlab 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

Click the name of any function in the list for more details about the execution of that function. The function detail report for `codistributed.mtimes` includes this listing:



Parents (calling functions)

No parent

Lines where the most time was spent.

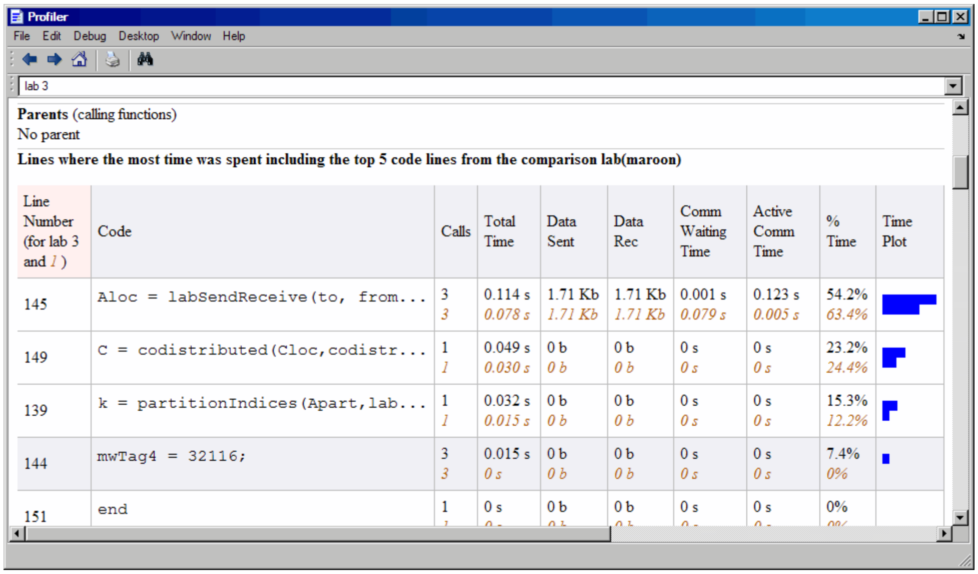
Line Number	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
145	`Aloc = labSendReceive(to, from...`	3	0.078 s	1.71 Kb	1.71 Kb	0.079 s	0.005 s	63.4%	▬▬▬
149	`C = codistributed(Cloc,codistr...`	1	0.030 s	0 b	0 b	0 s	0 s	24.4%	▬
139	`k = partitionIndices(Apart,lab...`	1	0.015 s	0 b	0 b	0 s	0 s	12.2%	■
151	`end`	1	0 s	0 b	0 b	0 s	0 s	0%	
150	`end`	1	0 s	0 b	0 b	0 s	0 s	0%	
All other lines			0.000 s	0 b	0 b	0 s	0 s	0.0%	
Totals			0.124 s	1.71 Kb	1.71 Kb	0.079 s	0.005 s	100%	

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

The code that is displayed in the report is taken 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**, allowing you to 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 (lab) 3 compared to worker (lab) 1 because these are the workers that spend the most versus least amount of time executing the code.

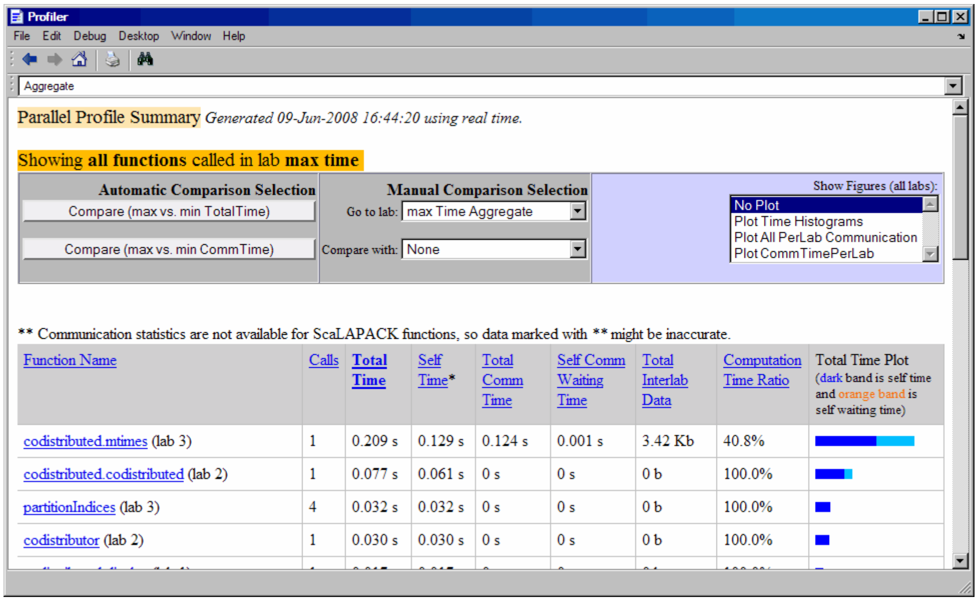


Parents (calling functions)
No parent

Lines where the most time was spent including the top 5 code lines from the comparison lab(maroon)

Line Number (for lab 3 and *1*)	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
145	`Aloc = labSendReceive(to, from...`	3 *3*	0.114 s *0.078 s*	1.71 Kb *1.71 Kb*	1.71 Kb *1.71 Kb*	0.001 s *0.079 s*	0.123 s *0.005 s*	54.2% *63.4%*	▬
149	`C = codistributed(Cloc,codistr...`	1 *1*	0.049 s *0.030 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	23.2% *24.4%*	▬
139	`k = partitionIndices(Apart,lab...`	1 *1*	0.032 s *0.015 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	15.3% *12.2%*	▬
144	`mwTag4 = 32116;`	3 *3*	0.015 s *0 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	7.4% *0%*	▪
151	`end`	1	0 s	0 b	0 b	0 s	0 s	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 09-Jun-2008 16:44:20 using real time.*

Showing **all functions** called in lab **max time**

Automatic Comparison Selection
Compare (max vs. min TotalTime)
Compare (max vs. min CommTime)

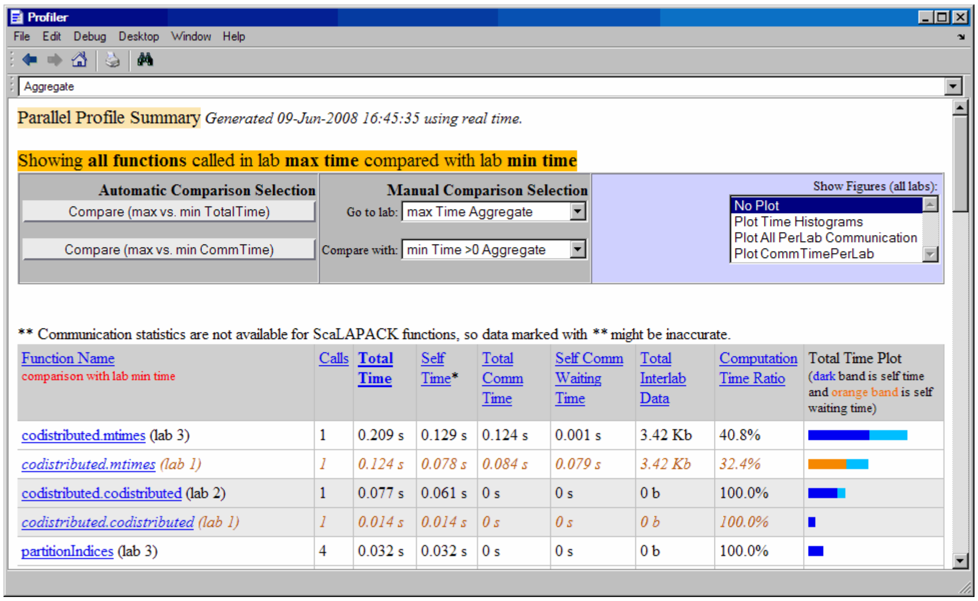
Manual Comparison Selection
Go to lab: max Time Aggregate
Compare with: None

Show Figures (all labs):
No Plot
Plot Time Histograms
Plot All PerLab Communication
Plot CommTimePerLab

** 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 Interlab Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
codistributed.mtimes (lab 3)	1	0.209 s	0.129 s	0.124 s	0.001 s	3.42 Kb	40.8%	
codistributed.codistributed (lab 2)	1	0.077 s	0.061 s	0 s	0 s	0 b	100.0%	
partitionIndices (lab 3)	4	0.032 s	0.032 s	0 s	0 s	0 b	100.0%	
codistributor (lab 2)	1	0.030 s	0.030 s	0 s	0 s	0 b	100.0%	

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 `codistributed.mtimes` from workers 3 and 1, because they spent the maximum and minimum times on this function. Similarly, other functions are listed.



Parallel Profile Summary *Generated 09-Jun-2008 16:45:35 using real time.*

Showing **all functions** called in lab **max time** compared with lab **min time**

Automatic Comparison Selection: Compare (max vs. min TotalTime) | Compare (max vs. min CommTime)

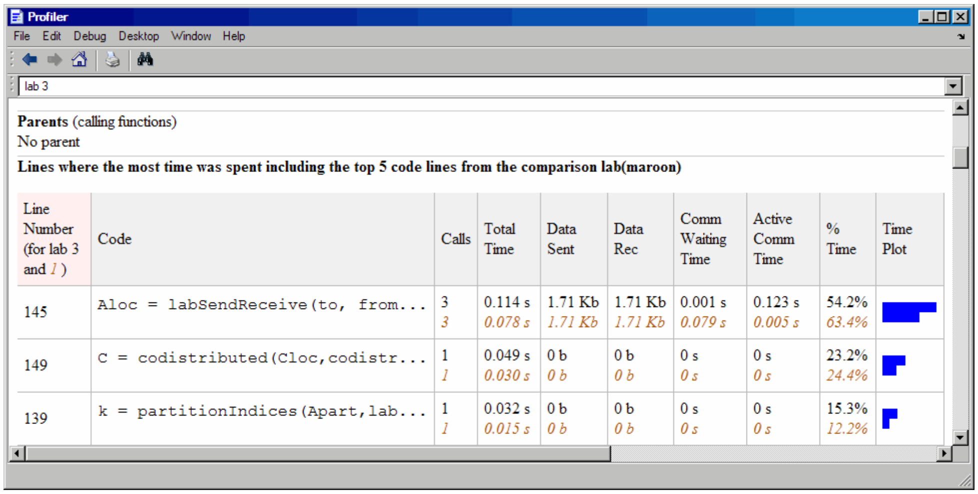
Manual Comparison Selection: Go to lab: max Time Aggregate | Compare with: min Time >0 Aggregate

Show Figures (all labs): No Plot | Plot Time Histograms | Plot All PerLab Communication | Plot CommTimePerLab

** Communication statistics are not available for ScaLAPACK functions, so data marked with ** might be inaccurate.

Function Name comparison with lab min time	Calls	**Total Time**	Self Time*	Total Comm Time	Self Comm Waiting Time	Total Interlab Data	Computation Time Ratio	Total Time Plot (dark band is self time and orange band is self waiting time)
codistributed.mtimes (lab 3)	1	0.209 s	0.129 s	0.124 s	0.001 s	3.42 Kb	40.8%	🟦🟦
codistributed.mtimes (lab 1)	*1*	*0.124 s*	*0.078 s*	*0.084 s*	*0.079 s*	*3.42 Kb*	*32.4%*	🟧🟦
codistributed.codistributed (lab 2)	1	0.077 s	0.061 s	0 s	0 s	0 b	100.0%	🟦
codistributed.codistributed (lab 1)	*1*	*0.014 s*	*0.014 s*	*0 s*	*0 s*	*0 b*	*100.0%*	🟦
partitionIndices (lab 3)	4	0.032 s	0.032 s	0 s	0 s	0 b	100.0%	🟦

Click on a function name in the summary listing of a comparison to get a detailed comparison. The detailed comparison for `codistributed.mtimes` looks like this, displaying line-by-line data from both workers:

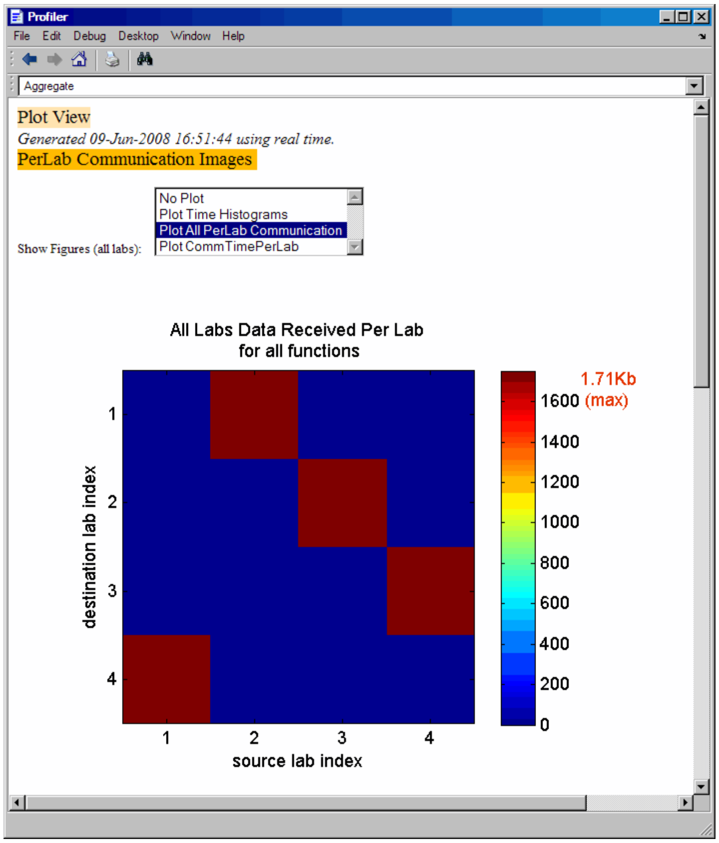


Parents (calling functions)
No parent

Lines where the most time was spent including the top 5 code lines from the comparison lab(maroon)

Line Number (for lab 3 and *1*)	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
145	`Aloc = labSendReceive(to, from...`	3 *3*	0.114 s *0.078 s*	1.71 Kb *1.71 Kb*	1.71 Kb *1.71 Kb*	0.001 s *0.079 s*	0.123 s *0.005 s*	54.2% *63.4%*	▬
149	`C = codistributed(Cloc,codistr...`	1 *1*	0.049 s *0.030 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	23.2% *24.4%*	▬
139	`k = partitionIndices(Apart,lab...`	1 *1*	0.032 s *0.015 s*	0 b *0 b*	0 b *0 b*	0 s *0 s*	0 s *0 s*	15.3% *12.2%*	▬

To see plots of communication data, select **Plot All PerLab 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.



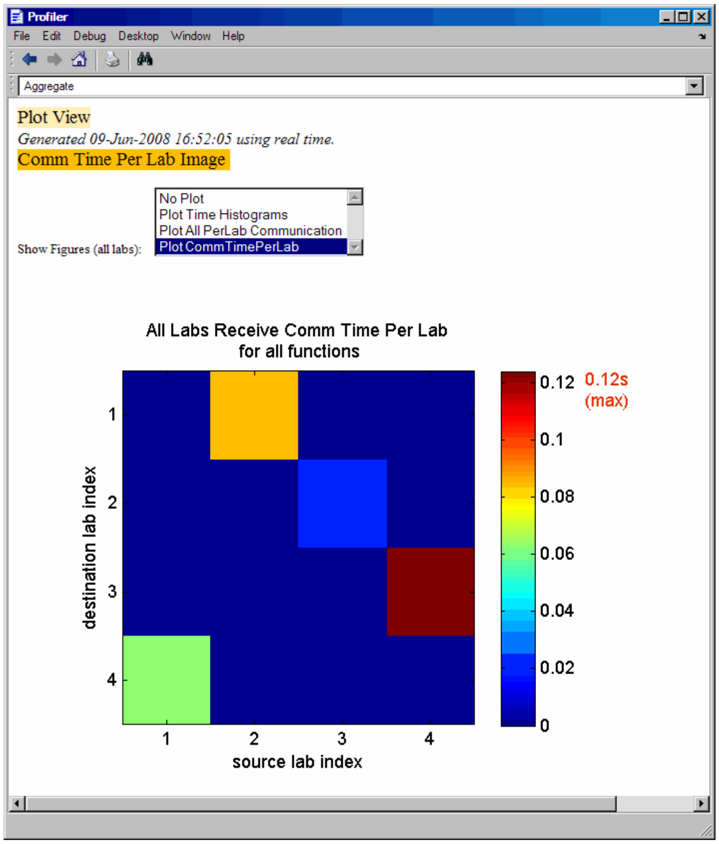
Plot View

Generated 09-Jun-2008 16:51:44 using real time.

PerLab Communication Images

Show Figures (all labs): Plot All PerLab Communication

To see only a plot of interworker communication times, select **Plot CommTimePerLab** in the **Show Figures** menu.



Plot View

Generated 09-Jun-2008 16:52:05 using real time.

Comm Time Per Lab Image

Show Figures (all labs): Plot CommTimePerLab

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.

Parallel Profiler Demos

To see demos that show further usage of the parallel profiler for work load distribution and balancing, use the help browser to access the Parallel Profiler Demos in Parallel Computing

Toolbox > Demos. Demos are also available on the Web at `http://www.mathworks.com/products/parallel-computing/demos.html`.

Benchmarking Performance

In this section...

Demos

Several benchmarking demos can help you understand and evaluate performance of the parallel computing products. You can access these demos in the Help Browser under the `Parallel Computing Toolbox` node: expand the node for `Demos` then `Benchmarks`.

HPC Challenge Benchmarks

Several MATLAB files are available to demonstrate HPC Challenge benchmark performance. You can find the files in the folder *`matlabroot`*`/toolbox/distcomp/examples/benchmark/hpcchallenge`. Each file is self-documented with explanatory comments. These files are not self-contained demos, but rather require that you know enough about your cluster to be able to provide the necessary information when using these files.

Troubleshooting and Debugging

In this section...

Object Data Size Limitations

The size limit of data transfers among the parallel computing objects is limited by the Java Virtual Machine (JVM) memory allocation. This limit applies to single transfers of data between client and workers in any job using an MJS cluster, or in any `parfor`-loop. The approximate size limitation depends on your system architecture:

System Architecture	Maximum Data Size Per Transfer (approx.)
64-bit	2.0 GB
32-bit	600 MB

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 mdce service under a different user with rights to log on as a service. See the section "Set the User" in the MATLAB Distributed Computing 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.
- Windows HPC Server (including CCS), LSF®, PBS Pro, TORQUE, and mpiexec 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 `MDCE_DECODE_FUNCTION` was not defined before the MATLAB worker started.
  - The decode function was not on the worker's path.
- If using mpiexec:
  - The passphrase to smpd was incorrect or missing.
  - The smpd daemon was not running on all the specified machines.

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 MJS

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 “Admin Center” in the MATLAB Distributed Computing Server documentation.

Detailed instructions for other methods of diagnosing connection problems between the client and MJS 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 MJS

If you cannot locate your MJS with `parcluster`, the most likely reasons for this failure are:

- The MJS is currently not running.
- Firewalls do not allow traffic from the client to the MJS.
- The client and the MJS are not running the same version of the software.
- The client and the MJS cannot resolve each other’s short hostnames.

MJS Cannot See the Client

If a warning message says that the MJS cannot open a TCP connection to the client computer, the most likely reasons for this are

- Firewalls do not allow traffic from the MJS to the client.
- The MJS cannot resolve the short hostname of the client computer. Use `pctconfig` to change the hostname that the MJS 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...
```

Evaluate Functions in a Cluster

In many cases, the tasks of a job are all the same, or there are a limited number of different kinds of tasks in a job. Parallel Computing Toolbox software offers a solution for these cases that alleviates you from having to define individual tasks and jobs when evaluating a function in a cluster of workers. The two ways of evaluating a function on a cluster are described in the following sections:

Evaluate Functions Synchronously

In this section...

Scope of dfeval

When you evaluate a function in a cluster of computers with `dfeval`, you provide basic required information, such as the function to be evaluated, the number of tasks to divide the job into, and the variable into which the results are returned. *Synchronous* (sync) evaluation in a cluster means that your MATLAB session is blocked until the evaluation is complete and the results are assigned to the designated variable. So you provide the necessary information, while Parallel Computing Toolbox software handles all the job-related aspects of the function evaluation.

When executing the `dfeval` function, the toolbox performs all these steps of running a job:

1. Finds a job manager or scheduler
2. Creates a job
3. Creates tasks in that job
4. Submits the job to the queue in the job manager or scheduler
5. Retrieves the results from the job
6. Destroys the job

By allowing the system to perform all the steps for creating and running jobs with a single function call, you do not have access to the full flexibility offered by Parallel Computing Toolbox software. However, this narrow functionality meets the requirements of many straightforward applications. To focus the scope of `dfeval`, the following limitations apply:

- You can pass property values to the job object; but you cannot set any task-specific properties, including callback functions, unless you use configurations.
- All the tasks in the job must have the same number of input arguments.
- All the tasks in the job must have the same number of output arguments.
- If you are using a third-party scheduler instead of the job manager, you must use configurations in your call to `dfeval`. See "Cluster Profiles" on page 6-12, and the reference page for `dfeval`.
- You do not have direct access to the job manager, job, or task objects, i.e., there are no objects in your MATLAB workspace to manipulate (though you can get them using `findResource` and the properties of the scheduler object). Note that `dfevalasync` returns a job object.
- Without access to the objects and their properties, you do not have control over the handling of errors.

Arguments of dfeval

Suppose the function `myfun` accepts three input arguments, and generates two output arguments. To run a job with four tasks that call `myfun`, you could type

```
[X, Y] = dfeval(@myfun, {a1 a2 a3 a4}, {b1 b2 b3 b4}, {c1 c2 c3 c4});
```

The number of elements of the input argument cell arrays determines the number of tasks in the job. All input cell arrays must have the same number of elements. In this example, there are four tasks.

Because `myfun` returns two arguments, the results of your job will be assigned to two cell arrays, `X` and `Y`. These cell arrays will have four elements each, for the four tasks. The first element of `X` will have the first output argument from the first task, the first element of `Y` will have the second argument from the first task, and so on.

The following table shows how the job is divided into tasks and where the results are returned.

Task Function Call	Results
`myfun(a1, b1, c1)`	X{1}, Y{1}
`myfun(a2, b2, c2)`	X{2}, Y{2}
`myfun(a3, b3, c3)`	X{3}, Y{3}
`myfun(a4, b4, c4)`	X{4}, Y{4}

So using one `dfeval` line would be equivalent to the following code, except that `dfeval` can run all the statements simultaneously on separate machines.

```
[X{1}, Y{1}] = myfun(a1, b1, c1);
[X{2}, Y{2}] = myfun(a2, b2, c2);
[X{3}, Y{3}] = myfun(a3, b3, c3);
[X{4}, Y{4}] = myfun(a4, b4, c4);
```

For further details and examples of the `dfeval` function, see the `dfeval` reference page.

Example — Use dfeval

Suppose you have a function called `averages`, which returns both the mean and median of three input values. The function might look like this.

```
function [mean_, median_] = averages (in1, in2, in3)
% AVERAGES Return mean and median of three input values
mean_ = mean([in1, in2, in3]);
median_ = median([in1, in2, in3]);
```

You can use `dfeval` to run this function on four sets of data using four tasks in a single job. The input data can be represented by the four vectors,

```
[1 2 6]
[10 20 60]
[100 200 600]
[1000 2000 6000]
```

A quick look at the first set of data tells you that its mean is 3, while its median is 2. So,

```
[x,y] = averages(1,2,6)
x =
     3
y =
     2
```

When calling `dfeval`, its input requires that the data be grouped together such that the first input argument to each task function is in the first cell array argument to `dfeval`, all second input arguments to the task functions are grouped in the next cell array, and so on. Because we want to evaluate four sets of data with four tasks, each of the three cell arrays will have four elements. In this example, the first arguments for the task functions are 1, 10, 100, and 1000. The second inputs to the task functions are 2, 20, 200, and 2000. With the task inputs arranged thus, the call to `dfeval` looks like this.

```
[A, B] = dfeval(@averages, {1 10 100 1000}, ...
    {2 20 200 2000}, {6 60 600 6000}, 'jobmanager', ...
    'MyJobManager', 'FileDependencies', {'averages.m'})

A =
    [   3]
    [  30]
    [ 300]
    [3000]

B =
    [   2]
    [  20]
    [ 200]
    [2000]
```

Notice that the first task evaluates the first element of the three cell arrays. The results of the first task are returned as the first elements of each of the two output values. In this case, the first task returns a mean of 3 and median of 2. The second task returns a mean of 30 and median of 20.

If the original function were written to accept one input vector, instead of three input values, it might make the programming of `dfeval` simpler. For example, suppose your task function were

```
function [mean_, median_] = avgs (V)
% AVGS Return mean and median of input vector
mean_ = mean(V);
median_ = median(V);
```

Now the function requires only one argument, so a call to `dfeval` requires only one cell array. Furthermore, each element of that cell array can be a vector containing all the values required for an individual task. The first vector is sent as a single argument to the first task, the second vector to the second task, and so on.

```
[A,B] = dfeval(@avgs, {[1 2 6] [10 20 60] ...
    [100 200 600] [1000 2000 6000]}, 'jobmanager', ...
    'MyJobManager', 'FileDependencies', {'avgs.m'})

A =
    [   3]
    [  30]
    [ 300]
    [3000]

B =
    [   2]
    [  20]
    [ 200]
    [2000]
```

If you cannot vectorize your function, you might have to manipulate your data arrangement for using `dfeval`. Returning to our original data in this example, suppose you want to start with data in three vectors.

```
v1 = [1 2 6];
v2 = [10 20 60];
v3 = [100 200 600];
v4 = [1000 2000 6000];
```

First put all your data in a single matrix.

```
dataset = [v1; v2; v3; v4]
dataset =

       1           2           6
      10          20          60
     100         200         600
    1000        2000        6000
```

Then make cell arrays containing the elements in each column.

```
c1 = num2cell(dataset(:,1));
c2 = num2cell(dataset(:,2));
c3 = num2cell(dataset(:,3));
```

Now you can use these cell arrays as your input arguments for `dfeval`.

```
[A, B] = dfeval(@averages, c1, c2, c3, 'jobmanager', ...
    'MyJobManager', 'FileDependencies', {'averages.m'})

A =
    [   3]
    [  30]
    [ 300]
    [3000]

B =
    [   2]
    [  20]
    [ 200]
    [2000]
```

Evaluate Functions Asynchronously

The dfeval function operates synchronously, that is, it blocks the MATLAB command line until its execution is complete. If you want to send a job to the job manager and get access to the command line while the job is being run *asynchronously* (async), you can use the dfevalasync function.

The dfevalasync function operates in the same way as dfeval, except that it does not block the MATLAB command line, and it does not directly return results.

To asynchronously run the example of the previous section, type

```
job1 = dfevalasync(@averages, 2, c1, c2, c3, 'jobmanager', ...
    'MyJobManager', 'FileDependencies', {'averages.m'});
```

Note that you have to specify the number of output arguments that each task will return (2, in this example).

The MATLAB session does not wait for the job to execute, but returns the prompt immediately. Instead of assigning results to cell array variables, the function creates a job object in the MATLAB workspace that you can use to access job status and results.

You can use the MATLAB session to perform other operations while the job is being run on the cluster. When you want to get the job's results, you should make sure it is finished before retrieving the data.

```
waitForState(job1, 'finished')
results = getAllOutputArguments(job1)

results =
    [   3]    [   2]
    [  30]    [  20]
    [ 300]    [ 200]
    [3000]    [2000]
```

The structure of the output arguments is now slightly different than it was for dfeval. The getAllOutputArguments function returns all output arguments from all tasks in a single cell array, with one row per task. In this example, each row of the cell array `results` will have two elements. So, `results{1,1}` contains the first output argument from the first task, `results{1,2}` contains the second argument from the first task, and so on.

For further details and examples of the `dfevalasync` function, see the `dfevalasync` reference page.

Program Independent Jobs

An Independent job is one whose tasks do not directly communicate with each other, that is, the tasks are independent of each other. The tasks do not need to run simultaneously, and a worker might 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.

The following sections describe how to program independent jobs:

Use a Local Cluster

In this section...

Create and Run Jobs with a Local Cluster

For jobs that require more control than the functionality offered by such high level constructs as `spmd` and `parfor`, 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 might not offer any performance enhancement, so 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 as well as 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 8-3
- "Create a Job" on page 8-3
- "Create Tasks" on page 8-5
- "Submit a Job to the Cluster" on page 8-5
- "Fetch the Job's Results" on page 8-6

Note that the objects that the client session uses to interact with the cluster are only references to data that is actually contained in the cluster's 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's job storage location, creates the job object `job1` in the client session, and if you omit the semicolon at the end of the command, displays some information about the job.

```
job1 = createJob(c)

 Job ID 2 Information
 ====================

                Type: Independent
            Username: eng864
               State: pending
          SubmitTime:
           StartTime:
    Running Duration: 0 days 0h 0m 0s

 - Data Dependencies

       AttachedFiles: {}
     AdditionalPaths: c:\temp

 - Associated Task(s)

      Number Pending: 0
      Number Running: 0
     Number Finished: 0
    Task ID of Errors: []
```

You can use the `get` function to see all the properties of this job object.

```
get(job1)
                Tag: ''
               Name: 'Job1'
            JobData: []
      AttachedFiles: {}
    AdditionalPaths: {'c:\temp'}
         CreateTime: 'Mon Nov 28 13:59:33 EST 2011'
         SubmitTime: ''
          StartTime: ''
         FinishTime: ''
           Username: 'eng864'
           UserData: []
              Tasks: [0x1 parallel.task.CJSTask]
              State: 'pending'
               Type: 'Independent'
             Parent: [1x1 parallel.cluster.Local]
                 ID: 1
```

Note that the job's `State` property is `pending`. This means the job has not yet been submitted (queued) for running, so you can now add tasks to it.

The scheduler's display now indicates the existence of your job, which is the pending one.

```
c

 Local Cluster Information
 =========================

                        Profile: local
                       Modified: false
                           Host: node345
                     NumWorkers: 2
             JobStorageLocation: C:\MATLAB\local_cluster_jobs\R2012a
              ClusterMatlabRoot: C:\apps\matlab
                OperatingSystem: windows
```

```
- Assigned 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 will 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.

```
get(job1,'Tasks')

 CJSTask: 5-by-1
 ================

    #    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 as many as twelve workers on your machine, and distributes the tasks of `job1` to these workers for evaluation.

Fetch the Job's Results

The results of each task's evaluation are stored in the task object's `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
get(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 MJS process for the local scheduler. When you submit a job for evaluation to the local cluster, the scheduler starts a MATLAB worker for each task in the job, but only up to 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` scheduler profile, up to a maximum of twelve. 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 MJS, nor with any other workers that might also be running on your client machine under the mdce service. Multiple MATLAB sessions on your computer can each start its own local scheduler with its own twelve workers, but these groups do not interact with each other, so you cannot combine local groups of workers to increase your local cluster size.

When you end your MATLAB client session, its local scheduler and any workers that happen to be running at that time also stop immediately.

Use a Cluster with a Supported Scheduler

In this section...

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 (MJS), Platform LSF (Load Sharing Facility), Microsoft Windows HPC Server (including CCS), PBS Pro, or a TORQUE scheduler.

This section assumes you have anMJS, LSF, PBS Pro, TORQUE, or Windows HPC Server (including CCS and HPC Server 2008) scheduler installed and running on your network. For more information about LSF, see `http://www.platform.com/Products/`. For more information about Windows HPC Server, see `http://www.microsoft.com/hpc`. With all of these cluster types, the basic job programming sequence is the same:

- "Define and Select a Profile" on page 8-9
- "Find a Cluster" on page 8-9
- "Create a Job" on page 8-11
- "Create Tasks" on page 8-12
- "Submit a Job to the Job Queue" on page 8-13
- "Retrieve the Job's Results" on page 8-14

Note that the objects that the client session uses to interact with the MJS are only references to data that is actually contained in the MJS, 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 MJS. You can find existing jobs using the `findJob` function or the `Jobs` property of the MJS 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');
```

```
c =

 MJS Cluster Information
 =======================

                        Profile: MyProfile
                       Modified: false
                           Host: node345
                     NumWorkers: 1
             JobStorageLocation: Database on node345
              ClusterMatlabRoot: C:\apps\matlab
                OperatingSystem: windows

 - Assigned Jobs

                 Number Pending: 0
                  Number Queued: 0
                 Number Running: 0
                Number Finished: 0

 - MJS Specific Properties

                           Name: my_mjs
               AllHostAddresses: 0:0:0:0
                 NumBusyWorkers: 0
                 NumIdleWorkers: 1
                       Username: mylogin
                  SecurityLevel: 0 (No security)
     IsUsingSecureCommunication: false
```

You can view all the accessible properties of the cluster object with the `get` function:

```
get(c)
```

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 job manager, `jm`, and creates a job object, `job1`, in the client session.

```
job1 = createJob(c)

Job ID 91 Information
 =====================

                 Type: Independent
             Username: mylogin
                State: pending
           SubmitTime:
            StartTime:
     Running Duration: 0 days 0h 0m 0s

 - Data Dependencies

        AttachedFiles: {}
      AdditionalPaths: {}

 - Associated Task(s)

       Number Pending: 0
       Number Running: 0
      Number Finished: 0
    Task ID of Errors: []
```

Use `get` to see all the accessible properties of this job object.

```
get(job1)
```

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
```

```
MJS Cluster Information
 =======================

 - Assigned 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" on page 8-15.

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.

```
get(job1,'Tasks')

MJSTask: 5-by-1
 ================

    #    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 an MJS, ff 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 MATLAB pool, run the `poolStartup` function. (This occurs when executing `matlabpool open` or when running other types of jobs that form and use a MATLAB pool.)
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 the Job's 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
```

Share Code

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:

- "Access Files Directly" on page 8-15
- "Pass Data Between Sessions" on page 8-16
- "Pass MATLAB Code for Startup and Finish" on page 8-17

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.
- By putting the `path` command in any of the appropriate startup files for the worker:
  - *`matlabroot`*`\toolbox\local\startup.m`
  - *`matlabroot`*`\toolbox\distcomp\user\jobStartup.m`
  - *`matlabroot`*`\toolbox\distcomp\user\taskStartup.m`

  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 MJS and worker services of MATLAB Distributed Computing Server software run by setting the `MDCEUSER` value in the `mdce_def` file before starting the services. For Microsoft Windows operating systems, there is also `MDCEPASS` for providing the account password for the specified user. For an explanation of service default settings and the `mdce_def` file, see “Define Script Defaults” in the MATLAB Distributed Computing Server System Administrator’s Guide.

Pass Data Between Sessions

A number of properties on task and job objects are designed for passing code or data from client to MJS 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 provided to the task constructor. 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.
- `AttachedFiles` — This property of the job object lists all the folders and files that get zipped and sent to the workers. On the worker, the data is unzipped, and the entries defined in the property are added to the search path of the MATLAB worker session.
- `AdditionalPaths` — This property of the job object provides paths that are added to the MATLAB workers’ search path, reducing the need for data transfers in a shared file system.

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 “Object Data Size Limitations” on page 6-44.

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/distcomp/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 MATLAB 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/distcomp/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/distcomp/user` folder of the worker's MATLAB installation.

Manage Objects in the MJS

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 8-18
- "Recover Objects" on page 8-18
- "Reset Callback Properties (MJS Only)" on page 8-19
- "Remove Objects Permanently" on page 8-19

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 Distributed Computing Server software or other cluster resources remain in place. When the client session ends, only the local reference objects are lost, not the actual objects in the cluster.

Therefore, if you have submitted your job to the 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 Distributed Computing 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 MJS cluster by the object `c`, you can create objects that reference all those objects contained in that MJS. All the jobs contained in the MJS are accessible in cluster object's `Jobs` property, which is an array of job objects:

```
all_jobs = get(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 MJS cluster `c`.

Reset Callback Properties (MJS 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 MJS is stopped and restarted. The ways to permanently remove jobs from the job manager are explained in the following sections:

- “Delete Selected Objects” on page 8-19
- “Start an MJS from a Clean State” on page 8-20

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 an MJS from a Clean State. When an MJS starts, by default it starts so that it resumes its former session with all jobs intact. Alternatively, an MJS 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 MJS 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" in the MATLAB Distributed Computing Server System Administrator's Guide.

Use the Generic Scheduler Interface

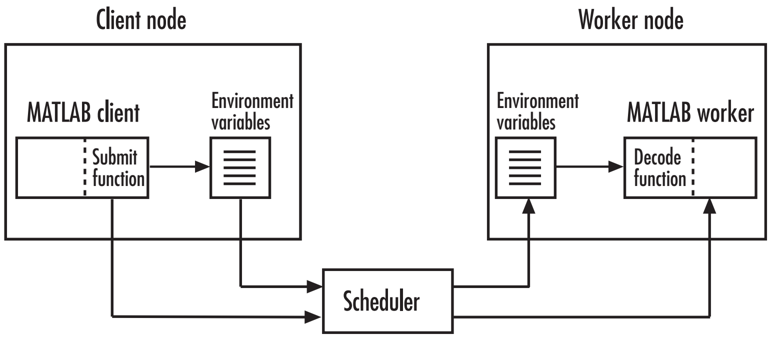
In this section...

Overview

Parallel Computing Toolbox software provides a generic interface that lets you interact with third-party schedulers, or use your own scripts for distributing tasks to other nodes on the cluster for evaluation.

Because each job in your application is comprised of several tasks, the purpose of your scheduler is to allocate a cluster node for the evaluation of each task, or to *distribute* each task to a cluster node. The scheduler starts remote MATLAB worker sessions on the cluster nodes to evaluate individual tasks of the job. To evaluate its task, a MATLAB worker session needs access to certain information, such as where to find the job and task data. The generic scheduler interface provides a means of getting tasks from your Parallel Computing Toolbox client session to your scheduler and thereby to your cluster nodes.

To evaluate a task, a worker requires five parameters that you must pass from the client to the worker. The parameters can be passed any way you want to transfer them, but because a particular one must be an environment variable, the examples in this section pass all parameters as environment variables.



Note Whereas the MJS keeps MATLAB workers running between tasks, a third-party scheduler runs MATLAB workers for only as long as it takes each worker to evaluate its one task.

MATLAB Client Submit Function

When you submit a job to a cluster, the function identified by the cluster object's `IndependentSubmitFcn` property executes in the MATLAB client session. You set the cluster's `IndependentSubmitFcn` property to identify the submit function and any arguments you might want to send to it. For example, to use a submit function called `mysubmitfunc`, you set the property with the command

```
set(c, 'IndependentSubmitFcn', @mysubmitfunc)
```

where `c` is the cluster object in the client session, created with the `parcluster` function. In this case, the submit function gets called with its three default arguments: cluster, job, and properties object, in that order. The function declaration line of the function might look like this:

```
function mysubmitfunc(cluster, job, props)
```

Inside the function of this example, the three argument objects are known as `cluster`, `job`, and `props`.

You can write a submit function that accepts more than the three default arguments, and then pass those extra arguments by including them in the definition of the `IndependentSubmitFcn` property.

```
time_limit = 300
testlocation = 'Plant30'
set(c, 'IndependentSubmitFcn', {@mysubmitfunc, time_limit, testlocation})
```

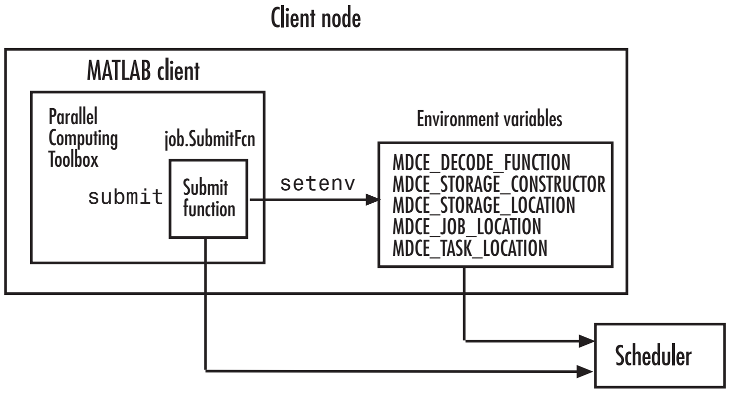
In this example, the submit function requires five arguments: the three defaults, along with the numeric value of `time_limit` and the string value of `testlocation`. The function's declaration line might look like this:

```
function mysubmitfunc(cluster, job, props, localtimeout, plant)
```

The following discussion focuses primarily on the minimum requirements of the submit and decode functions.

This submit function has three main purposes:

- To identify the decode function that MATLAB workers run when they start
- To make information about job and task data locations available to the workers via their decode function
- To instruct your scheduler how to start a MATLAB worker on the cluster for each task of your job



Identify the Decode Function

The client's submit function and the worker's decode function work together as a pair. Therefore, the submit function must identify its corresponding decode function. The submit function does this by setting the environment variable

`MDCE_DECODE_FUNCTION`. The value of this variable is a string identifying the name of the decode function *on the path of the MATLAB worker*. Neither the decode function itself nor its name can be passed to the worker in a job or task property; the file must already exist before the worker starts. For more information on the decode function, see "MATLAB Worker Decode Function" on page 8-27. Standard decode functions for independent and communicating jobs are provided with the product. If your submit functions make use of the definitions in these decode functions, you do not have to provide your own decode functions. For example, to use the standard decode function for independent jobs, in your submit function set `MDCE_DECODE_FUNCTION` to `'parallel.cluster.generic.independentDecodeFcn'`.

Pass Job and Task Data

The third input argument (after cluster and job) to the submit function is the object with the properties listed in the following table.

You do not set the values of any of these properties. They are automatically set by the toolbox so that you can program your submit function to forward them to the worker nodes.

Property Name	Description
`StorageConstructor`	String. Used internally to indicate that a file system is used to contain job and task data.
`StorageLocation`	String. Derived from the cluster `JobStorageLocation` property.
`JobLocation`	String. Indicates where this job's data is stored.
`TaskLocations`	Cell array. Indicates where each task's data is stored. Each element of this array is passed to a separate worker.
`NumberOfTasks`	Double. Indicates the number of tasks in the job. You do not need to pass this value to the worker, but you can use it within your submit function.

With these values passed into your submit function, the function can pass them to the worker nodes by any of several means. However, because the name of the decode function must be passed as an environment variable, the examples that follow pass all the other necessary property values also as environment variables.

The submit function writes the values of these object properties out to environment variables with the `setenv` function.

Define Scheduler Command to Run MATLAB Workers

The submit function must define the command necessary for your scheduler to start MATLAB workers. The actual command is specific to your scheduler and network configuration. The commands for some popular schedulers are listed in the following table. This table also indicates whether or not the scheduler automatically passes environment variables with its submission. If not, your command to the scheduler must accommodate these variables.

Scheduler	Scheduler Command	Passes Environment Variables
LSF	`bsub`	Yes, by default.
PBS	`qsub`	Command must specify which variables to pass.
Sun™ Grid Engine	`qsub`	Command must specify which variables to pass.

Your submit function might also use some of these properties and others when constructing and invoking your scheduler command. `cluster`, `job`, and `props` (so named only for this example) refer to the first three arguments to the submit function.

Argument Object	Property
`cluster`	`MatlabCommandToRun`
`cluster`	`ClusterMatlabRoot`

Argument Object	Property
`job`	`NumWorkersRange`
`props`	`NumberOfTasks`

Example — Write the Submit Function

The submit function in this example uses environment variables to pass the necessary information to the worker nodes. Each step below indicates the lines of code you add to your submit function.

1 Create the function declaration. There are three objects automatically passed into the submit function as its first three input arguments: the cluster object, the job object, and the props object.

```
function mysubmitfunc(cluster, job, props)
```

This example function uses only the three default arguments. You can have additional arguments passed into your submit function, as discussed in "MATLAB Client Submit Function" on page 8-22.

2 Identify the values you want to send to your environment variables. For convenience, you define local variables for use in this function.

```
decodeFcn = 'mydecodefunc';
jobLocation = get(props, 'JobLocation');
taskLocations = get(props, 'TaskLocations'); %This is a cell array
storageLocation = get(props, 'StorageLocation');
storageConstructor = get(props, 'StorageConstructor');
```

The name of the decode function that must be available on the MATLAB worker path is `mydecodefunc`.

3 Set the environment variables, other than the task locations. All the MATLAB workers use these values when evaluating tasks of the job.

```
setenv('MDCE_DECODE_FUNCTION', decodeFcn);
setenv('MDCE_JOB_LOCATION', jobLocation);
setenv('MDCE_STORAGE_LOCATION', storageLocation);
setenv('MDCE_STORAGE_CONSTRUCTOR', storageConstructor);
```

Your submit function can use any names you choose for the environment variables, with the exception of `MDCE_DECODE_FUNCTION`; the MATLAB worker looks for its decode function identified by this variable. If you use alternative names for the other environment variables, be sure that the corresponding decode function also uses your alternative variable names. You can see the variable names used in the standard decode function by typing

```
edit parallel.cluster.generic.independentDecodeFcn
```

4 Set the task-specific variables and scheduler commands. This is where you instruct your scheduler to start MATLAB workers for each task.

```
for i = 1:props.NumberOfTasks
    setenv('MDCE_TASK_LOCATION', taskLocations{i});
    constructSchedulerCommand;
end
```

The line `constructSchedulerCommand` represents the code you write to construct and execute your scheduler's submit command. This command is typically a string that combines the scheduler command with necessary flags, arguments, and values derived from the values of your object properties. This command is inside the `for`-loop so that your scheduler gets a command to start a MATLAB worker on the cluster for each task.

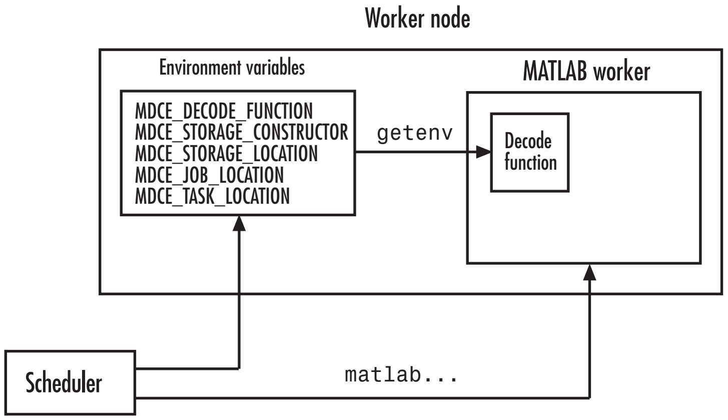
Note If you are not familiar with your network scheduler, ask your system administrator for help.

MATLAB Worker Decode Function

The sole purpose of the MATLAB worker's decode function is to read certain job and task information into the MATLAB worker session. This information could be stored in disk files on the network, or it could be available as environment variables on the worker node. Because the discussion of the submit function illustrated only the usage of environment variables, so does this discussion of the decode function.

When working with the decode function, you must be aware of the

- Name and location of the decode function itself
- Names of the environment variables this function must read



Note Standard decode functions are now included in the product. If your submit functions make use of the definitions in these decode functions, you do not have to provide your own decode functions. For example, to use the standard decode function for independent jobs, in your submit function set `MDCE_DECODE_FUNCTION` to `'parallel.cluster.generic.independentDecodeFcn'`. The remainder of this section is useful only if you use names and settings other than the standards used in the provided decode functions.

Identify File Name and Location

The client's submit function and the worker's decode function work together as a pair. For more information on the submit function, see "MATLAB Client Submit Function" on page 8-22. The decode function on the worker is identified by the submit function as the value of the environment variable `MDCE_DECODE_FUNCTION`. The environment variable must be copied from the client node to the worker node. Your scheduler might perform this task for you automatically; if it does not, you must arrange for this copying.

The value of the environment variable `MDCE_DECODE_FUNCTION` defines the filename of the decode function, but not its location. The file cannot be passed as part of the job `AdditionalPaths` or `AttachedFiles` property, because the function runs in the MATLAB worker before that session has access to the job. Therefore, the file location must be available to the MATLAB worker as that worker starts.

Note The decode function must be available on the MATLAB worker's path.

You can get the decode function on the worker's path by either moving the file into a folder on the path (for example, `*matlabroot*/toolbox/local`), or by having the scheduler use `cd` in its command so that it starts the MATLAB worker from within the folder that contains the decode function.

In practice, the decode function might be identical for all workers on the cluster. In this case, all workers can use the same decode function file if it is accessible on a shared drive.

When a MATLAB worker starts, it automatically runs the file identified by the `MDCE_DECODE_FUNCTION` environment variable. This decode function runs *before* the worker does any processing of its task.

Read the Job and Task Information

When the environment variables have been transferred from the client to the worker nodes (either by the scheduler or some other means), the decode function of the MATLAB worker can read them with the `getenv` function.

With those values from the environment variables, the decode function must set the appropriate property values of the object that is its argument. The property values that must be set are the same as those in the corresponding submit function, except that instead of the cell array `TaskLocations`, each worker has only the individual string `TaskLocation`, which is one element of the `TaskLocations` cell array. Therefore, the properties you must set within the decode function on its argument object are as follows:

- `StorageConstructor`
- `StorageLocation`
- `JobLocation`

- `TaskLocation`

Example — Write the Decode Function

The decode function must read four environment variables and use their values to set the properties of the object that is the function's output.

In this example, the decode function's argument is the object `props`.

```
function props = workerDecodeFunc(props)
% Read the environment variables:
storageConstructor = getenv('MDCE_STORAGE_CONSTRUCTOR');
storageLocation = getenv('MDCE_STORAGE_LOCATION');
jobLocation = getenv('MDCE_JOB_LOCATION');
taskLocation = getenv('MDCE_TASK_LOCATION');
%
% Set props object properties from the local variables:
set(props, 'StorageConstructor', storageConstructor);
set(props, 'StorageLocation', storageLocation);
set(props, 'JobLocation', jobLocation);
set(props, 'TaskLocation', taskLocation);
```

When the object is returned from the decode function to the MATLAB worker session, its values are used internally for managing job and task data.

Example — Program and Run a Job in the Client

1. Create a Scheduler Object

You use the `parcluster` function to create an object representing the cluster in your local MATLAB client session. Use a profile based on the generic type of cluster

```
c = parcluster('MyGenericProfile')
```

If your cluster uses a shared file system for workers to access job and task data, set the `JobStorageLocation` and `HasSharedFilesystem` properties to specify where the job data is stored and that the workers should access job data directly in a shared file system.

```
set(c, 'JobStorageLocation', '\\share\scratch\jobdata')
set(c, 'HasSharedFilesystem', true)
```

Note All nodes require access to the folder specified in the cluster object's `JobStorageLocation` property.

If `JobStorageLocation` is not set, the default location for job data is the current working directory of the MATLAB client the first time you use `parcluster` to create an object for this type of cluster, which might not be accessible to the worker nodes.

If MATLAB is not on the worker's system path, set the `ClusterMatlabRoot` property to specify where the workers are to find the MATLAB installation.

```
set(c, 'ClusterMatlabRoot', '\\apps\matlab\')
```

You can look at all the property settings on the scheduler object. If no jobs are in the `JobStorageLocation` folder, the `Jobs` property is a 0-by-1 array. All settable property values on a scheduler object are local to the MATLAB client, and are lost when you close the client session or when you remove the object from the client workspace with `delete` or `clear all`.

```
get(c)
```

You must set the `IndependentSubmitFcn` property to specify the submit function for this cluster.

```
set(c, 'IndependentSubmitFcn', @mysubmitfunc)
```

With the scheduler object and the user-defined submit and decode functions defined, programming and running a job is now similar to doing so with any other type of supported scheduler.

2. Create a Job

You create a job with the `createJob` function, which creates a job object in the client session. The job data is stored in the folder specified by the cluster object's `JobStorageLocation` property.

```
j = createJob(c)
```

This statement creates the job object `j` in the client session. Use `get` to see the properties of this job object.

```
get(j)
```

Note Properties of a particular job or task should be set from only one computer at a time.

This generic scheduler job has somewhat different properties than a job that uses an MJS. For example, this job has no callback functions.

The job's `State` property is `pending`. This state means the job has not been queued for running yet. This new job has no tasks, so its `Tasks` property is a 0-by-1 array.

The cluster's `Jobs` property is now a 1-by-1 array of job objects, indicating the existence of your job.

```
get(c)
```

3. Create Tasks

After you have created your job, you can create tasks for the job. Tasks define the functions to be evaluated by the workers during the running of the job. Often, the tasks of a job are identical except for different arguments or data. In this example, each task generates a 3-by-3 matrix of random numbers.

```
createTask(j, @rand, 1, {3,3});
createTask(j, @rand, 1, {3,3});
createTask(j, @rand, 1, {3,3});
createTask(j, @rand, 1, {3,3});
createTask(j, @rand, 1, {3,3});
```

The `Tasks` property of `j` is now a 5-by-1 matrix of task objects.

```
get(j,'Tasks')
```

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.

4. Submit a Job to the Job Queue

To run your job and have its tasks evaluated, you submit the job to the scheduler's job queue.

```
submit(j)
```

The scheduler distributes the tasks of `j` to MATLAB workers for evaluation.

The job runs asynchronously. If you need to wait for it to complete before you continue in your MATLAB client session, you can use the `wait` function.

```
wait(j)
```

This function pauses MATLAB until the `State` property of `j` is `'finished'` or `'failed'`.

5. Retrieve the Job's Results

The results of each task's evaluation are stored in that task object's `OutputArguments` property as a cell array. Use `fetchOutputs` to retrieve the results from all the tasks in the job.

```
results = fetchOutputs(j);
```

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
```

Supplied Submit and Decode Functions

There are several submit and decode functions provided with the toolbox for your use with the generic scheduler interface. These files are in the folder

`matlabroot``/toolbox/distcomp/examples/integration`

In this folder are subdirectories for each of several types of scheduler.

Depending on your network and cluster configuration, you might need to modify these files before they will work in your situation. Ask your system administrator for help.

At the time of publication, there are folders for PBS (`pbs`), and Platform LSF (`lsf`) schedulers, generic UNIX-based scripts (`ssh`), Sun Grid Engine (`sge`), and mpiexec on Microsoft Windows operating systems (`winmpiexec`). In addition, the `pbs`, `lsf`, and `sge` folders have subfolders called `shared`, `nonshared`, and `remoteSubmission`, which contain scripts for use in particular cluster configurations. Each of these subfolders contains a file called `README`, which provides instruction on where and how to use its scripts.

For each scheduler type, the folder (or configuration subfolder) contains wrappers, submit functions, and other job management scripts for independent and communicating jobs. For example, the folder *`matlabroot`*`/toolbox/distcomp/examples/integration/pbs/shared` contains the following files for use with a PBS scheduler:

Filename	Description
`independentSubmitFcn.m`	Submit function for a independent job
`communicatingSubmitFcn.m`	Submit function for a communicating job
`independentJobWrapper.sh`	Script that is submitted to PBS to start workers that evaluate the tasks of an independent job
`communicatingJobWrapper.sh`	Script that is submitted to PBS to start labs that evaluate the tasks of a communicating job
`deleteJobFcn.m`	Script to delete a job from the scheduler
`extractJobId.m`	Script to get the job's ID from the scheduler
`getJobStateFcn.m`	Script to get the job's state from the scheduler
`getSubmitString.m`	Script to get the submission string for the scheduler

These files are all programmed to use the standard decode functions provided with the product, so they do not have specialized decode functions.

The folder for other scheduler types contain similar files. As more files or solutions for more schedulers might become available at any time, visit the support page for this product on the MathWorks Web site at `http://www.mathworks.com/support/product/product.html?product=DM`. This Web page also provides contact information in case you have any questions.

Manage Jobs with Generic Scheduler

While you can use the `get`, `cancel`, and `delete` methods on jobs that use the generic scheduler interface, by default these methods access or affect only the job data where it is stored on disk. To cancel or delete a job or task that is currently running or queued, you must provide instructions to the scheduler directing it what to do and when to do it. To accomplish this, the toolbox provides a means of saving data associated with each job or task from the scheduler, and a set of properties to define instructions for the scheduler upon each cancel or destroy request.

Save Job Scheduler Data

The first requirement for job management is to identify the job from the scheduler's perspective. When you submit a job to the scheduler, the command to do the submission in your submit function can return from the scheduler some data about the job. This data typically includes a job ID. By storing that job ID with the job, you can later refer to the job by this ID when you send management commands to the scheduler. Similarly, you can store information, such as an ID, for each task. The toolbox function that stores this scheduler data is `setJobSchedulerData`.

If your scheduler accommodates submission of entire jobs (collection of tasks) in a single command, you might get back data for the whole job and/or for each task. Part of your submit function might be structured like this:

```
for ii = 1:props.NumberOfTasks
    define scheduler command per task
  end
  submit job to scheduler
  data_array = parse data returned from scheduler %possibly NumberOfTasks-by-2 matrix
  setJobSchedulerData(scheduler, job, data_array)
```

If your scheduler accepts only submissions of individual tasks, you might get return data pertaining to only each individual tasks. In this case, your submit function might have code structured like this:

```
for ii = 1:props.NumberOfTasks
    submit task to scheduler
    %Per-task settings:
    data_array(1,ii) = ... parse string returned from scheduler
    data_array(2,ii) = ... save ID returned from scheduler
    etc
  end
  setJobSchedulerData(scheduler, job, data_array)
```

Define Scheduler Commands in User Functions

With the scheduler data (such as the scheduler's ID for the job or task) now stored on disk along with the rest of the job data, you can write code to control what the scheduler should do when that particular job or task is canceled or destroyed.

For example, you might create these four functions:

- `myCancelJob.m`
- `myDeleteJob.m`
- `myCancelTask.m`
- `myDeleteTask.m`

Your `myCancelJob.m` function defines what you want to communicate to your scheduler in the event that you use the `cancel` function on your job from the MATLAB client. The toolbox takes care of the job state and any data management with the job data on disk, so your `myCancelJob.m` function needs to deal only with the part of the job currently running or queued with the scheduler. The toolbox function that retrieves scheduler data from the job is `getJobSchedulerData`. Your cancel function might be structured something like this:

```
function myCancelTask(sched, job)

    array_data = getJobSchedulerData(sched, job)
    job_id = array_data(...) % Extract the ID from the data, depending on how
                             % it was stored in the submit function above.
    command to scheduler canceling job job_id
```

In a similar way, you can define what do to for deleting a job, and what to do for canceling and deleting tasks.

Destroy or Cancel a Running Job

After your functions are written, you set the appropriate properties of the cluster object with handles to your functions. The corresponding cluster properties are:

- `CancelJobFcn`
- `DeleteJobFcn`
- `CancelTaskFcn`
- `DeleteTaskFcn`

You can set the properties in the Cluster Profile Manager for your cluster, or on the command line:

```
c = parcluster('MyGenericProfile');
% set required properties
set(c, 'CancelJobFcn', @myCancelJob)
set(c, 'DeleteJobFcn', @myDeleteJob)
set(c, 'CancelTaskFcn', @myCancelTask)
set(c, 'DeleteTaskFcn', @myDeleteTask)
```

Continue with job creation and submission as usual.

```
j1 = createJob(c);
for ii = 1:n
    t(ii) = createTask(j1,...)
end
submit(j1)
```

While the job is running or queued, you can cancel or delete the job or a task.

This command cancels the task and moves it to the finished state, and triggers execution of `myCancelTask`, which sends the appropriate commands to the scheduler:

```
cancel(t(4))
```

This command deletes job data for `j1`, and triggers execution of `myDeleteJob`, which sends the appropriate commands to the scheduler:

```
delete(j1)
```

Get State Information About a Job or Task

When using a third-party scheduler, it is possible that the scheduler itself can have more up-to-date information about your jobs than what is available to the toolbox from the job storage location. To retrieve that information from the scheduler, you can write a function to do that, and set the value of the `GetJobStateFcn` property as a handle to your function.

Whenever you use a toolbox function such as `get`, `wait`, etc., that accesses the state of a job on the generic scheduler, after retrieving the state from storage, the toolbox runs the function specified by the `GetJobStateFcn` property, and returns its result in place of the stored state. The function you write for this purpose must return a valid string value for the `State` of a job object.

When using the generic scheduler interface in a nonshared file system environment, the remote file system might be slow in propagating large data files back to your local data location. Therefore, a job's `State` property might indicate that the job is finished some time before all its data is available to you.

Summary

The following list summarizes the sequence of events that occur when running a job that uses the generic scheduler interface:

1. Provide a submit function and a decode function. Be sure the decode function is on all the MATLAB workers' paths.

The following steps occur in the MATLAB client session:

2. Define the `IndependentSubmitFcn` property of your scheduler object to point to the submit function.

3. Send your job to the scheduler.

   ```
   submit(job)
   ```

4. The client session runs the submit function.

5. The submit function sets environment variables with values derived from its arguments.

6. The submit function makes calls to the scheduler — generally, a call for each task (with environment variables identified explicitly, if necessary).

The following step occurs in your network:

7. For each task, the scheduler starts a MATLAB worker session on a cluster node.

The following steps occur in each MATLAB worker session:

8. The MATLAB worker automatically runs the decode function, finding it on the path.
9. The decode function reads the pertinent environment variables.
10. The decode function sets the properties of its argument object with values from the environment variables.
11. The MATLAB worker uses these object property values in processing its task without your further intervention.

Program Communicating Jobs

Communicating jobs are those in which the workers (or *labs*) can communicate with each other during the evaluation of their tasks. The following sections describe how to program communicating jobs:

Introduction

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. In this configuration, workers are sometimes also referred to as *labs*.

In principle, creating and running communicating jobs is similar to programming independent jobs:

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 8-13.
6. Retrieve the results.

The differences between independent jobs and communicating jobs are summarized in the following table.

Independent Job	**Communicating Job**
MATLAB sessions, called *workers*, perform the tasks but do not communicate with each other.	MATLAB sessions, sometimes called *labs*, 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	Tasks run simultaneously, so you can run the job only on as many

Independent Job	Communicating Job
the workers become available, so a worker can perform several of the tasks in a job.	workers as are available at run time. The start of the job might be delayed until the required number of workers is available.

A communicating job has only one task that runs simultaneously on every worker. 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.

Use a Cluster with a Supported Scheduler

In this section...

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 (MJS, local scheduler, Microsoft Windows HPC Server (including CCS), Platform LSF, PBS Pro, TORQUE, or mpiexec).

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 `pmode`, `matlabpool`, `spmd`, and `parfor`.) See "Use the Generic Scheduler Interface" on page 9-8.

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 labs. 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 “Cluster Profiles” on page 6-12. 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 9-4) 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:

```
set(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:

```
set(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]
```

Use the Generic Scheduler Interface

Introduction

This section discusses programming communicating jobs using the generic scheduler interface. This interface lets you execute jobs on your cluster with any scheduler you might have.

The principles of using the generic scheduler interface for communicating jobs are the same as those for distributed jobs. The overview of the concepts and details of submit and decode functions for distributed jobs are discussed fully in "Use the Generic Scheduler Interface" on page 8-21 in the chapter on Programming Distributed Jobs.

Code in the Client

Configure the Scheduler Object

Coding a communicating job for a generic scheduler involves the same procedure as coding an independent job.

1 Create an object representing your cluster with `parcluster`.

2 Set the appropriate properties on the cluster object if they are not defined in the profile. Because the scheduler itself is often common to many users and applications, it is probably best to use a profile for programming these properties. See "Cluster Profiles" on page 6-12.

   Among the properties required for a communicating job is `CommunicatingSubmitFcn`. You can write your own communicating submit and decode functions, or use those come with the product for various schedulers and platforms; see the following section, "Supplied Submit and Decode Functions" on page 9-9.

3. Use `createCommunicatingJob` to create a communicating job object for your cluster.

4. Create a task, run the job, and retrieve the results as usual.

Supplied Submit and Decode Functions

There are several submit and decode functions provided with the toolbox for your use with the generic scheduler interface. These files are in the folder

`matlabroot``/toolbox/distcomp/examples/integration`

In this folder are subfolders for each of several types of scheduler.

Depending on your network and cluster configuration, you might need to modify these files before they will work in your situation. Ask your system administrator for help.

At the time of publication, there are folders for PBS (`pbs`), and Platform LSF (`lsf`) schedulers, generic UNIX-based scripts (`ssh`), Sun Grid Engine (`sge`), and mpiexec on Microsoft Windows operating systems (`winmpiexec`). In addition, the `pbs`, `lsf`, and `sge` folders have subfolders called `shared`, `nonshared`, and `remoteSubmission`, which contain scripts for use in particular cluster configurations. Each of these subfolders contains a file called `README`, which provides instruction on where and how to use its scripts.

For each scheduler type, the folder (or configuration subfolder) contains wrappers, submit functions, and other job management scripts for independent and communicating jobs. For example, the folder *`matlabroot`*`/toolbox/distcomp/examples/integration/pbs/shared` contains the following files for use with a PBS scheduler:

Filename	Description
`independentSubmitFcn.m`	Submit function for an independent job
`communicatingSubmitFcn.m`	Submit function for a connunicating job
`independentJobWrapper.sh`	Script that is submitted to PBS to start workers that evaluate the tasks of an independent job

Filename	Description
`communicatingJobWrapper.sh`	Script that is submitted to PBS to start labs that evaluate the tasks of a communicating job
`deleteJobFcn.m`	Script to delete a job from the scheduler
`extractJobId.m`	Script to get the job's ID from the scheduler
`getJobStateFcn.m`	Script to get the job's state from the scheduler
`getSubmitString.m`	Script to get the submission string for the scheduler

These files are all programmed to use the standard decode functions provided with the product, so they do not have specialized decode functions. For communicating jobs, the standard decode function provided with the product is `parallel.cluster.generic.communicatingDecodeFcn`. You can view the required variables in this file by typing

```
edit parallel.cluster.generic.communicatingDecodeFcn
```

The folder for other scheduler types contain similar files. As more files or solutions for more schedulers might become available at any time, visit the support page for this product on the MathWorks Web site at `http://www.mathworks.com/support/product/product.html?product=DM`. This Web page also provides contact information in case you have any questions.

Further Notes on Communicating Jobs

In this section...

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 (labs), 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.

GPU Computing

When to Use a GPU for Matrix Operations

In this section...

Capabilities

This chapter describes how 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 the techniques described in this chapter might offer improved performance.

Several options are available for using your GPU:

- Transferring data between the MATLAB workspace and the GPU
- Evaluating built-in functions on the GPU
- Running MATLAB code on the GPU
- Creating kernels from PTX files for execution on the GPU
- Choosing one of multiple GPU cards to use

The particular workflows for these capabilities are described in the following sections of this chapter.

Requirements

For GPU computing requirements regarding supported GPU cards, platforms, drivers, and device compute capability, see the Parallel Computing Toolbox requirements web page at:

```
http://www.mathworks.com/products/parallel-computing/requirements.html
```

Demos

Demos showing the usage of the GPU are available in the Demos node under Parallel Computing Toolbox in the help browser. You can also access the product demos by entering the following command at the MATLAB prompt:

```
demo toolbox parallel
```

Using GPUArray

In this section...

Transfer Data Between Workspace and GPU

Send Data to the GPU

A GPUArray in MATLAB represents data that is stored on the GPU. Use the `gpuArray` function to transfer an array from the MATLAB workspace to the GPU:

```
N = 6;
M = magic(N);
G = gpuArray(M);
```

`G` is now a MATLAB GPUArray object that represents the data of the magic square stored on the GPU. The data provided as input to `gpuArray` must be nonsparse, and either `'single'`, `'double'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, or `'logical'`. (For more information, see "Data Types" on page 10-27.)

Retrieve Data from the GPU

Use the `gather` function to retrieve data from the GPU to the MATLAB workspace. This takes data that is on the GPU represented by a GPUArray object, and makes it available in the MATLAB workspace as a regular MATLAB variable. You can use `isequal` to verify that you get the correct data back:

```
G = gpuArray(ones(100, 'uint32'));
D = gather(G);
OK = isequal(D, ones(100, 'uint32'))
```

Examples: Transferring Data

Transfer Data 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);
```

Transfer Data of a Specified Precision. Create a matrix of double-precision random data in MATLAB, and then transfer the matrix as single-precision from MATLAB to the GPU:

```
X = rand(1000);
G = gpuArray(single(X));
```

Construct an Array for Storing on the GPU. Construct a 100-by-100 matrix of `uint32` ones and transfer it to the GPU. You can accomplish this with a single line of code:

```
G = gpuArray(ones(100, 'uint32'));
```

Create GPU Data Directly

A number of static methods on the GPUArray class allow you to directly construct arrays on the GPU without having to transfer them from the MATLAB workspace. These constructors require only array size and data class information, so they can construct an array without any element data from the workspace. Use any of the following to directly create an array on the GPU:

`parallel.gpu.GPUArray.ones`	`parallel.gpu.GPUArray.colon`
`parallel.gpu.GPUArray.zeros`	`parallel.gpu.GPUArray.rand`
`parallel.gpu.GPUArray.inf`	`parallel.gpu.GPUArray.randi`
`parallel.gpu.GPUArray.nan`	`parallel.gpu.GPUArray.randn`
`parallel.gpu.GPUArray.true`	`parallel.gpu.GPUArray.linspace`
`parallel.gpu.GPUArray.false`	`parallel.gpu.GPUArray.logspace`
`parallel.gpu.GPUArray.eye`	

For a complete list of available static methods in any release, type

```
methods('parallel.gpu.GPUArray')
```

The static constructors appear at the bottom of the output from this command.

For help on any one of the constructors, type

```
help parallel.gpu.GPUArray/functionname
```

For example, to see the help on the `colon` constructor, type

```
help parallel.gpu.GPUArray/colon
```

Example: Construct an Identity Matrix on the GPU

To create a 1024-by-1024 identity matrix of type `int32` on the GPU, type

```
II = parallel.gpu.GPUArray.eye(1024,'int32');
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 = parallel.gpu.GPUArray.ones(100, 100, 50);
size(G)
   100   100    50
classUnderlying(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 = parallel.gpu.GPUArray.zeros(8192, 1);
size(Z)
        8192           1
```

For a column vector, the size of the second dimension is 1.

Control the Random Stream

The following functions control the random number stream on the GPU:

`parallel.gpu.rng`
`parallel.gpu.RandStream`

These functions perform in the same way as `rng` and `RandStream` in MATLAB, but with certain limitations. For more information on the use and limits of these functions, type

```
help parallel.gpu.rng
help parallel.gpu.RandStream
```

The GPU uses the combined multiplicative recursive generator to create uniform random values, and uses inversion for creating normal values. This is not the default stream in a client MATLAB session on the CPU, but is the equivalent of

```
RandStream('CombRecursive','NormalTransform','Inversion');
```

However, a MATLAB worker session does have the same default stream as its GPU, even if it is a worker in a local cluster on the same machine. That is, a MATLAB client and workers do not have the same default stream.

In most cases, it does not matter that the default random stream on the GPU is not the same as the default stream in MATLAB on the CPU. But if you need to reproduce the same stream on both GPU and CPU, you can set the CPU random stream accordingly, and use the same seed to set both streams:

```
seed=0; n=4;

cpu_stream = RandStream('CombRecursive','Seed',seed,'NormalTransform','Inversion');
RandStream.setGlobalStream(cpu_stream);

gpu_stream = parallel.gpu.RandStream('CombRecursive','Seed',seed);
parallel.gpu.RandStream.setGlobalStream(gpu_stream);
```

```
r = rand(n);
R = parallel.gpu.GPUArray.rand(n);
isequal(r,R)
    1
```

Examine GPUArray Characteristics

There are several functions available for examining the characteristics of a GPUArray object:

Function	**Description**
`classUnderlying`	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
`length`	Length of vector or largest array dimension
`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 = gpuArray(rand(100));
s = size(G)
    100   100
```

Built-In Functions That Support GPUArray

A subset of the MATLAB built-in functions supports the use of GPUArray. Whenever any of these functions is called with at least one GPUArray as an input argument, it executes on the GPU and returns a GPUArray as the result. You can mix input from GPUArray and MATLAB workspace data in the same function call. These functions include the discrete Fourier transform (`fft`), matrix multiplication (`mtimes`), and left matrix division (`mldivide`).

The following functions and their symbol operators are enhanced to accept GPUArray input arguments so that they execute on the GPU:

`abs`	`conv2`	`floor`	`log`	`rem`
`acos`	`cos`	`fprintf`	`log10`	`repmat`
`acosh`	`cosh`	`full`	`log1p`	`reshape`
`acot`	`cot`	`gamma`	`log2`	`round`
`acoth`	`coth`	`gammaln`	`logical`	`sec`
`acsc`	`csc`	`gather`	`lt`	`sech`
`acsch`	`csch`	`ge`	`lu`	`shiftdim`
`all`	`ctranspose`	`gt`	`mat2str`	`sign`
`any`	`cumprod`	`horzcat`	`max`	`sin`
`arrayfun`	`cumsum`	`hypot`	`meshgrid`	`single`
`asec`	`det`	`ifft`	`min`	`sinh`
`asech`	`diag`	`ifft2`	`minus`	`size`
`asin`	`diff`	`ifftn`	`mldivide`	`sort`
`asinh`	`disp`	`imag`	`mod`	`sprintf`
`atan`	`display`	`ind2sub`	`mrdivide`	`sqrt`
`atan2`	`dot`	`int16`	`mtimes`	`sub2ind`
`atanh`	`double`	`int2str`	`ndgrid`	`subsasgn`
`beta`	`eig`	`int32`	`ndims`	`subsindex`
`betaln`	`eps`	`int64`	`ne`	`subsref`
`bitand`	`eq`	`int8`	`norm`	`sum`
`bitcmp`	`erf`	`inv`	`not`	`svd`
`bitor`	`erfc`	`ipermute`	`num2str`	`tan`
`bitshift`	`erfcinv`	`isempty`	`numel`	`tanh`
`bitxor`	`erfcx`	`isequal`	`permute`	`times`
`bsxfun`	`erfinv`	`isequaln`	`plot (and related)`	`transpose`
`cast`	`exp`	`isfinite`	`plus`	`tril`
`cat`	`expm1`	`isinf`	`power`	`triu`
`ceil`	`filter`	`islogical`	`prod`	`uint16`
`chol`	`filter2`	`isnan`	`qr`	`uint32`
`circshift`	`find`	`isreal`	`rdivide`	`uint64`
`classUnderlying`	`fft`	`issorted`	`real`	`uint8`
`colon`	`fft2`	`ldivide`	`reallog`	`uminus`
`complex`	`fftn`	`le`	`realpow`	`uplus`
`conj`	`fix`	`length`	`realsqrt`	`vertcat`
`conv`				

See the release notes for information about updates for individual functions.

For the complete list of available functions that support GPUArrays in your current version, use the `methods` function on the GPUArray class:

```
 methods('parallel.gpu.GPUArray')
```

To get help on specific overloaded functions, and to learn about any restrictions concerning their support for GPUArray objects, type:

```
help parallel.gpu.GPUArray/functionname
```

For example, to see the help on the overload of `lu`, type

```
help parallel.gpu.GPUArray/lu
```

The following functions are not methods of the GPUArray class, but they do work with GPUArray data:

```
 angle                ifftshift            rank
 fliplr               kron                 squeeze
 flipud               mean                 rot90
 flipdim              perms                trace
 fftshift
```

In most cases, if any of the input arguments to these functions is a GPUArray, their output arrays are GPUArrays. If the output is always scalar, it is MATLAB data in the workspace. 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`, etc.

Example: Calling Functions on GPUArray Objects

This example uses the `fft` and `real` functions, along with the arithmetic operators `+` and `*`. All the calculations are performed on the GPU, then `gather` retrieves the data from the GPU back to the MATLAB workspace.

```
Ga = gpuArray(rand(1000, 'single'));
Gfft = fft(Ga);
Gb = (real(Gfft) + Ga) * 6;
G = gather(Gb);
```

The whos command is instructive for showing where each variable's data is stored.

```
whos
 Name        Size            Bytes  Class

 G       1000x1000     4000000  single
 Ga      1000x1000         108  parallel.gpu.GPUArray
 Gb      1000x1000         108  parallel.gpu.GPUArray
 Gfft    1000x1000         108  parallel.gpu.GPUArray
```

Notice that all the arrays are stored on the GPU (GPUArray), except for G, which is the result of the gather function.

Execute MATLAB Code on a GPU

In this section...

MATLAB Code vs. GPUArray Objects

You have two options for performing MATLAB calculations on the GPU:

- You can transfer or create data on the GPU, and use the resulting GPUArray as input to enhanced built-in functions that support them. For more information and a list of functions that support GPUArray as inputs, see "Built-In Functions That Support GPUArray" on page 10-8.
- You can run your own MATLAB function file on a GPU.

Your decision on which solution to adopt depends on whether the functions you require are enhanced to support GPUArray, and the performance impact of transferring data to/from the GPU.

Running Your MATLAB Functions on the GPU

To execute your MATLAB function on the GPU, call `arrayfun` with a function handle to the MATLAB function as the first input argument:

```
result = arrayfun(@myFunction, arg1, arg2);
```

Subsequent arguments provide inputs to the MATLAB function. These input arguments can be workspace data or GPUArray. If any of the input arguments is a GPUArray, the function executes on the GPU and returns a GPUArray. (If none of the inputs is GPUArray, then `arrayfun` executes in the CPU.)

See the `arrayfun` reference page for descriptions of the available options.

Example: Running Your MATLAB Code

In this example, a small function applies correction data to an array of measurement data. The function defined in the file `myCal.m` is:

```
function c = myCal(rawdata, gain, offst)
c = (rawdata .* gain) + offst;
```

The function performs only element-wise operations when applying a gain factor and offset to each element of the `rawdata` array.

Create some 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 might keep the correction data on the GPU so that you do not have to transfer it for each application:

```
gn   = gpuArray(rand(1000))/100 + 0.995;
offs = gpuArray(rand(1000))/50  - 0.01;
```

Run your calibration function on the GPU:

```
corrected = arrayfun(@myCal, meas, gn, offs);
```

This runs on the GPU because the input arguments `gn` and `offs` are already in GPU memory.

Retrieve the corrected results from the GPU to the MATLAB workspace:

```
results = gather(corrected);
```

Supported MATLAB Code

The function passed into `arrayfun` can contain the following built-in MATLAB functions and operators:

`abs`	`double`	`log2`	`sin`
`and`	`eps`	`log10`	`single`
`acos`	`eq`	`log1p`	`sinh`
`acosh`	`erf`	`logical`	`sqrt`
`acot`	`erfc`	`lt`	`tan`
`acoth`	`erfcinv`	`max`	`tanh`
`acsc`	`erfcx`	`min`	`times`
`acsch`	`erfinv`	`minus`	`true`
`asec`	`exp`	`mod`	`uint8`
`asech`	`expm1`	`NaN`	`uint16`
`asin`	`false`	`ne`	`uint32`
`asinh`	`fix`	`not`	`xor`
`atan`	`floor`	`or`	
`atan2`	`gamma`	`pi`	
`atanh`	`gammaln`	`plus`	`+`
`beta`	`ge`	`power`	`-`
`betaln`	`gt`	`rand`	`.*`
`bitand`	`hypot`	`randi`	`./`
`bitcmp`	`imag`	`randn`	`.\`
`bitor`	`Inf`	`rdivide`	`.^`
`bitshift`	`int8`	`real`	`==`
`bitxor`	`int16`	`reallog`	`~=`
`ceil`	`int32`	`realmax`	`<`
`complex`	`intmax`	`realmin`	`<=`
`conj`	`intmin`	`realpow`	`>`
`cos`	`isfinite`	`realsqrt`	`>=`
`cosh`	`isinf`	`rem`	`&`
`cot`	`isnan`	`round`	`|`
`coth`	`ldivide`	`sec`	`~`
`csc`	`le`	`sech`	`&&`
`csch`	`log`	`sign`	`||`

Scalar expansion versions of the following:

```
*
/
\
^
```

Branching instructions:

```
break
continue
else
elseif
for
if
return
while
```

Generating Random Numbers on the GPU

The function you pass to `arrayfun` for execution on the GPU can contain the random number generator functions `rand`, `randi`, and `randn`. However, the GPU does not support the complete functionality of these that MATLAB does.

`arrayfun` on the GPU supports the following forms of random matrix generation:

```
rand                          randi
rand()                        randi()
rand('single')                randi(IMAX, ...)
rand('double')                randi([IMIN IMAX], ...)
randn                         randi(..., 'single')
randn()                       randi(..., 'double')
randn('single')               randi(..., 'int32')
randn('double')               randi(..., 'uint32')
```

You do not specify the array size for random generation. Instead, the number of generated random values is determined by the sizes of the input variables to your function. In effect, there will be enough random number elements to satisfy the needs of any input or output variables.

For example, suppose your function `myfun.m` contains the following code that includes generating and using the random matrix `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, where the number of random elements for `R` is determined by the size of `X`, so you do not need to specify it. The following code passes the GPUArray matrix `G` to `myfun` on the GPU.

```
G = 2*parallel.gpu.GPUArray.ones(4,4)
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.

Limitations and Restrictions

The following limitations apply to the code within the function that arrayfun or bsxfun is evaluating on a GPU.

- Nested and anonymous functions do not have access to their parent function workspace.
- Overloading the supported functions is not allowed.
- The code cannot call scripts.
- Indexing (subsasgn, subsref) is not supported.
- The following language features are not supported: persistent or global variables; parfor, spmd, switch, and try/catch.
- All double calculations are IEEE-compliant, but because of hardware limitations, single calculations are not.
- The supported data type conversions are single, double, int8, uint8, int16, uint16, int32, uint32, and logical.
- Like arrayfun in MATLAB, matrix exponential power, multiplication, and division (^, *, /, \) perform element-wise calculations only.
- There is no ans variable to hold unassigned computation results. Make sure to explicitly assign to variables the results of all calculations that you need to access.
- When generating random matrices with rand, randi, or randn, you do not need to specify the matrix size, and each element of the matrix has its own random stream.

Identify and Select a GPU from Multiple GPUs

If you have only one GPU in your computer, that GPU is the default. If you have more than one GPU card in your computer, you can use the following functions to identify and select which card you want to use:

Function	Description
`gpuDeviceCount`	The number of GPU cards in your computer
`gpuDevice`	Select which card to use, or see which card is selected and view its properties

Example: Selecting a GPU

This example shows how to identify and select a GPU for your computations.

1 Determine how many GPU devices are in your computer:

```
gpuDeviceCount

    2
```

2 With two devices, the first is the default. You can examine its properties to determine if that is the one you want to use:

```
gpuDevice

parallel.gpu.CUDADevice handle
Package: parallel.gpu

Properties:
                    Name: 'Tesla C1060'
                   Index: 1
       ComputeCapability: '1.3'
          SupportsDouble: 1
           DriverVersion: 4
      MaxThreadsPerBlock: 512
        MaxShmemPerBlock: 16384
      MaxThreadBlockSize: [512 512 64]
             MaxGridSize: [65535 65535]
```

```
                SIMDWidth: 32
              TotalMemory: 4.2948e+09
               FreeMemory: 4.2563e+09
      MultiprocessorCount: 30
             ClockRateKHz: 1296000
              ComputeMode: 'Default'
     GPUOverlapsTransfers: 1
   KernelExecutionTimeout: 0
          CanMapHostMemory: 1
          DeviceSupported: 1
           DeviceSelected: 1
```

If this is the device you want to use, you can proceed.

3. To use another device, call `gpuDevice` with the index of the other card, and view its properties to verify that it is the one you want. For example, this step chooses and views the second device (indexing is 1-based):

   ```
   gpuDevice(2)
   ```

Note If you select a device that does not have sufficient compute capability, you get a warning and you will not be able to use that device.

Execute CUDA or PTX Code on the GPU

In this section...

Create Kernels from CU Files

This section explains how to make a kernel from CU and PTX (parallel thread execution) files.

Compile a PTX 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 the Kernel Object

With a `.cu` file and a `.ptx` file you can create a kernel 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` kernel objects.

Run the Kernel

Use the `feval` function to evaluate the kernel on the GPU. The following examples show how to execute a kernel using GPUArray objects and MATLAB workspace data.

Use Workspace Data

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');

o = 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 GPU Data

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'));

o1 = 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:

```
o2 = feval(k, o1, i2);

r1 = gather(o1);
r2 = gather(o2);
```

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 argument 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.

Kernel 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]
    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.

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. 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.

Provide 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:

```
k = parallel.gpu.CUDAKernel('myfun.ptx', 'float *, const float *, float');
```

In parsing C prototype, the supported C 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`
		`bool`
		`char`, `unsigned char`, `char2`, `uchar2`

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 data. They are passed (after being cast into the requested type) directly to the kernel as scalars.
- 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 card, 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.

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).

Complete Kernel Workflow

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 card.

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 inputs of `1`. By default, a kernel runs on one thread.

```
>> o = feval(k, 1, 1);
o =
    2
```

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 as the first example, you need to specify the entry point name this time to distinguish it.

   ```
   k = parallel.gpu.CUDAKernel('test.ptx', 'add2', 'test.cu');
   ```

4. When you run the kernel, you need to set the number of threads correctly for the vectors you want to add.

   ```
   >> o = feval(k, 1, 1);
   o =
       2
   >> N = 128;
   >> k.ThreadBlockSize = N;
   >> o = feval(k, ones(N, 1), ones(N, 1));
   ```

GPU Characteristics and Limitations

In this section...

Data Types

Code in a function passed to `arrayfun` for execution on the GPU can use only these GPU native data types: `single`, `double`, `int32`, `uint32`, and `logical`.

The overloaded functions for GPUArrays support these types where appropriate. GPUArrays also support the storing of data types in addition to these. This allows a GPUArray to be used with kernels written for these alternative data types, such as `int8`, `uint8`, etc.

Complex Numbers

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 which 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)` will generate an error; instead you should call `sqrt(complex(x))`.

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)`	`x >= 1`

Function	Input Range for Real Output
`acsc(x)`	`x >= 1`
`asec(x)`	`x >= 1`
`asech(x)`	`0 <= x <= 1`
`asin(x)`	`abs(x) <= 1`
`atanh`	`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`

Object Reference

Data

codistributed	Access data of arrays distributed among workers in MATLAB pool
codistributor1d	1-D distribution scheme for codistributed array
codistributor2dbc	2-D block-cyclic distribution scheme for codistributed array
Composite	Access nondistributed data on multiple labs from client
distributed	Access data of distributed arrays from client
GPUArray	Array of data stored on GPU

Graphics Processing Unit

CUDAKernel	Kernel executable on GPU
GPUArray	Array of data stored on GPU
GPUDevice	Graphics Processing Unit (GPU)

Jobs and Tasks in a Cluster

parallel.Cluster	Access cluster properties and behaviors
parallel.Job	Access job properties and behaviors
parallel.Task	Access task properties and behaviors
parallel.Worker	Access worker that ran task

Generic Scheduler Interface Tools

RemoteClusterAccess	Connect to schedulers when client utilities are not available locally

Objects — Alphabetical List

ccsscheduler

Purpose

Access Microsoft Windows HPC Server scheduler

Constructor

`findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description

A ccsscheduler object provides access to your network's Windows HPC Server (including CCS) scheduler, which controls the job queue, and distributes job tasks to workers or labs for execution.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getDebugLog`	Read output messages from job run in CJS cluster

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`ClusterVersion`	Version of HPC Server scheduler

`Configuration`	Specify configuration to apply to object or toolbox function
`DataLocation`	Specify folder where job data is stored
`HasSharedFilesystem`	Specify whether nodes share data location
`JobDescriptionFile`	Name of XML job description file for Microsoft Windows HPC Server scheduler
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`JobTemplate`	Name of job template for HPC Server 2008 scheduler
`SchedulerHostname`	Name of host running Microsoft Windows HPC Server scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object
`UseSOAJobSubmission`	Allow service-oriented architecture (SOA) submission on HPC Server 2008 cluster

See Also

`genericscheduler`, `jobmanager`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`, `torquescheduler`

codistributed

Purpose

Access data of arrays distributed among workers in MATLAB pool

Constructor

`codistributed`, `codistributed.build`

Description

Data of distributed arrays that exist on the labs are accessible from the other labs as codistributed array objects.

Codistributed arrays on labs that you create inside `spmd` statements can be accessed via distributed arrays on the client.

Methods

`codistributed.cell`	Create codistributed cell array
`codistributed.colon`	Distributed colon operation
`codistributed.eye`	Create codistributed identity matrix
`codistributed.false`	Create codistributed false array
`codistributed.Inf`	Create codistributed array of `Inf` values
`codistributed.NaN`	Create codistributed array of Not-a-Number values
`codistributed.ones`	Create codistributed array of ones
`codistributed.rand`	Create codistributed array of uniformly distributed pseudo-random numbers
`codistributed.randn`	Create codistributed array of normally distributed random values
`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
`codistributed.true`	Create codistributed true array
`codistributed.zeros`	Create codistributed array of zeros
`gather`	Transfer distributed array data 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
`isaUnderlying`	True if distributed array's underlying elements are of specified class
`iscodistributed`	True for codistributed array
`redistribute`	Redistribute codistributed array with another distribution scheme
`sparse`	Create sparse distributed or codistributed matrix

codistributor1d

Purpose

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.

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

`Dimension`	Distributed dimension of codistributor1d object
`Partition`	Partition scheme of codistributor1d object

Purpose

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.

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

`BlockSize`	Block size of codistributor2dbc object
`codistributor2dbc.defaultBlockSize`	Default block size for codistributor2dbc distribution scheme
`LabGrid`	Lab grid of codistributor2dbc object
`Orientation`	Orientation of codistributor2dbc object

Composite

Purpose

Access nondistributed data on multiple labs from client

Constructor

`Composite`

Description

Variables that exist on the labs running an `spmd` statement are accessible on the client as a Composite object. A Composite resembles a cell array with one element for each lab. So for Composite `C`:

```
C{1} represents value of C on lab1
C{2} represents value of C on lab2
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 labs
`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

CUDAKernel

Purpose

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 "Execute CUDA or PTX Code on the GPU" on page 10-19.

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 1 or 2. Neither element of this vector can exceed the corresponding element in the vector of the `MaxGridSize` property of the `GPUDevice` object.

Property Name	Description
`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.
`EntryPoint`	(read-only) A string 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 data for a vector input/output, or the size of an output argument.
`ArgumentTypes`	(read-only) Cell array of strings, the same length as `NumRHSArguments`. Each of the strings 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 data and GPUArray, and to see which of the kernel inputs are being treated as outputs.

See Also

`GPUArray`, `GPUDevice`

distributed

Purpose

Access data of distributed arrays from client

Constructor

`distributed`

Description

Data of distributed arrays that exist on the labs are accessible on the client as a distributed array. A distributed array resembles a normal array in the way you access and manipulate its elements, but none of its data exists on the client.

Codistributed arrays that you create inside `spmd` statements are accessible via distributed arrays on the client. You can also create a distributed array explicitly on the client with the `distributed` function.

Methods

`distributed.cell`	Create distributed cell array
`distributed.eye`	Create distributed identity matrix
`distributed.false`	Create distributed false array
`distributed.Inf`	Create distributed array of `Inf` values
`distributed.NaN`	Create distributed array of Not-a-Number values
`distributed.ones`	Create distributed array of ones
`distributed.rand`	Create distributed array of uniformly distributed pseudo-random numbers
`distributed.randn`	Create distributed array of normally distributed random values
`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
`distributed.true`	Create distributed true array
`distributed.zeros`	Create distributed array of zeros
`gather`	Transfer distributed array data or GPUArray to local workspace
`isaUnderlying`	True if distributed array's underlying elements are of specified class
`isdistributed`	True for distributed array
`sparse`	Create sparse distributed or codistributed matrix

genericscheduler

Purpose

Access generic scheduler

Constructor

`findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description

A genericscheduler object provides access to your network's scheduler, which distributes job tasks to workers or labs for execution. The generic scheduler interface requires use of the MATLAB code submit function on the client and the MATLAB code decode function on the worker node.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getJobSchedulerData`	Get specific user data for job on generic scheduler
`setJobSchedulerData`	Set specific user data for job on generic scheduler

Properties

`CancelJobFcn`	Specify function to run when canceling job on generic scheduler
`CancelTaskFcn`	Specify function to run when canceling task on generic scheduler

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`Configuration`	Specify configuration to apply to object or toolbox function
`DataLocation`	Specify folder where job data is stored
`DestroyJobFcn`	Specify function to run when destroying job on generic scheduler
`DestroyTaskFcn`	Specify function to run when destroying task on generic scheduler
`GetJobStateFcn`	Specify function to run when querying job state on generic scheduler
`HasSharedFilesystem`	Specify whether nodes share data location
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`MatlabCommandToRun`	MATLAB command that generic scheduler runs to start lab
`ParallelSubmitFcn`	Specify function to run when parallel job submitted to generic scheduler
`SubmitFcn`	Specify function to run when job submitted to generic scheduler

`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`ccsscheduler`, `jobmanager`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`, `torquescheduler`

Purpose

Array of data stored on GPU

Constructor

`gpuArray` converts an array in the MATLAB workspace into a GPUArray with data stored on the GPU device.

Also, the following static methods create GPUArray data:

`parallel.gpu.GPUArray.colon`	`parallel.gpu.GPUArray.ones`
`parallel.gpu.GPUArray.eye`	`parallel.gpu.GPUArray.rand`
`parallel.gpu.GPUArray.false`	`parallel.gpu.GPUArray.randi`
`parallel.gpu.GPUArray.inf`	`parallel.gpu.GPUArray.randn`
`parallel.gpu.GPUArray.linspace`	`parallel.gpu.GPUArray.true`
`parallel.gpu.GPUArray.logspace`	`parallel.gpu.GPUArray.zeros`
`parallel.gpu.GPUArray.nan`	

You can get help on any of these methods with the command

```
help parallel.gpu.GPUArray.methodname
```

where *`methodname`* is the name of the method. For example, to get help on `rand`, type

```
help parallel.gpu.GPUArray.rand
```

The following methods control the random number stream on the GPU:

`parallel.gpu.RandStream`
`parallel.gpu.rng`

Description

A `GPUArray` object represents an array of data stored on the GPU. You can use the data for direct calculations, or in CUDA kernels that execute on the GPU. You can return data to the MATLAB workspace with the `gather` function.

Methods

`arrayfun`	Apply function to each element of array on GPU
`bsxfun`	Binary singleton expansion function for GPUArray
`existsOnGPU`	Determine if GPUArray or CUDAKernel is available on GPU

Other overloaded methods for a GPUArray object are too numerous to list here. Most resemble and behave the same as built-in MATLAB functions. See "Using GPUArray" on page 10-4. For the complete list of those supported, use the `methods` function on the GPUArray class:

```
methods('parallel.gpu.GPUArray')
```

Among the GPUArray methods there are several for examining the characteristics of a GPUArray object. Most behave like the MATLAB functions of the same name:

Function	Description
`classUnderlying`	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
`length`	Length of vector or largest array dimension
`ndims`	Number of dimensions in the array
`size`	Size of array dimensions

See Also

`CUDAKernel`, `GPUDevice`

Purpose

Graphics Processing Unit (GPU)

Constructor

`gpuDevice`

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.

Methods

The following convenience functions let you identify and select a GPU device:

`gpuDevice`	Query or select GPU device
`gpuDeviceCount`	Number of GPU devices present
`reset`	Reset GPU device and clear its memory

Methods of the class include the following:

Method Name	Description
`parallel.gpu.GPUDevice.isAvailable(idx)`	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.
`wait(dev)`	Blocks MATLAB until GPU calculations are complete on specified device.

For the complete list, use the `methods` function on the `GPUDevice` class:

```
methods('parallel.gpu.GPUDevice')
```

You can get help on any of the class methods with the command

```
help parallel.gpu.GPUDevice.methodname
```

where `methodname` is the name of the method. For example, to get help on `isAvailable`, type

```
help parallel.gpu.GPUDevice.isAvailable
```

Properties

A `GPUDevice` object has the following read-only properties:

Property Name	Description
`Name`	Name of the CUDA device.
`Index`	Index by which you can select the device.
`ComputeCapability`	Computational capability of the CUDA device. Must meet required specification.
`SupportsDouble`	Indicates if this device can support double precision operations.
`DriverVersion`	The CUDA device driver version currently in use. Must meet required specification.
`MaxThreadsPerBlock`	Maximum supported number of threads per block during CUDAKernel execution.
`MaxShmemPerBlock`	Maximum supported amount of shared memory that can be used by a thread block during CUDAKernel execution.
`MaxThreadBlockSize`	Maximum size in each dimension for thread block. 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.
`SIMDWidth`	Number of simultaneously executing threads.
`TotalMemory`	Total available memory (in bytes) on the device.

Property Name	Description
FreeMemory	Free memory (in bytes) on the device. This property is available only for the currently selected device, and has the value NaN for unselected devices.
MultiprocessorCount	The number of vector processors present on the device. The total core count of the device is 8 times this property.
ClockRateKHz	Peak clock rate of the GPU in kHz.
ComputeMode	The compute mode of the device, according to the following values: 'Default' — The device is not restricted and can support multiple CUDA sessions simultaneously. That is, MATLAB can share the GPU with other applications. 'Exclusive thread' or 'Exclusive process' — The device can be used by only one CUDA session. MATLAB can use this device only when no other application is already using it. 'Prohibited' — The device cannot be used.
GPUOverlapsTransfers	Indicates if the device supports overlapped transfers.
KernelExecutionTimeout	Indicates if the device can abort long-running kernels. If true, the operating system places an upper bound on the time allowed for the CUDA kernel to execute, after which the CUDA driver times out the kernel and returns an error.
CanMapHostMemory	Indicates if the device supports mapping host memory into the CUDA address space.
DeviceSupported	Indicates if toolbox can use this this device. Not all devices are supported; for example, if their ComputeCapability is insufficient, the toolbox cannot use them.
DeviceSelected	Indicates if this is the currently selected device.

See Also `CUDAKernel`, `GPUArray`

Purpose

Define job behavior and properties when using job manager

Constructor

`createJob`

Container Hierarchy

Parent	`jobmanager` object
Children	`task` objects

Description

A job object contains all the tasks that define what each worker does as part of the complete job execution. A job object is used only with a job manager as scheduler.

Methods

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`diary`	Display or save Command Window text of batch job
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler
`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Properties

`AuthorizedUsers`	Specify users authorized to access job
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishedFcn`	Specify callback to execute after task or job runs
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`MaximumNumberOfWorkers`	Specify maximum number of workers to perform job tasks
`MinimumNumberOfWorkers`	Specify minimum number of workers to perform job tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path
`QueuedFcn`	Specify function file to execute when job is submitted to job manager queue
`RestartWorker`	Specify whether to restart MATLAB workers before evaluating job tasks

`RunningFcn`	Specify function file to execute when job or task starts running
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object
`Tasks`	Tasks contained in job object
`Timeout`	Specify time limit to complete task or job
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also

`paralleljob`, `simplejob`, `simpleparalleljob`

jobmanager

Purpose Control job queue and execution

Constructor `findResource`

Container Hierarchy

Parent	None
Children	`job`, `paralleljob`, and `worker` objects

Description A jobmanager object provides access to the job manager, which controls the job queue, distributes job tasks to workers or labs for execution, and maintains job results. The job manager is provided with the MATLAB Distributed Computing Server product, and its use as a scheduler is optional.

Methods

`changePassword`	Prompt user to change job manager password or MJS password
`clearLocalPassword`	Delete local store of user's job manager password
`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`demote`	Demote job in cluster queue
`findJob`	Find job objects stored in scheduler
`pause`	Pause job manager queue
`promote`	Promote job in MJS cluster queue
`resume`	Resume processing queue in job manager

Properties

`BusyWorkers`	Workers currently running tasks
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`Configuration`	Specify configuration to apply to object or toolbox function
`HostAddress`	IP address of host running job manager or worker session
`Hostname`	Name of host running job manager or worker session
`IdleWorkers`	Idle workers available to run tasks
`IsUsingSecureCommunication`	True if job manager and workers use secure communication
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`Name`	Name of job manager, job, or worker object
`NumberOfBusyWorkers`	Number of workers currently running tasks
`NumberOfIdleWorkers`	Number of idle workers available to run tasks
`PromptForPassword`	Specify if system should prompt for password when authenticating user
`SecurityLevel`	Security level controlling access to job manager and its jobs

`State`	Current state of task, job, job manager, or worker
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also

`ccsscheduler`, `genericscheduler`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`, `torquescheduler`

Purpose

Access local scheduler on client machine

Constructor

`findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description

A localscheduler object provides access to your client machine's local scheduler, which controls the job queue, and distributes job tasks to workers or labs for execution on the client machine.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getDebugLog`	Read output messages from job run in CJS cluster

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`Configuration`	Specify configuration to apply to object or toolbox function

`DataLocation`	Specify folder where job data is stored
`HasSharedFilesystem`	Specify whether nodes share data location
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`jobmanager`

Purpose Access Platform LSF scheduler

Constructor `findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description An lsfscheduler object provides access to your network's Platform LSF scheduler, which controls the job queue, and distributes job tasks to workers or labs for execution.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getDebugLog`	Read output messages from job run in CJS cluster
`setupForParallelExecution`	Set options for submitting parallel jobs to scheduler

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterName`	Name of Platform LSF cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers

`ClusterSize`	Number of workers available to scheduler
`Configuration`	Specify configuration to apply to object or toolbox function
`DataLocation`	Specify folder where job data is stored
`HasSharedFilesystem`	Specify whether nodes share data location
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`MasterName`	Name of Platform LSF master node
`ParallelSubmissionWrapperScript`	Script that scheduler runs to start labs
`SubmitArguments`	Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`ccsscheduler`, `genericscheduler`, `jobmanager`, `mpiexec`, `pbsproscheduler`, `torquescheduler`

Purpose

Define MATLAB pool job behavior and properties when using job manager

Constructor

`createMatlabPoolJob`

Container Hierarchy

Parent	`jobmanager` object
Children	`task` object

Description

A matlabpooljob object contains all the information needed to define what each lab does as part of the complete job execution. A MATLAB pool job uses one worker in a MATLAB pool to run a parallel job on the other labs of the pool. A matlabpooljob object is used only with a job manager as scheduler.

Methods

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`diary`	Display or save Command Window text of batch job
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler

`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Properties

`AuthorizedUsers`	Specify users authorized to access job
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishedFcn`	Specify callback to execute after task or job runs
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`MaximumNumberOfWorkers`	Specify maximum number of workers to perform job tasks
`MinimumNumberOfWorkers`	Specify minimum number of workers to perform job tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path

`QueuedFcn`	Specify function file to execute when job is submitted to job manager queue
`RestartWorker`	Specify whether to restart MATLAB workers before evaluating job tasks
`RunningFcn`	Specify function file to execute when job or task starts running
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object
`Task`	First task contained in MATLAB pool job object
`Tasks`	Tasks contained in job object
`Timeout`	Specify time limit to complete task or job
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also

`paralleljob`, `simplematlabpooljob`, `simpleparalleljob`

Purpose

Directly access mpiexec for job distribution

Constructor

`findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description

An mpiexec object provides direct access to the mpiexec executable for distribution of a job's tasks to workers or labs for execution.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getDebugLog`	Read output messages from job run in CJS cluster

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`Configuration`	Specify configuration to apply to object or toolbox function

`DataLocation`	Specify folder where job data is stored
`EnvironmentSetMethod`	Specify means of setting environment variables for mpiexec scheduler
`HasSharedFilesystem`	Specify whether nodes share data location
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`MpiexecFileName`	Specify pathname of executable mpiexec command
`SubmitArguments`	Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object
`WorkerMachineOsType`	Specify operating system of nodes on which mpiexec scheduler will start labs

See Also

`ccsscheduler`, `genericscheduler`, `jobmanager`, `lsfscheduler`, `pbsproscheduler`, `torquescheduler`

parallel.Cluster

Purpose Access cluster properties and behaviors

Constructors `parcluster`

Container Hierarchy

Parent	None
Children	parallel.Job

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 (MJS) and common job scheduler (CJS). The MJS is available in the MATLAB Distributed Computer Server. The CJS clusters encompass all other types, including the local, generic, and third-party schedulers.

The following table describes the available types of cluster objects.

Cluster Type	Description
parallel.cluster.MJS	Interact with MATLAB job scheduler (MJS) cluster on-premises
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 Platform LSF
parallel.cluster.PBSPro	Interact with CJS cluster running Altair PBS Pro
parallel.cluster.Torque	Interact with CJS cluster running TORQUE

Cluster Type	Description
parallel.cluster.Generic	Interact with CJS cluster using the generic interface
parallel.cluster.Mpiexec	Interact with CJS cluster using mpiexec from local host

Methods

Common to All Cluster Types

`batch`	Run MATLAB script or function on worker
`createCommunicatingJob`	Create communicating job on cluster
`createJob`	Create distributed or independent job
`findJob`	Find job objects stored in scheduler
`isequal`	True if clusters have same property values
`matlabpool`	Open or close pool of MATLAB sessions for parallel computation
`saveAsProfile`	Save cluster properties to specified profile
`saveProfile`	Save modified cluster properties to its current profile

MJS

`changePassword`	Prompt user to change job manager password or MJS password
`demote`	Demote job in cluster queue

`logout`	Log out of MJS cluster
`pause`	Pause job manager queue
`promote`	Promote job in MJS cluster queue
`resume`	Resume processing queue in job manager

HPC Server, PBS Pro, LSF, TORQUE, 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
`Modified`	True if any properties in this cluster have been modified
`NumWorkers`	Number of workers available for this cluster
`OperatingSystem`	Operating system of nodes used by cluster
`Profile`	Profile used to build this cluster
`Type`	Type of this cluster
`UserData`	Data associated with cluster object within client session

MJS

MJS 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

Property	Description
`HasSecureCommunication`	True if cluster is using secure communication
`Name`	Name of this cluster
`NumBusyWorkers`	Number of workers currently running tasks
`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
`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
`JobDescriptionFile`	Name of XML job description file to use when creating jobs

Property	Description
`JobTemplate`	Name of job template to use for jobs submitted to HPC Server
`HasSharedFilesystem`	Specify whether client and cluster nodes share JobStorageLocation
`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`	Define resources to request for communicating jobs
`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 Platform LSF cluster
`CommunicatingJobWrapper`	Script cluster runs to start workers
`HasSharedFilesystem`	Specify whether client and cluster nodes share JobStorageLocation
`SubmitArguments`	Specify additional arguments to use when submitting jobs

Generic

Generic cluster objects 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
`IndependentSubmitFcn`	Function to run when submitting independent job
`HasSharedFilesystem`	Specify whether client and cluster nodes share JobStorageLocation

Help

For general command-line help on using clusters, type:

```
help cluster
```

For help on a specific method or property of any type of cluster object, type `help parallel.cluster.<cluster-type>.<mthd-or-prop>`. For example:

```
help parallel.cluster.MJS.BusyWorkers
help parallel.cluster.HPCServer.JobTemplate
help parallel.cluster.generic.IndependentSubmitFcn
```

See Also

`parallel.Job`, `parallel.Task`, `parallel.Worker`

parallel.Job

Purpose

Access job properties and behaviors

Constructors

`createCommunicatingJob`, `createJob`, `findJob`

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 MJS cluster
parallel.job.MJSCommunicatingJob	Job of communicating tasks on MJS cluster
parallel.job.CJSIndependentJob	Job of independent tasks on CJS cluster
parallel.job.CJSCommunicatingJob	Job of communicating tasks on CJS cluster

Methods

All job type objects have the same methods, described in the following table.

`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
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler
`wait`	Wait for job to change state or for GPU calculation to complete

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
`CreateTime`	Time at which job was created
`FinishTime`	Time at which job finished running
`ID`	Job's numeric identifier
`JobData`	Data made available to all workers for job's tasks
`Name`	Name of job

Property	Description
`Parent`	Cluster object containing this job
`StartTime`	Time at which job started running
`State`	State of job: `'pending'`, `'queued'`, `'running'`, `'finished'`, or `'failed'`
`SubmitTime`	Time at which job was submitted to queue
`Tag`	Label associated with job
`Tasks`	Array of task objects contained in job
`Type`	Job type: `'independent'`, `'pool'`, or `'spmd'`
`UserData`	Data associated with job object
`Username`	Name of user who owns job

MJS Jobs

MJS independent job objects and MJS 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 queued

Property	Description
`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 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 help on any of the parallel.Job methods or properties, type `help parallel.job.<job-type>.<mthd-or-prop>`. For example:

```
help parallel.job.MJSIndependentJob.NumWorkersRange
help parallel.job.CJSCommunicatingJob.fetchOutputs
```

See Also

`parallel.Cluster`, `parallel.Task`, `parallel.Worker`

parallel.Task

Purpose

Access task properties and behaviors

Constructors

`createTask`, `findTask`

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 MJS 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
`wait`	Wait for job to change state or for GPU calculation to complete

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
`CreateTime`	When task was created
`Diary`	Text produced by execution of task object's function
`Error`	Task error information
`ErrorIdentifier`	Task error identifier
`ErrorMessage`	Message from task error
`FinishTime`	When task finished running
`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
`StartTime`	When task started running
`State`	Current state of task
`UserData`	Data associated with this task object
`Worker`	Object representing worker that ran this task

MJS Tasks

MJS 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 to complete task

CJS Tasks

CJS task objects have no properties beyond the properties common to all clusters.

Help

To get help on any of the parallel.Task methods or properties, type `help parallel.task.<task-type>.<mthd-or-prop>`. For example:

```
help parallel.task.CJSTask.cancel
help parallel.task.MJSTask.RunningFcn
```

See Also

`parallel.Cluster`, `parallel.Job`, `parallel.Worker`

Purpose

Access worker that ran task

Constructors

You do not construct worker objects directly in the workspace. A 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 MJS cluster
parallel.cluster.CJSWorker	MATLAB worker on CJS cluster

Methods

There are no methods for a parallel.Worker object other than generic methods for any objects in the work space, such as `delete`, etc.

Properties

MJS Worker

The following table describes the properties of an MJS worker.

Property	Description
`AllHostAddresses`	IP addresses of worker host
`Name`	Name of worker, set when worker session started
`Parent`	MJS 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 worker ran; the value of the MATLAB function `computer` executed on the worker
`Host`	Host name where worker executed task
`ProcessId`	Process identifier for worker

See Also

`parallel.Cluster`, `parallel.Job`, `parallel.Task`

Purpose

Define parallel job behavior and properties when using job manager

Constructor

`createParallelJob`

Container Hierarchy

Parent	`jobmanager` object
Children	`task` objects

Description

A paralleljob object contains all the tasks that define what each lab does as part of the complete job execution. A parallel job runs simultaneously on all labs and uses communication among the labs during task evaluation. A paralleljob object is used only with a job manager as scheduler.

Methods

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`diary`	Display or save Command Window text of batch job
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler

`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Properties

`AuthorizedUsers`	Specify users authorized to access job
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishedFcn`	Specify callback to execute after task or job runs
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`MaximumNumberOfWorkers`	Specify maximum number of workers to perform job tasks
`MinimumNumberOfWorkers`	Specify minimum number of workers to perform job tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path

`QueuedFcn`	Specify function file to execute when job is submitted to job manager queue
`RestartWorker`	Specify whether to restart MATLAB workers before evaluating job tasks
`RunningFcn`	Specify function file to execute when job or task starts running
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object
`Tasks`	Tasks contained in job object
`Timeout`	Specify time limit to complete task or job
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also

`job`, `simplejob`, `simpleparalleljob`

pbsproscheduler

Purpose

Access PBS Pro scheduler

Constructor

`findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description

A pbsproscheduler object provides access to your network's PBS Pro scheduler, which controls the job queue, and distributes job tasks to workers or labs for execution.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getDebugLog`	Read output messages from job run in CJS cluster
`setupForParallelExecution`	Set options for submitting parallel jobs to scheduler

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler

`Configuration`	Specify configuration to apply to object or toolbox function
`DataLocation`	Specify folder where job data is stored
`HasSharedFilesystem`	Specify whether nodes share data location
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`ParallelSubmissionWrapperScript`	Script that scheduler runs to start labs
`RcpCommand`	Command to copy files from client
`ResourceTemplate`	Resource definition for PBS Pro or TORQUE scheduler
`RshCommand`	Remote execution command used on worker nodes during parallel job
`ServerName`	Name of current PBS Pro or TORQUE server machine
`SubmitArguments`	Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`ccsscheduler`, `genericscheduler`, `jobmanager`, `lsfscheduler`, `mpiexec`, `torquescheduler`

Purpose

Connect to schedulers when client utilities are not available locally

Constructor

```
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. 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 string 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 boolean 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 the integration scripts provided in *`matlabroot`*`/toolbox/distcomp/examples/integration`. For example, the scripts for PBS in a nonshared file system are in

```
matlabroot/toolbox/distcomp/examples/integration/pbs/nonshared
```

Methods

Method Name	Description
connect	RemoteClusterAccess.connect(clusterHost) establishes a connection to the specified host using the user credential options supplied in the constructor. File mirroring is not supported. RemoteClusterAccess.connect(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.
disconnect	RemoteClusterAccess.disconnect() disconnects the existing remote connection. The connect method must have already been called.
doLastMirrorForJob	RemoteClusterAccess.doLastMirrorForJob(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	RemoteClusterAccess.getRemoteJobLocation(jobID, remoteOS) returns the full path to the remote job location for the supplied jobID. Valid values for remoteOS are 'pc' and 'unix'.
isJobUsingConnection	RemoteClusterAccess.isJobUsingConnection(jobID) returns true if the job is currently being mirrored.

Method Name	Description
`resumeMirrorForJob`	`RemoteClusterAccess.resumeMirrorForJob(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] = RemoteClusterAccess.runCommand(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`	`RemoteClusterAccess.startMirrorForJob(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`	`RemoteClusterAccess.stopMirrorForJob(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`.

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
`DataLocation`	Location on the remote host for files that are being mirrored.
`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.
`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');
remoteConnection.connect('headnode1','/tmp/filemirror');
remoteConnection.startMirrorForJob(job);
submit(job)
% Wait for the job to finish
job.wait();

% Ensure that all the local files are up to date, and remove the
% remote files
remoteConnection.doLastMirrorForJob(job);
```

```
% Get the output arguments for the job
results = job.getAllOutputArguments()
```

For more detailed examples, see the integration scripts provided in *`matlabroot`*`/toolbox/distcomp/examples/integration`. For example, the scripts for PBS in a nonshared file system are in

`matlabroot``/toolbox/distcomp/examples/integration/pbs/nonshared`

Purpose

Define job behavior and properties when using local or third-party scheduler

Constructor

`createJob`

Container Hierarchy

Parent	`ccsscheduler`, `genericscheduler`, `localscheduler`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`, or `torquescheduler` object
Children	`simpletask` objects

Description

A simplejob object contains all the tasks that define what each worker does as part of the complete job execution. A simplejob object is used only with a local or third-party scheduler.

Methods

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`diary`	Display or save Command Window text of batch job
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler

`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Properties

`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object
`Tasks`	Tasks contained in job object

`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also

`job`, `paralleljob`, `simpleparalleljob`

simplematlabpooljob

Purpose

Define MATLAB pool job behavior and properties when using local or third-party scheduler

Constructor

`createMatlabPoolJob`

Container Hierarchy

Parent	`ccsscheduler`, `genericscheduler`, `localscheduler`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`, or `torquescheduler` object
Children	`simpletask` object

Description

A simplematlabpooljob object contains all the information needed to define what each lab does as part of the complete job execution. A MATLAB pool job uses one worker in a MATLAB pool to run a parallel job on the other labs of the pool. A simplematlabpooljob object is used only with a local or third-party scheduler.

Methods

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`diary`	Display or save Command Window text of batch job
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler

`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Properties

`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`MaximumNumberOfWorkers`	Specify maximum number of workers to perform job tasks
`MinimumNumberOfWorkers`	Specify minimum number of workers to perform job tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object

`Task`	First task contained in MATLAB pool job object
`Tasks`	Tasks contained in job object
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also

`matlabpooljob`, `paralleljob`, `simpleparalleljob`

Purpose

Define parallel job behavior and properties when using local or third-party scheduler

Constructor

`createParallelJob`

Container Hierarchy

Parent	`ccsscheduler`, `genericscheduler`, `localscheduler`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`, or `torquescheduler` object
Children	`simpletask` objects

Description

A simpleparalleljob object contains all the tasks that define what each lab does as part of the complete job execution. A parallel job runs simultaneously on all labs and uses communication among the labs during task evaluation. A simpleparalleljob object is used only with a local or third-party scheduler.

Methods

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`diary`	Display or save Command Window text of batch job
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`load`	Load workspace variables from batch job
`submit`	Queue job in scheduler

`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Properties

`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`MaximumNumberOfWorkers`	Specify maximum number of workers to perform job tasks
`MinimumNumberOfWorkers`	Specify minimum number of workers to perform job tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object

`Tasks`	Tasks contained in job object
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

See Also `job`, `paralleljob`, `simplejob`

simpletask

Purpose

Define task behavior and properties when using local or third-party scheduler

Constructor

`createTask`

Container Hierarchy

Parent	`simplejob`, `simplematlabpooljob`, or `simpleparalleljob` object
Children	None

Description

A simpletask object defines what each lab or worker does as part of the complete job execution. A simpletask object is used only with a local or third-party scheduler.

Methods

`cancel`	Cancel job or task
`destroy`	Remove job or task object from parent and memory
`waitForState`	Wait for object to change state

Properties

`CaptureCommandWindowOutput`	Specify whether to return Command Window output
`CommandWindowOutput`	Text produced by execution of task object's function
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`Error`	Task error information
`ErrorIdentifier`	Task error identifier
`ErrorMessage`	Message from task error

`FinishTime`	When task or job finished
`Function`	Function called when evaluating task
`ID`	Object identifier
`InputArguments`	Input arguments to task object
`Name`	Name of job manager, job, or worker object
`NumberOfOutputArguments`	Number of arguments returned by task function
`OutputArguments`	Data returned from execution of task
`Parent`	Parent object of job or task
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`UserData`	Specify data to associate with object

See Also

`task`

task

Purpose

Define task behavior and properties when using job manager

Constructor

`createTask`

Container Hierarchy

Parent	`job`, `matlabpooljob`, or `paralleljob` object
Children	None

Description

A task object defines what each lab or worker does as part of the complete job execution. A task object is used only with a job manager as scheduler.

Methods

`cancel`	Cancel job or task
`destroy`	Remove job or task object from parent and memory
`waitForState`	Wait for object to change state

Properties

`AttemptedNumberOfRetries`	Number of times failed task was rerun
`CaptureCommandWindowOutput`	Specify whether to return Command Window output
`CommandWindowOutput`	Text produced by execution of task object's function
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`Error`	Task error information
`ErrorIdentifier`	Task error identifier

`ErrorMessage`	Message from task error
`FailedAttemptInformation`	Information returned from failed task
`FinishedFcn`	Specify callback to execute after task or job runs
`FinishTime`	When task or job finished
`Function`	Function called when evaluating task
`ID`	Object identifier
`InputArguments`	Input arguments to task object
`MaximumNumberOfRetries`	Specify maximum number of times to rerun failed task
`NumberOfOutputArguments`	Number of arguments returned by task function
`OutputArguments`	Data returned from execution of task
`Parent`	Parent object of job or task
`RunningFcn`	Specify function file to execute when job or task starts running
`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`Timeout`	Specify time limit to complete task or job
`UserData`	Specify data to associate with object
`Worker`	Worker session that performed task

See Also `simpletask`

Purpose

Access TORQUE scheduler

Constructor

`findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description

A torquescheduler object provides access to your network's TORQUE scheduler, which controls the job queue, and distributes job tasks to workers or labs for execution.

Methods

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in scheduler
`getDebugLog`	Read output messages from job run in CJS cluster
`setupForParallelExecution`	Set options for submitting parallel jobs to scheduler

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler

`Configuration`	Specify configuration to apply to object or toolbox function
`DataLocation`	Specify folder where job data is stored
`HasSharedFilesystem`	Specify whether nodes share data location
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`ParallelSubmissionWrapperScript`	Script that scheduler runs to start labs
`RcpCommand`	Command to copy files from client
`ResourceTemplate`	Resource definition for PBS Pro or TORQUE scheduler
`RshCommand`	Remote execution command used on worker nodes during parallel job
`ServerName`	Name of current PBS Pro or TORQUE server machine
`SubmitArguments`	Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`ccsscheduler`, `genericscheduler`, `jobmanager`, `lsfscheduler`, `mpiexec`, `pbsproscheduler`

Purpose Access information about MATLAB worker session

Constructor `getCurrentWorker`

Container Hierarchy

Parent	`jobmanager` object
Children	None

Description A worker object represents the MATLAB worker session that evaluates tasks in a job scheduled by a job manager. Only worker sessions started with the `startworker` script can be represented by a worker object.

Methods None

Properties

`Computer`	Information about computer on which worker is running
`CurrentJob`	Job whose task this worker session is currently evaluating
`CurrentTask`	Task that worker is currently running
`HostAddress`	IP address of host running job manager or worker session
`Hostname`	Name of host running job manager or worker session
`JobManager`	Job manager that this worker is registered with
`Name`	Name of job manager, job, or worker object
`PreviousJob`	Job whose task this worker previously ran

`PreviousTask`	Task that this worker previously ran
`State`	Current state of task, job, job manager, or worker

See Also

`jobmanager`, `simpletask`, `task`

Function Reference

Parallel Code Execution

Parallel Code on a MATLAB Pool

`batch`	Run MATLAB script or function on worker
`Composite`	Create Composite object
`distributed`	Create distributed array from data in client workspace
`matlabpool`	Open or close pool of MATLAB sessions for parallel computation
`parfor`	Execute code loop in parallel
`spmd`	Execute code in parallel on MATLAB pool

Profiles, Input, and Output

`diary`	Display or save Command Window text of batch job
`exist`	Check whether Composite is defined on labs
`load`	Load workspace variables from batch job
`parallel.clusterProfiles`	Names of all available cluster profiles
`parallel.defaultClusterProfile`	Examine or set default cluster profile

`parallel.exportProfile`	Export one or more profiles to file
`parallel.importProfile`	Import cluster profiles from file
`pctRunOnAll`	Run command on client and all workers in matlabpool
`subsasgn`	Subscripted assignment for Composite
`subsref`	Subscripted reference for Composite

Interactive Functions

`mpiprofile`	Profile parallel communication and execution times
`pmode`	Interactive Parallel Command Window

Distributed and Codistributed Arrays

Toolbox Functions

`codistributed`	Create codistributed array from replicated local data
`codistributed.build`	Create codistributed array from distributed data
`codistributed.colon`	Distributed colon operation
`codistributor`	Create codistributor object for codistributed arrays
`codistributor1d`	Create 1-D codistributor object for codistributed arrays
`codistributor1d.defaultPartition`	Default partition for codistributed array
`codistributor2dbc`	Create 2-D block-cyclic codistributor object for codistributed arrays
`codistributor2dbc.defaultLabGrid`	Default computational grid for 2-D block-cyclic distributed arrays
`for`	`for`-loop over distributed range
`gather`	Transfer distributed array data 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

`isaUnderlying`	True if distributed array's underlying elements are of specified class
`isComplete`	True if codistributor object is complete
`isreplicated`	True for replicated array
`redistribute`	Redistribute codistributed array with another distribution scheme

Overloaded MATLAB Functions

`codistributed.cell`	Create codistributed cell array
`codistributed.eye`	Create codistributed identity matrix
`codistributed.false`	Create codistributed false array
`codistributed.Inf`	Create codistributed array of `Inf` values
`codistributed.NaN`	Create codistributed array of Not-a-Number values
`codistributed.ones`	Create codistributed array of ones
`codistributed.rand`	Create codistributed array of uniformly distributed pseudo-random numbers
`codistributed.randn`	Create codistributed array of normally distributed random values
`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
`codistributed.true`	Create codistributed true array
`codistributed.zeros`	Create codistributed array of zeros
`distributed.cell`	Create distributed cell array
`distributed.eye`	Create distributed identity matrix
`distributed.false`	Create distributed false array
`distributed.Inf`	Create distributed array of `Inf` values
`distributed.NaN`	Create distributed array of Not-a-Number values
`distributed.ones`	Create distributed array of ones
`distributed.rand`	Create distributed array of uniformly distributed pseudo-random numbers
`distributed.randn`	Create distributed array of normally distributed random values
`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
`distributed.true`	Create distributed true array
`distributed.zeros`	Create distributed array of zeros
`sparse`	Create sparse distributed or codistributed matrix

Jobs and Tasks

Job Creation

`createJob`	Create distributed or independent job
`createMatlabPoolJob`	Create MATLAB pool job
`createParallelJob`	Create parallel job object
`createTask`	Create new task in job
`dfeval`	Evaluate function using cluster
`dfevalasync`	Evaluate function asynchronously using cluster
`findResource`	Find available parallel computing resources
`jobStartup`	File for user-defined options to run when job starts
`mpiLibConf`	Location of MPI implementation
`mpiSettings`	Configure options for MPI communication
`pctconfig`	Configure settings for Parallel Computing Toolbox client session
`poolStartup`	File for user-defined options to run on each worker when MATLAB pool starts

`setupForParallelExecution`	Set options for submitting parallel jobs to scheduler
`taskFinish`	User-defined options to run on worker when task finishes
`taskStartup`	User-defined options to run on worker when task starts

Job Management

`cancel`	Cancel job or task
`changePassword`	Prompt user to change job manager password or MJS password
`clearLocalPassword`	Delete local store of user's job manager password
`demote`	Demote job in cluster queue
`destroy`	Remove job or task object from parent and memory
`findJob`	Find job objects stored in scheduler
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`getDebugLog`	Read output messages from job run in CJS cluster
`getJobSchedulerData`	Get specific user data for job on generic scheduler
`pause`	Pause job manager queue
`promote`	Promote job in MJS cluster queue
`resume`	Resume processing queue in job manager
`setJobClusterData`	Set specific user data for job on generic cluster

`setJobSchedulerData`	Set specific user data for job on generic scheduler
`submit`	Queue job in scheduler
`wait`	Wait for job to change state or for GPU calculation to complete
`waitForState`	Wait for object to change state

Task Execution Information

`getAttachedFilesFolder`	Folder into which AttachedFiles are written
`getCurrentCluster`	Cluster object that submitted current task
`getCurrentJob`	Job object whose task is currently being evaluated
`getCurrentJobmanager`	Job manager object that scheduled current task
`getCurrentTask`	Task object currently being evaluated in this worker session
`getCurrentWorker`	Worker object currently running this session
`getFileDependencyDir`	Directory where `FileDependencies` are written on worker machine

Object Control

`clear`	Remove objects from MATLAB workspace
`get`	Object properties
`inspect`	Open Property Inspector
`length`	Length of object array

`methods`	List functions of object class
`set`	Configure or display object properties
`size`	Size of object array

Interlab Communication Within a Communicating Job

`gcat`	Global concatenation
`gop`	Global operation across all labs
`gplus`	Global addition
`labBarrier`	Block execution until all labs reach this call
`labBroadcast`	Send data to all labs or receive data sent to all labs
`labindex`	Index of this lab
`labProbe`	Test to see if messages are ready to be received from other lab
`labReceive`	Receive data from another lab
`labSend`	Send data to another lab
`labSendReceive`	Simultaneously send data to and receive data from another lab
`numlabs`	Total number of labs operating in parallel on current job
`pload`	Load file into parallel session
`psave`	Save data from parallel job session

Graphics Processing Unit

`arrayfun`	Apply function to each element of array on GPU
`bsxfun`	Binary singleton expansion function for GPUArray
`existsOnGPU`	Determine if GPUArray or CUDAKernel is available on GPU
`feval`	Evaluate kernel on GPU
`gather`	Transfer distributed array data or GPUArray to local workspace
`gpuArray`	Create array on GPU
`gpuDevice`	Query or select GPU device
`gpuDeviceCount`	Number of GPU devices present
`parallel.gpu.CUDAKernel`	Create GPU CUDA kernel object from PTX and CU code
`reset`	Reset GPU device and clear its memory
`setConstantMemory`	Set some constant memory on GPU
`wait`	Wait for job to change state or for GPU calculation to complete

Utilities

`help`	Help for toolbox functions in Command Window
`pctRunDeployedCleanup`	Clean up after deployed parallel applications

Functions — Alphabetical List

arrayfun

Purpose

Apply function to each element of array on GPU

Syntax

```
A = arrayfun(FUN, B)
A = arrayfun(FUN, B, C, ...)
[A, B, ...] = arrayfun(FUN, C, ...)
```

Description

This method of a GPUArray object is very similar in behavior to the MATLAB function `arrayfun`, except that the actual evaluation of the function happens on the GPU, not on the CPU. Thus, any required data not already on the GPU is moved to GPU memory, the MATLAB function passed in for evaluation is compiled for the GPU, and then executed on the GPU. All the output arguments return as GPUArray objects, whose data you can retrieve with the `gather` method.

`A = arrayfun(FUN, B)` applies the function specified by `FUN` to each element of the GPUArray `B`, and returns the results in GPUArray `A`. `A` is the same size as `B`, and `A(i,j,...)` is equal to `FUN(B(i,j,...))`. `FUN` is a function handle to a function that takes one input argument and returns a scalar value. `FUN` must return values of the same class each time it is called. The input data must be a arrays of one of the following types: `single`, `double`, `int32`, `uint32`, `logical`, or GPUArray. The order in which `arrayfun` computes elements of `A` is not specified and should not be relied on.

`FUN` must be a handle to a function that is written in the MATLAB language (i.e., not a built-in function or a mex function); it must not be a nested, anonymous, or sub-function; and the MATLAB file that defines the function must contain exactly one function definition.

The subset of the MATLAB language that is currently supported for execution on the GPU can be found in “Execute MATLAB Code on a GPU” on page 10-12.

`A = arrayfun(FUN, B, C, ...)` evaluates `FUN` using elements of arrays `B`, `C`, ... as input arguments. The resulting GPUArray element `A(i,j,...)` is equal to `FUN(B(i,j,...), C(i,j,...), ...)`. The inputs `B`, `C`, ... must all have the same size or be scalar. Any scalar inputs are scalar expanded before being input to the function `FUN`.

One or more of the inputs B, C, ... must be a GPUArray; any of the others can reside in CPU memory. Each array that is held in CPU memory is converted to a GPUArray before calling the function on the GPU. If you plan to use an array in several different arrayfun calls, it is more efficient to convert that array to a GPUArray before making the series of calls to arrayfun.

[A, B, ...] = arrayfun(FUN, C, ...), where FUN is a function handle to a function that returns multiple outputs, returns GPUArrays A, B, ..., each corresponding to one of the output arguments of FUN. arrayfun calls FUN each time with as many outputs as there are in the call to arrayfun. FUN can return output arguments having different classes, but the class of each output must be the same each time FUN is called. This means that all elements of A must be the same class; B can be a different class from A, but all elements of B must be of the same class, etc.

Although the MATLAB arrayfun function allows you to specify optional parameter name/value pairs, the GPUArray arrayfun method does not support these options.

Examples

If you define a MATLAB function as follows:

```
function [o1, o2] = aGpuFunction(a, b, c)
o1 = a + b;
o2 = o1 .* c + 2;
```

You can evaluate this on the GPU.

```
s1 = gpuArray(rand(400));
s2 = gpuArray(rand(400));
s3 = gpuArray(rand(400));
[o1, o2] = arrayfun(@aGpuFunction, s1, s2, s3);
whos
  Name         Size          Bytes  Class

  o1        400x400            108  parallel.gpu.GPUArray
  o2        400x400            108  parallel.gpu.GPUArray
  s1        400x400           108  parallel.gpu.GPUArray
  s2        400x400           108  parallel.gpu.GPUArray
  s3        400x400           108  parallel.gpu.GPUArray
```

Use `gather` to retrieve the data from the GPU to the MATLAB workspace.

```
d = gather(o2);
```

See Also

`bsxfun` | `gather` | `gpuArray`

Purpose

Run MATLAB script or function on worker

Syntax

```
j = batch('aScript')
j = batch(myCluster, 'aScript')
j = batch(fcn, N, {x1, ..., xn})
j = batch(myCluster, fcn, N, {x1, ..., xn})
j = batch(..., 'p1', v1, 'p2', v2, ...)
```

Arguments

`j`	The batch job object.
`'aScript'`	The script of MATLAB code to be evaluated by the MATLAB pool job.
`myCluster`	Cluster object representing cluster compute resources.
`fcn`	Function handle or string of function name to be evaluated by the MATLAB pool job.
`N`	The number of output arguments from the evaluated function.
`{x1, ..., xn}`	Cell array of input arguments to the function.
p1, *p2*	Object properties or other arguments to control job behavior.
`v1`, `v2`	Initial values for corresponding object properties or arguments.

Description

`j = batch('aScript')` runs the script `aScript.m` on a worker in the cluster defined in the default cluster profile. The function returns `j`, a handle to the job object that runs the script. The script file `aScript.m` is added to the `AttachedFiles` property of the job and copied to the worker.

`j = batch(myCluster, 'aScript')` is identical to `batch('aScript')` except that the script runs on a worker according to the cluster identified by the cluster object `myCluster`.

`j = batch(fcn, N, {x1, ..., xn})` runs the function specified by a function handle or function name, `fcn`, on a worker in the cluster identified 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`, returning `N` output arguments. The function file for `fcn` is added to the `AttachedFiles` property of the job and copied to the worker.

`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 identified by the cluster object `myCluster`.

`j = batch(..., '`*`p1`*`', v1, '`*`p2`*`', v2, ...)` allows additional parameter-value pairs that modify the behavior of the job. These parameters support batch for functions and scripts, unless otherwise indicated. The accepted parameters are:

- `'Workspace'` — A 1-by-1 struct to define the workspace on the worker just before the script is called. 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.
- `'Profile'` — A single string that is the name of a cluster profile to use to identify the cluster. If this option is omitted, the default profile is used to identify the cluster and is applied to the job and task properties.
- `'AdditionalPaths'` — A string or cell array of strings that defines paths to be added to the workers' search path before the script or function is executed.
- `'AttachedFiles'` — A string or cell array of strings. Each string in the list identifies either a file or a folder, which is transferred to

the worker. The script being run is always added to the list of files sent to the worker.

- `'CurrentFolder'` — A string indicating in what folder the script executes. There is no guarantee that this folder exists on the worker. The default value for this property is the `cwd` of MATLAB when the `batch` command is executed. If the string for this argument is `'.'`, there is no change in folder before batch execution.
- `'CaptureDiary'` — A boolean flag to indicate that the toolbox should collect the diary from the function call. See the `diary` function for information about the collected data. The default is `true`.
- `'Matlabpool'` — A nonnegative scalar integer that defines the number of workers to make into a MATLAB pool for the job to run on *in addition* to the worker running the batch job itself. The script or function uses this pool for execution of statements such as `parfor` and `spmd` that are inside the batch code. A value of `N` for the property `Matlabpool` is effectively the same as adding a call to `matlabpool N` into the code. Because the MATLAB pool requires `N` workers in addition to the worker running the batch, there must be at least `N+1` workers available on the cluster. (See "Run a Batch Parallel Loop" on page 1-9.) The default value is 0, which causes the script or function to run on only the single worker without a MATLAB pool.

Tips

As a matter of good programming practice, when you no longer need it, you should delete the job created by the batch function so that it does not continue to consume cluster storage resources.

Examples

Run a batch script on a worker, without using a MATLAB pool:

```
 j = batch('script1');
```

Run a batch script that requires two additional files for execution:

```
j = batch('myScript','AttachedFiles',{'mscr1.m','mscr2.m'});
wait(j);
load(j);
```

Run a batch MATLAB pool job on a remote cluster, using eight workers for the MATLAB pool in addition to the worker running the batch script. Capture the diary, and load the results of the job into the workspace. This job requires a total of nine workers:

```
 j = batch('script1', 'matlabpool', 8, 'CaptureDiary', true);
 wait(j);   % Wait for the job to finish
 diary(j)   % Display the diary
 load(j)    % Load job workspace data into client workspace
```

Run a batch MATLAB pool job on a local worker, which employs two other local workers for the pool. Note, this requires a total of three workers in addition to the client, all on the local machine:

```
 j = batch('script1', 'Profile', 'local', ...
     'matlabpool', 2);
```

Clean up a batch job's data after you are finished with it:

```
delete(j)
```

Run a batch function on a cluster that generates a 10-by-10 random matrix:

```
c = parcluster();
j = batch(c, @rand, 1, {10, 10});

wait(j)   % Wait for the job to finish
diary(j)  % Display the diary

r = fetchOutputs(j) % Get results into a cell array
r{1}                % Display result
```

See Also

diary | findJob | load | wait

Purpose

Binary singleton expansion function for GPUArray

Syntax

```
C = bsxfun(FUN, A, B)
```

Description

This method of a GPUArray object is similar in behavior to the MATLAB function `bsxfun`, except that the actual evaluation of the function happens on the GPU, not on the CPU.

`C = bsxfun(FUN, A, B)` applies the element-by-element binary operation specified by the function handle FUN to arrays `A` and `B`, with singleton expansion enabled. If `A` or `B` is a GPUArray, `bsxfun` moves all other required data to the GPU and performs its calculation on the GPU. The output array `C` is a GPUArray, whose data you can retrieve with `gather`. `FUN` must be a handle to one of the following built-in functions:

```
@plus     Plus
@minus    Minus
@times    Array multiply
@rdivide  Right array divide
@ldivide  Left array divide
@power    Array power

@max      Binary maximum
@min      Binary minimum

@rem      Remainder after division
@mod      Modulus after division

@atan2    Four-quadrant inverse tangent
@hypot    Square root of sum of squares

@eq       Equal
@ne       Not equal
@lt       Less than
@le       Less than or equal
@gt       Greater than
@ge       Greater than or equal
```

`@and`	Element-wise logical AND
`@or`	Element-wise logical OR
`@xor`	Logical EXCLUSIVE OR

The corresponding dimensions of `A` and `B` must be equal to each other, or equal to one. Whenever a dimension of `A` or `B` is singleton (equal to 1), `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 0.

The size of the output array `C` is such that each dimension is the larger of the two input arrays in that dimension for nonzero size, or zero otherwise. Notice in the following code how dimensions of size 1 are scaled up or down to match the size of the corresponding dimension in the other argument:

```
R1 = parallel.gpu.GPUArray.rand(2,5,4);
R2 = parallel.gpu.GPUArray.rand(2,1,4,3);
R = bsxfun(@plus,R1,R2);
size(R)
  2     5     4     3

R1 = parallel.gpu.GPUArray.rand(2,2,0,4);
R2 = parallel.gpu.GPUArray.rand(2,1,1,4);
R = bsxfun(@plus,R1,R2);
size(R)
  2     2     0     4
```

Examples

Subtract the column means from each element of a matrix:

```
A = parallel.gpu.GPUArray.rand(8);
M = bsxfun(@minus, A, mean(A));
```

See Also

`arrayfun` | `gather` | `gpuArray`

Purpose

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 in a job manager, any worker sessions that are evaluating tasks belonging to the job object will be restarted.

Examples

Cancel a task. Note afterward the task's `State`, `ErrorMessage`, and `OutputArguments` properties.

```
job1 = createJob(jm);
t = createTask(job1, @rand, 1, {3,3});
cancel(t)
get(t)
                           ID: 1
                     Function: @rand
      NumberOfOutputArguments: 1
               InputArguments: {[3]  [3]}
              OutputArguments: {1x0 cell}
   CaptureCommandWindowOutput: 0
          CommandWindowOutput: ''
```

```
           State: 'finished'
    ErrorMessage: 'Task cancelled by user'
 ErrorIdentifier: 'distcomp:task:Cancelled'
         Timeout: Inf
      CreateTime: 'Fri Oct 22 11:38:39 EDT 2004'
       StartTime: 'Fri Oct 22 11:38:46 EDT 2004'
      FinishTime: 'Fri Oct 22 11:38:46 EDT 2004'
          Worker: []
          Parent: [1x1 distcomp.job]
        UserData: []
      RunningFcn: []
     FinishedFcn: []
```

See Also

`destroy` | `submit`

Purpose

Prompt user to change job manager password or MJS password

Syntax

```
changePassword(jm-sched)
changePassword(mjs-c)
changePassword(obj, username)
```

Arguments

`jm-sched`	Job manager scheduler object on which password is changing
`mjs-c`	MJS cluster object on which password is changing
`username`	User whose password is changed

Description

`changePassword(jm-sched)` or `changePassword(mjs-c)` prompts the user to change the password for the current user. The user's current password must be entered as well as the new password.

`changePassword(obj, username)` prompts the admin user to change the password for the specified user. The admin user's password must be entered as well as the user's new password. This enables the admin user to reset a password if the user has forgotten it.

For more information on job manager or MJS security, see "Set MJS Cluster Security".

See Also

`clearLocalPassword` | `logout`

clear

Purpose

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.

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.

Examples

This example creates two job objects on the job manager `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 job manager 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
```

See Also

`createJob` | `createTask` | `findJob` | `findTask` | `parcluster`

Purpose

Delete local store of user's job manager password

Syntax

```
clearLocalPassword(jm)
```

Arguments

`jm`	Job manager associated with the local password

Description

`clearLocalPassword(jm)` clears the current user's password on the local computer.

When you call a privileged action (for example, with `findResource`, `createJob`, or `submit`) on a job manager running with a higher security level, a popup dialog requires you to enter your user name and password for authentication. To prevent having to enter it for every subsequent privileged action, you can choose to have the password remembered in a local password store. The `clearLocalPassword` function removes your password from this local store. This means any subsequent call to a privileged action on this computer requires you to re-authenticate with a valid password. Clearing your password might be useful after you have finished working on a shared machine.

For more information on job manager security, see "Set MJS Cluster Security".

See Also

`changePassword`

Purpose

Create codistributed array from replicated local data

Syntax

```
C = codistributed(X)
C = codistributed(X, codist)
C = codistributed(X, codist, lab)
C = codistributed(C1, codist)
```

Description

`C = codistributed(X)` distributes a replicated `X` using the default codistributor. `X` must be a replicated array, that is, it must have the same value on all labs. `size(C)` is the same as `size(X)`.

`C = codistributed(X, codist)` distributes a replicated `X` using the codistributor `codist`. `X` must be a replicated array, namely it must have the same value on all labs. `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, codist, lab)` distributes a local array `X` that resides on the lab identified by `lab`, using the codistributor `codist`. Local array `X` must be defined on all labs, but only the value from `lab` is used to construct `C`. `size(C)` is the same as `size(X)`.

`C = codistributed(C1, codist)` where the input array `C1` is already a codistributed array, redistributes the array `C1` according to the distribution scheme defined by codistributor `codist`. This is the same as calling `C = redistribute(C1, codist)`. If the specified distribution scheme is that same as that already in effect, then the result is the same as the input.

Tips

`gather` essentially performs the inverse of `codistributed`.

Examples

Create a 1000-by-1000 codistributed array C1 using the default distribution scheme.

```
spmd
    N = 1000;
    X = magic(N);          % Replicated on every lab
    C1 = codistributed(X); % Partitioned among the labs
```

```
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
```

See Also

`codistributor1d` | `codistributor2dbc` | `gather` | `globalIndices` | `getLocalPart` | `redistribute` | `size` | `subsasgn` | `subsref`

Purpose

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 labs. 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 labs, 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 labs, globalInd has the values:
    % 1:251    on lab 1
    % 252:501  on lab 2
    % 502:751  on lab 3
    % 752:1001 on lab 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 | globalIndices | getLocalPart | redistribute | size | subsasgn | subsref

Purpose

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 labs,

```
spmd(4)
    C = codistributed.cell(1000);
end
```

creates a 1000-by-1000 distributed cell array `C`, distributed by its second dimension (columns). Each lab 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 lab contains a 10-by-`labindex` local piece of `C`.

See Also

`cell` | `distributed.cell`

Purpose

Distributed colon operation

Syntax

```
codistributed.colon(a,d,b)
codistributed.colon(a,b)
```

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 lab 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:

- `codist` — A codistributor object specifying the distribution scheme of the resulting vector. 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 vector, skipping some error checking steps.

Examples

Partition the vector `1:10` into four subvectors among four labs.

```
spmd(4); C = codistributed.colon(1,10), end
Lab 1:
  This lab stores C(1:3).
          LocalPart: [1 2 3]
      Codistributor: [1x1 codistributor1d]
Lab 2:
  This lab stores C(4:6).
          LocalPart: [4 5 6]
      Codistributor: [1x1 codistributor1d]
Lab 3:
  This lab stores C(7:8).
          LocalPart: [7 8]
      Codistributor: [1x1 codistributor1d]
Lab 4:
  This lab stores C(9:10).
          LocalPart: [9 10]
      Codistributor: [1x1 codistributor1d]
```

See Also colon | codistributor1d | codistributor2dbc | for

Purpose

Create codistributed identity matrix

Syntax

```
C = codistributed.eye(n)
C = codistributed.eye(m, n)
C = codistributed.eye([m, n])
C = eye(n, codist)
C = eye(m, n, codist)
C = eye([m, n], codist)
```

Description

`C = codistributed.eye(n)` creates an `n`-by-`n` codistributed identity matrix of underlying class double.

`C = codistributed.eye(m, n)` or `C = codistributed.eye([m, n])` creates an `m`-by-`n` codistributed matrix of underlying class double with ones on the diagonal and zeros elsewhere.

Optional arguments to `codistributed.eye` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `eye` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.
- `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.

`C = eye(n, codist)` is the same as `C = codistributed.eye(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
```

```
    C = eye(8, codistributor1d());
end
```

`C = eye(m, n, codist)` and `C = eye([m, n], codist)` are the same as `C = codistributed.eye(m, n)` and `C = codistributed.eye([m, n])`, respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    C = codistributed.eye(1000);
end
```

creates a 1000-by-1000 codistributed double array `C`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `C`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    C = eye(10, 10, 'uint16', codist);
end
```

creates a 10-by-10 codistributed uint16 array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`eye` | `codistributed.ones` | `codistributed.speye` | `codistributed.zeros` | `distributed.eye`

Purpose

Create codistributed false array

Syntax

```
F = codistributed.false(n)
F = codistributed.false(m, n, ...)
F = codistributed.false([m, n, ...])
F = false(n, codist)
F = false(m, n, ..., codist)
F = false([m, n, ...], codist)
```

Description

`F = codistributed.false(n)` creates an `n`-by-`n` codistributed array of logical zeros.

`F = codistributed.false(m, n, ...)` or `F = codistributed.false([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of logical zeros.

Optional arguments to `codistributed.false` 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.

`F = false(n, codist)` is the same as `F = codistributed.false(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
    F = false(8, codistributor1d());
end
```

`F = false(m, n, ..., codist)` and `F = false([m, n, ...], codist)` are the same as `F = codistributed.false(m, n, ...)` and `F = codistributed.false([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    F = false(1000, codistributor());
end
```

creates a 1000-by-1000 codistributed false array `F`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `F`.

```
spmd
    codist = codistributor('1d', 2, 1:numlabs);
    F = false(10, 10, codist);
end
```

creates a 10-by-10 codistributed false array `F`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `F`.

See Also

`false` | `codistributed.true` | `distributed.false`

Purpose

Create codistributed array of `Inf` values

Syntax

```
C = codistributed.Inf(n)
C = codistributed.Inf(m, n, ...)
C = codistributed.Inf([m, n, ...])
C = Inf(n, codist)
C = Inf(m, n, ..., codist)
C = Inf([m, n, ...], codist)
```

Description

`C = codistributed.Inf(n)` creates an `n`-by-`n` codistributed matrix of `Inf` values.

`C = codistributed.Inf(m, n, ...)` or `C = codistributed.Inf([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of `Inf` values.

Optional arguments to `codistributed.Inf` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `Inf` function: `'double'` (the default), or `'single'`.
- `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.

`C = Inf(n, codist)` is the same as `C = codistributed.Inf(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
    C = Inf(8, codistributor1d());
end
```

`C = Inf(m, n, ..., codist)` and `C = Inf([m, n, ...], codist)` are the same as `C = codistributed.Inf(m, n, ...)` and `C = codistributed.Inf([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    C = Inf(1000, codistributor())
end
```

creates a 1000-by-1000 codistributed double matrix `C`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `C`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    C = Inf(10, 10, 'single', codist);
end
```

creates a 10-by-10 codistributed single array `C`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `C`.

See Also

`Inf` | `codistributed.NaN` | `distributed.Inf`

Purpose

Create codistributed array of Not-a-Number values

Syntax

```
C = codistributed.NaN(n)
C = codistributed.NaN(m, n, ...)
C = codistributed.NaN([m, n, ...])
C = NaN(n, codist)
C = NaN(m, n, ..., codist)
C = NaN([m, n, ...], codist)
```

Description

`C = codistributed.NaN(n)` creates an `n`-by-`n` codistributed matrix of `NaN` values.

`C = codistributed.NaN(m, n, ...)` or `C = codistributed.NaN([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of `NaN` values.

Optional arguments to `codistributed.NaN` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `NaN` function: `'double'` (the default), or `'single'`.
- `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.

`C = NaN(n, codist)` is the same as `C = codistributed.NaN(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
    C = NaN(8, codistributor1d());
end
```

`C = NaN(m, n, ..., codist)` and `C = NaN([m, n, ...], codist)` are the same as `C = codistributed.NaN(m, n, ...)` and `C = codistributed.NaN([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    C = NaN(1000, codistributor())
end
```

creates a 1000-by-1000 codistributed double matrix `C` of `NaN` values, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `C`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    C = NaN(10, 10, 'single', codist);
end
```

creates a 10-by-10 codistributed single array `C`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `C`.

See Also

`NaN` | `codistributed.Inf` | `distributed.NaN`

Purpose

Create codistributed array of ones

Syntax

```
C = codistributed.ones(n)
C = codistributed.ones(m, n, ...)
C = codistributed.ones([m, n, ...])
C = ones(n, codist)
C = ones(m, n, codist)
C = ones([m, n], codist)
```

Description

`C = codistributed.ones(n)` creates an `n`-by-`n` codistributed matrix of ones of class double.

`C = codistributed.ones(m, n, ...)` or `C = codistributed.ones([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of ones.

Optional arguments to `codistributed.ones` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `ones` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.
- `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.

`C = ones(n, codist)` is the same as `C = codistributed.ones(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
```

```
    C = ones(8, codistributor1d());
end
```

`C = ones(m, n, codist)` and `C = ones([m, n], codist)` are the same as `C = codistributed.ones(m, n, ...)` and `C = codistributed.ones([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    C = codistributed.ones(1000, codistributor());
end
```

creates a 1000-by-1000 codistributed double array of ones, `C`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `C`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    C = ones(10, 10, 'uint16', codist);
end
```

creates a 10-by-10 codistributed uint16 array of ones, `C`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `C`.

See Also

`ones` | `codistributed.eye` | `codistributed.zeros` | `distributed.ones`

Purpose

Create codistributed array of uniformly distributed pseudo-random numbers

Syntax

```
R = codistributed.rand(n)
R = codistributed.rand(m, n, ...)
R = codistributed.rand([m, n, ...])
R = rand(n, codist)
R = rand(m, n, codist)
R = rand([m, n], codist)
```

Description

`R = codistributed.rand(n)` creates an `n`-by-`n` codistributed array of underlying class double.

`R = codistributed.rand(m, n, ...)` or `R = codistributed.rand([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of underlying class double.

Optional arguments to `codistributed.rand` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `rand` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.
- `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.

`R = rand(n, codist)` is the same as `R = codistributed.rand(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
    R = codistributed.rand(8, codistributor1d());
end
```

`R = rand(m, n, codist)` and `R = rand([m, n], codist)` are the same as `R = codistributed.rand(m, n, ...)` and `R = codistributed.rand([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Tips

When you use `rand` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

With four labs,

```
spmd(4)
    R = codistributed.rand(1000, codistributor())
end
```

creates a 1000-by-1000 codistributed double array `R`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `R`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    R = codistributed.rand(10, 10, 'uint16', codist);
end
```

creates a 10-by-10 codistributed uint16 array `R`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `R`.

See Also

`rand` | `codistributed.randn` | `codistributed.sprand` | `codistributed.sprandn` | `distributed.rand`

Purpose

Create codistributed array of normally distributed random values

Syntax

```
RN = codistributed.randn(n)
RN = codistributed.randn(m, n, ...)
RN = codistributed.randn([m, n, ...])
RN = randn(n, codist)
RN = randn(m, n, codist)
RN = randn([m, n], codist)
```

Description

`RN = codistributed.randn(n)` creates an `n`-by-`n` codistributed array of normally distributed random values with underlying class double.

`RN = codistributed.randn(m, n, ...)` and `RN = codistributed.randn([m, n, ...])` create an `m`-by-`n`-by-... codistributed array of normally distributed random values.

Optional arguments to `codistributed.randn` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `rand` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.
- `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.

`RN = randn(n, codist)` is the same as `RN = codistributed.randn(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
```

```
    RN = codistributed.randn(8, codistributor1d());
end
```

`RN = randn(m, n, codist)` and `RN = randn([m, n], codist)` are the same as `RN = codistributed.randn(m, n, ...)` and `RN = codistributed.randn([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Tips

When you use `randn` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

With four labs,

```
spmd(4)
    RN = codistributed.randn(1000);
end
```

creates a 1000-by-1000 codistributed double array `RN`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `RN`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    RN = randn(10, 10, 'uint16', codist);
end
```

creates a 10-by-10 codistributed uint16 array `RN`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `RN`.

See Also

`randn` | `codistributed.rand` | `codistributed.sprand` | `codistributed.sprandn` | `distributed.randn`

Purpose

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 labs 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

`spalloc` | `sparse` | `distributed.spalloc`

codistributed.speye

Purpose

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 labs,

```
spmd(4)
    CS = speye(1000, codistributor())
end
```

creates a 1000-by-1000 sparse codistributed double array `CS`, distributed by its second dimension (columns). Each lab 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 lab contains a 10-by-`labindex` local piece of `CS`.

See Also

`speye` | `distributed.speye` | `sparse`

codistributed.sprand

Purpose

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
```

Tips

When you use `sprand` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

With four labs,

```
spmd(4)
    CS = codistributed.sprand(1000, 1000, .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 lab contains a 1000-by-250 local piece of `CS`.

```
spmd(4)
    codist = codistributor1d(2, 1:numlabs);
    CS = sprand(10, 10, .1, codist);
end
```

creates a 10-by-10 codistributed double array `CS` with approximately 10 nonzeros. `CS` is distributed by its columns, and each lab contains a 10-by-`labindex` local piece of `CS`.

See Also

`sprand` | `codistributed.rand` | `distributed.sprandn`

codistributed.sprandn

Purpose

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
```

Tips

When you use `sprandn` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

With four labs,

```
spmd(4)
    CS = codistributed.sprandn(1000, 1000, .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 lab contains a 1000-by-250 local piece of `CS`.

```
spmd(4)
    codist = codistributor1d(2, 1:numlabs);
    CS = sprandn(10, 10, .1, codist);
end
```

creates a 10-by-10 codistributed double array `CS` with approximately 10 nonzeros. `CS` is distributed by its columns, and each lab contains a 10-by-`labindex` local piece of `CS`.

See Also

`sprandn | codistributed.rand | codistributed.randn | sparse | codistributed.speye | codistributed.sprand | distributed.sprandn`

codistributed.true

Purpose

Create codistributed true array

Syntax

```
T = codistributed.true(n)
T = codistributed.true(m, n, ...)
T = codistributed.true([m, n, ...])
T = true(n, codist)
T = true(m, n, ..., codist)
T = true([m, n, ...], codist)
```

Description

`T = codistributed.true(n)` creates an `n`-by-`n` codistributed array of logical ones.

`T = codistributed.true(m, n, ...)` or `T = codistributed.true([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of logical ones.

Optional arguments to `codistributed.true` 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.

`T = true(n, codist)` is the same as `T = codistributed.true(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
    T = true(8, codistributor1d());
end
```

T = true(m, n, ..., codist) and T = true([m, n, ...], codist) are the same as T = codistributed.true(m, n, ...) and T = codistributed.true([m, n, ...]), respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    T = true(1000, codistributor());
end
```

creates a 1000-by-1000 codistributed true array T, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of T.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs);
    T = true(10, 10, codist);
end
```

creates a 10-by-10 codistributed true array T, distributed by its columns. Each lab contains a 10-by-labindex local piece of T.

See Also

true | codistributed.false | distributed.true

codistributed.zeros

Purpose

Create codistributed array of zeros

Syntax

```
C = codistributed.zeros(n)
C = codistributed.zeros(m, n, ...)
C = codistributed.zeros([m, n, ...])
C = zeros(n, codist)
C = zeros(m, n, codist)
C = zeros([m, n], codist)
```

Description

`C = codistributed.zeros(n)` creates an `n`-by-`n` codistributed matrix of zeros of class double.

`C = codistributed.zeros(m, n, ...)` or `C = codistributed.zeros([m, n, ...])` creates an `m`-by-`n`-by-... codistributed array of zeros.

Optional arguments to `codistributed.zeros` must be specified after the required arguments, and in the following order:

- *`classname`* — Specifies the class of the codistributed array `C`. Valid choices are the same as for the regular `zeros` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.
- `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.

`C = zeros(n, codist)` is the same as `C = codistributed.zeros(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
```

```
    C = zeros(8, codistributor1d());
end
```

`C = zeros(m, n, codist)` and `C = zeros([m, n], codist)` are the same as `C = codistributed.zeros(m, n, ...)` and `C = codistributed.zeros([m, n, ...])`, respectively. You can also use the optional arguments with this syntax.

Examples

With four labs,

```
spmd(4)
    C = codistributed.zeros(1000, codistributor());
end
```

creates a 1000-by-1000 codistributed double array of zeros, `C`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `C`.

```
spmd(4)
    codist = codistributor('1d', 2, 1:numlabs)
    C = zeros(10, 10, 'uint16', codist);
end
```

creates a 10-by-10 codistributed uint16 array of zeros, `C`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `C`.

See Also

`zeros` | `codistributed.eye` | `codistributed.ones` | `distributed.zeros`

Purpose

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 string `'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 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 5-17.

`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 labs, 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, lab 1 contains a 2-by-1-by-4 segment, lab 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 labs, 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

Purpose

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 1-D codistributor 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 labs, 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 lab 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); % 1 spec 1st dimension (rows).
    C = codistributed.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 lab 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` | `codistributor1d` | `codistributor2dbc` | `redistribute`

codistributor1d.defaultPartition

Purpose

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 labs:

```
spmd
    P = codistributor1d.defaultPartition(10)
end
```

See Also

`codistributed` | `codistributed.colon` | `codistributor1d`

Purpose

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 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 5-17. The 2-D block-cyclic codistributor is used by the ScaLAPACK parallel matrix computation software library.

`codist = codistributor2dbc()` forms a 2-D block-cyclic 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 labs 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 = codistributed.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 lab 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`

Purpose

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` labs in the open MATLAB 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 MATLAB pool.

```
spmd
    grid = codistributor2dbc.defaultLabGrid
end
```

See Also

`codistributed` | `codistributor2dbc` | `numlabs`

Purpose

Create Composite object

Syntax

```
C = Composite()
C = Composite(nlabs)
```

Description

`C = Composite()` creates a Composite object on the client using labs from the MATLAB pool. The actual number of labs referenced by this Composite object depends on the size of the MATLAB 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 resource 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 labs to use. If `nlabs` is of size 2, it specifies the minimum and maximum number of labs to use. The actual number of labs used is the maximum number of labs compatible with the size of the MATLAB pool, and with other existing Composite objects. An error is thrown if the constraints on the number of labs 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

Create a Composite object with no defined entries, then assign its values:

```
c = Composite();  % One element per lab in the pool
for ii = 1:length(c)
    % Set the entry for each lab to zero
    c{ii} = 0;    % Value stored on each lab
end
```

See Also

`matlabpool | spmd`

Purpose

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.

`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 MATLAB pool available to run the body of `parfor` loops or `spmd` blocks.

`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

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);
```

Purpose

Create distributed or independent job

Syntax

```
obj = createJob(cluster)
obj = createJob()
obj = createJob(scheduler)
obj = createJob(..., 'p1', v1, 'p2', v2, ...)
job = createJob(..., 'Profile', 'profileName', ...)
obj = createJob(..., 'configuration', 'ConfigurationName',
   ...)
```

Arguments

`obj`	The job object.
`scheduler`	The scheduler object created by `findResource`.
`cluster`	The cluster object created by `parcluster`.
`p1`, *`p2`*	Object properties configured at object creation.
`v1`, `v2`	Initial values for corresponding object properties.

Description

Note `createJob` works with scheduler objects and their associated configurations, or with cluster objects and their associated profiles. Cluster objects and cluster profiles are introduced in R2012a. You can use a configuration name where a profile is expected, and that configuration is automatically converted into a profile for you; but other than this you should not mix the two interfaces.

`obj = createJob(cluster)` creates an independent job object for the identified cluster.

`obj = createJob()` creates a job using the scheduler identified by the default cluster profile and sets the property values of the job as specified in that profile.

`obj = createJob(scheduler)` creates a job object at the data location for the identified scheduler, or in the job manager. When you specify a `scheduler` without using the **`'configuration'`** option, no configuration is used, so no configuration properties are applied to the job object.

`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 `job` object reference page (if using a job manager or MJS) or `simplejob` object reference page (if using a third-party scheduler). If an invalid property name or property value is specified, the object will not be created.

Note that the property value pairs can be in any format supported by the `set` function, i.e., param-value string pairs, structures, and param-value cell array pairs. If a structure is used, the structure field names are job object property names and the field values specify the property values.

If you are using a third-party scheduler instead of a job manager, the job's data is stored in the location specified by the scheduler's `DataLocation` property, or the cluster's `JobStorageLocation` property.

`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 "Cluster Profiles" on page 6-12.

`obj = createJob(..., '`**`configuration`**`', 'ConfigurationName', ...)` creates a job object using the scheduler identified by the configuration and sets the property values of the job as specified in that configuration.

Examples

Independent 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);
```

Distributed Job

Construct a distributed job object using the default profile.

```
j2 = createJob();
```

Add tasks to the job.

```
for i = 1:10
    createTask(j2, @rand, 1, {10});
end
```

Run the job.

```
submit(j2);
```

Wait for the job to finish running, and retrieve the job results.

```
waitForState(j2);
out = getAllOutputArguments(j2);
```

Display the random matrix returned from the third task.

```
disp(out{3});
```

Destroy the job.

```
destroy(j2);
```

See Also

createParallelJob | createTask | findJob | findResource | parcluster | submit

Purpose

Create MATLAB pool job

Syntax

```
job = createMatlabPoolJob()
job = createMatlabPoolJob('p1', v1, 'p2', v2, ...)
job = createMatlabPoolJob(..., 'configuration',
   'ConfigurationName',...)
```

Arguments

`job`	The job object.
p1, *p2*	Object properties configured at object creation.
`v1`, `v2`	Initial values for corresponding object properties.

Description

`job = createMatlabPoolJob()` creates a MATLAB pool job using the scheduler identified by the default parallel configuration.

`job = createMatlabPoolJob('p1', v1, 'p2', v2, ...)` creates a MATLAB pool job with the specified property values. For a listing of the valid properties of the created object, see the `matlabpooljob` object reference page (if using a job manager) or `simplematlabpooljob` object reference page (if using a third-party scheduler). If an invalid property name or property value is specified, the object is not created. These values override any values in the default configuration.

`job = createMatlabPoolJob(..., '`**`configuration`**`', 'ConfigurationName',...)` creates a MATLAB pool job using the scheduler identified by the configuration and sets the property values of the job as specified in that configuration. For details about defining and applying configurations, see "Cluster Profiles" on page 6-12.

Examples

Construct a MATLAB pool job object.

```
j = createMatlabPoolJob('Name', 'testMatlabPooljob');
```

Add the task to the job.

```
createTask(j, @labindex, 1, {});
```

Set the number of workers required for parallel execution.

```
j.MinimumNumberOfWorkers = 5;
j.MaximumNumberOfWorkers = 10;
```

Run the job.

```
submit(j)
```

Wait until the job is finished.

```
waitForState(j, 'finished');
```

Retrieve the job results.

```
out = getAllOutputArguments(j);
```

Display the output.

```
celldisp(out);
```

Destroy the job.

```
destroy(j);
```

See Also

`createParallelJob` | `createTask` | `defaultParallelConfig` | `submit`

Purpose

Create parallel job object

Syntax

```
pjob = createParallelJob()
pjob = createParallelJob(scheduler)
pjob = createParallelJob(..., 'p1', v1, 'p2', v2, ...)
pjob = createParallelJob(..., 'configuration',
   'ConfigurationName',...)
```

Arguments

`pjob`	The parallel job object.
`scheduler`	The scheduler object created by `findResource`.
`p1`, *`p2`*	Object properties configured at object creation.
`v1`, `v2`	Initial values for corresponding object properties.

Description

`pjob = createParallelJob()` creates a parallel job using the scheduler identified by the default parallel configuration and sets the property values of the job as specified in the default configuration.

`pjob = createParallelJob(scheduler)` creates a parallel job object at the data location for the identified scheduler, or in the job manager. When you specify a `scheduler` without using the **`'configuration'`** option, no configuration is used, so no configuration properties are applied to the job object.

`pjob = createParallelJob(..., 'p1', v1, 'p2', v2, ...)` creates a parallel job object with the specified property values. For a listing of the valid properties of the created object, see the `paralleljob` object reference page (if using a job manager) or `simpleparalleljob` object reference page (if using a third-party scheduler). If an invalid property name or property value is specified, the object will not be created.

Property value pairs can be in any format supported by the set function, i.e., param-value string pairs, structures, and param-value cell array pairs. Future modifications to the job object result in a remote call to the job manager or modification to data at the scheduler's data location.

`pjob = createParallelJob(..., '`**`configuration`**`', 'ConfigurationName',...)` creates a parallel job object using the scheduler identified by the configuration and sets the property values of the job as specified in that configuration. For details about defining and applying configurations, see "Cluster Profiles" on page 6-12.

Examples

Construct a parallel job object using the default configuration.

```
pjob = createParallelJob();
```

Add the task to the job.

```
createTask(pjob, 'rand', 1, {3});
```

Set the number of workers required for parallel execution.

```
set(pjob,'MinimumNumberOfWorkers',3);
set(pjob,'MaximumNumberOfWorkers',3);
```

Run the job.

```
submit(pjob);
```

Wait for the job to finish running, and retrieve the job results.

```
waitForState(pjob);
out = getAllOutputArguments(pjob);
```

Display the random matrices.

```
celldisp(out);
out{1} =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
out{2} =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
out{3} =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
```

Destroy the job.

```
destroy(pjob);
```

See Also

`createJob` | `createTask` | `findJob` | `findResource` | `submit`

createTask

Purpose

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',...)
t = createTask(...,'configuration', 'ConfigurationName',...)
```

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 or parallel 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 `task` object reference page (if using a job manager) or `simpletask` object reference page (if using a third-party scheduler). If an invalid property name or property value is specified, the object will not be created.

Note that the property value pairs can be in any format supported by the `set` function, i.e., param-value string pairs, structures, and param-value cell array pairs. If a structure is used, the structure field names are task object property names and the field values specify the property values.

`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 "Cluster Profiles" on page 6-12.

`t = createTask(...,'configuration', 'ConfigurationName',...)` creates a task object with the property values specified in the configuration `ConfigurationName`.

Examples

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 = get(t, 'OutputArguments');
```

Show the 10-by-10 random matrix.

```
disp(taskoutput{1});
```

Create 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}});
```

See Also

`createJob` | `createParallelJob` | `findTask`

Purpose

Default parallel computing configuration

Syntax

```
[config, allconfigs] = defaultParallelConfig
[oldconfig, allconfigs] = defaultParallelConfig(newconfig)
```

Arguments

`config`	String indicating name of current default configuration
`allconfigs`	Cell array of strings indicating names of all available configurations
`oldconfig`	String indicating name of previous default configuration
`newconfig`	String specifying name of new default configuration

Description

The `defaultParallelConfig` function allows you to programmatically get or set the default parallel configuration and obtain a list of all valid configurations.

`[config, allconfigs] = defaultParallelConfig` returns the name of the default parallel computing configuration, as well as a cell array containing the names of all available configurations.

`[oldconfig, allconfigs] = defaultParallelConfig(newconfig)` sets the default parallel computing configuration to `newconfig` and returns the previous default configuration and a cell array containing the names of all available configurations. The previous configuration is provided so that you can reset the default configuration to its original setting at the end of your session.

The settings specified for `defaultParallelConfig` are saved as a part of your MATLAB preferences.

The cell array `allconfigs` always contains a configuration called `'local'` for the local scheduler. The default configuration returned by `defaultParallelConfig` is guaranteed to be found in `allconfigs`.

If the default configuration has been deleted, or if it has never been set, `defaultParallelConfig` returns `'local'` as the default configuration.

Examples

Read the name of the default parallel configuration that is currently in effect, and get a listing of all available configurations.

```
[ConfigNow ConfigList] = defaultParallelConfig
```

Select the configuration named `'MyConfig'` to be the default parallel configuration.

```
defaultParallelConfig('MyConfig')
```

See Also

`findResource` | `importParallelConfig` | `matlabpool` | `pmode`

Purpose

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

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 the cluster identified by the profile `myProfile`:

```
myCluster = parcluster('myProfile');
delete(myCluster.Jobs)
```

demote

Purpose

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.

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)`.

Examples

Create and submit multiple jobs to the job manager identified by the default parallel configuration:

```
c = parcluster();
j1 = createJob(c,'Name','Job A'); createTask(j1,@rand,1,{3});
j2 = createJob(c,'Name','Job B'); createTask(j2,@rand,1,{3});
j3 = createJob(c,'Name','Job C'); createTask(j3,@rand,1,{3});
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'
```

See Also

createJob | findJob | promote | submit

destroy

Purpose

Remove job or task object from parent and memory

Syntax

```
destroy(obj)
```

Arguments

`obj`	Job or task object, or array of objects, to be deleted from job storage memory.

Description

`destroy(obj)` removes the job object reference or task object reference `obj` from the local session, and removes the object from the job manager memory. When `obj` is destroyed, it becomes an invalid object. You can remove an invalid object from the workspace with the `clear` command.

If multiple references to an object exist in the workspace, destroying one reference to that object invalidates all the remaining references to it. You should remove these remaining references from the workspace with the `clear` command.

If `obj` is an array of job objects and one of the objects cannot be destroyed, the other objects in the array are destroyed and a warning is returned.

The task objects contained in a job will also be destroyed when a job object is destroyed. This means that any references to those task objects will also be invalid.

Tips

Because its data is lost when you destroy an object, `destroy` should be used after output data has been retrieved from a job object.

Examples

Destroy a job and its tasks. Assume the default configuration identifies a job manager.

```
jm = findResource(); % Use default config
j = createJob(jm, 'Name', 'myjob');
t = createTask(j, @rand, 1, {10});
destroy(j);
clear t
clear j
```

Note that task `t` is also destroyed as part of job `j`.

Destroy all finished jobs on the job manager.

```
[p,q,r,f] = findJob(jm)
destroy(f)
```

See Also

`createJob` | `createTask` | `findJob` | `findTask`

dfeval

Purpose

Evaluate function using cluster

Syntax

```
[y1,...,ym] = dfeval(F, x1,...,xn)
y = dfeval( ..., 'P1',V1,'P2',V2,...)
[y1,...,ym] = dfeval(F, x1,...,xn, ... 'configuration',
   'ConfigurationName',...)
```

Arguments

`F`	Function name, function handle, or cell array of function names or handles.
`x1, ..., xn`	Cell arrays of input arguments to the functions.
`y1, ..., ym`	Cell arrays of output arguments; each element of a cell array corresponds to each task of the job.
`'P1', V1, 'P2', V2, ...`	Property name/property value pairs for the created job object; can be name/value pairs or structures.

Description

`[y1,...,ym] = dfeval(F, x1,...,xn)` performs the equivalent of an `feval` in a cluster of machines using Parallel Computing Toolbox software. `dfeval` evaluates the function `F`, with arguments provided in the cell arrays `x1,...,xn`. `F` can be a function handle, a function name, or a cell array of function handles/function names where the length of the cell array is equal to the number of tasks to be executed. `x1,...,xn` are the inputs to the function `F`, specified as cell arrays in which the number of elements in the cell array equals the number of tasks to be executed. The first task evaluates function `F` using the first element of each cell array as input arguments; the second task uses the second element of each cell array, and so on. The sizes of `x1,...,xn` must all be the same.

The results are returned to `y1,...,ym`, which are column-based cell arrays, each of whose elements corresponds to each task that was created. The number of cell arrays (`m`) is equal to the number of output arguments returned from each task. For example, if the job has 10 tasks that each generate three output arguments, the results of `dfeval` are three cell arrays of 10 elements each. When evaluation is complete, `dfeval` destroys the job.

`y = dfeval( ..., 'P1',V1,'P2',V2,...)` accepts additional arguments for configuring different properties associated with the job. Valid properties and property values are

- Job object property value pairs, specified as name/value pairs or structures. (Properties of other object types, such as scheduler, task, or worker objects are not permitted. Use a configuration to set scheduler and task properties.)
- `'`**`JobManager`**`','JobManagerName'`. This specifies the job manager on which to run the job. If you do not use this property to specify a job manager, the default is to run the job on the first job manager returned by `findResource`.
- `'`**`LookupURL`**`','host:port'`. This makes a unicast call to the job manager lookup service at the specified host and port. The job managers available for this job are those accessible from this lookup service. For more detail, see the description of this option on the `findResource` reference page.
- `'`**`StopOnError`**`',`**`true`**`|{`**`false`**`}`. You may also set the value to logical `1` (`true`) or `0` (`false`). If true (`1`), any error that occurs during execution in the cluster will cause the job to stop executing. The default value is `0` (`false`), which means that any errors that occur will produce a warning but will not stop function execution.

`[y1,...,ym] = dfeval(F, x1,...,xn, ... '`**`configuration`**`', 'ConfigurationName',...)` evaluates the function `F` in a cluster by using all the properties defined in the configuration `ConfigurationName`. The configuration settings are used to find and initialize a scheduler, create a job, and create tasks. For details about defining and applying configurations, see "Cluster Profiles" on page 6-12. Note that configurations enable you to use `dfeval` with any type of scheduler.

Note that dfeval runs synchronously (sync); that is, it does not return the MATLAB prompt until the job is completed. For further discussion of the usage of dfeval, see "Evaluate Functions Synchronously" on page 7-2.

Examples

Create three tasks that return a 1-by-1, a 2-by-2, and a 3-by-3 random matrix.

```
y = dfeval(@rand,{1 2 3})
y =
    [    0.9501]
    [2x2 double]
    [3x3 double]
```

Create two tasks that return random matrices of size 2-by-3 and 1-by-4.

```
y = dfeval(@rand,{2 1},{3 4});
y{1}
ans =
    0.8132    0.1389    0.1987
    0.0099    0.2028    0.6038
y{2}
ans =
    0.6154    0.9218    0.1763    0.9355
```

Create two tasks, where the first task creates a 1-by-2 random array and the second task creates a 3-by-4 array of zeros.

```
y = dfeval({@rand @zeros},{1 3},{2 4});
y{1}
ans =
    0.0579    0.3529
y{2}
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

Create five random 2-by-4 matrices using `MyJobManager` to execute tasks, where the tasks time out after 10 seconds, and the function will stop if an error occurs while any of the tasks are executing.

```
y = dfeval(@rand,{2 2 2 2 2},{4 4 4 4 4}, ...
'JobManager','MyJobManager','Timeout',10,'StopOnError',true);
```

Evaluate the user function `myFun` using the cluster as defined in the configuration `myConfig`.

```
y = dfeval(@myFun, {task1args, task2args, task3args}, ...
    'configuration', 'myConfig', ...
    'FileDependencies', {'myFun.m'});
```

See Also

`dfevalasync` | `feval` | `findResource`

dfevalasync

Purpose

Evaluate function asynchronously using cluster

Syntax

```
Job = dfevalasync(F, numArgOut, x1,...,xn, 'P1',V1,'P2',V2,
   ...)
Job = dfevalasync(F, numArgOut, x1,...,xn,
   ... 'configuration', 'ConfigurationName',...)
```

Arguments

`Job`	Job object created to evaluation the function.
`F`	Function name, function handle, or cell array of function names or handles.
`numArgOut`	Number of output arguments from each task's execution of function `F`.
`x1, ..., xn`	Cell arrays of input arguments to the functions.
`'P1', V1, 'P2', V2,...`	Property name/property value pairs for the created job object; can be name/value pairs or structures.

Description

`Job = dfevalasync(F, numArgOut, x1,...,xn, 'P1',V1,'P2',V2,...)` is equivalent to `dfeval`, except that it runs asynchronously (async), returning to the prompt immediately with a single output argument containing the job object that it has created and sent to the cluster. You have immediate access to the job object before the job is completed. You can use `waitForState` to determine when the job is completed, and `getAllOutputArguments` to retrieve your results.

`Job = dfevalasync(F, numArgOut, x1,...,xn, ... 'configuration', 'ConfigurationName',...)` evaluates the function `F` in a cluster by using all the properties defined in the configuration `ConfigurationName`. The configuration settings are used to find and initialize a scheduler, create a job, and create tasks. For details about defining and applying configurations, see "Cluster Profiles" on page

6-12. Configurations enable you to use `dfevalasync` with any type of scheduler.

For further discussion on the usage of `dfevalasync`, see "Evaluate Functions Asynchronously" on page 7-8.

Examples

Execute a `sum` function distributed in three tasks.

```
job = dfevalasync(@sum,1,{[1,2],[3,4],[5,6]}, ...
        'jobmanager','MyJobManager');
```

When the job is finished, you can obtain the results associated with the job.

```
waitForState(job);
data = getAllOutputArguments(job)
data =
    [ 3]
    [ 7]
    [11]
```

`data` is an `M`-by-`numArgOut` cell array, where `M` is the number of tasks.

See Also

`dfeval` | `feval` | `getAllOutputArguments` | `waitForState`

diary

Purpose

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.

See Also

`diary` | `batch` | `load`

Purpose

Create distributed array from data in client workspace

Syntax

```
D = distributed(X)
```

Description

`D = distributed(X)` creates a distributed array from `X`. `X` is an array stored on the MATLAB client, and `D` is a distributed array stored in parts on the workers of the open MATLAB pool.

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 static constructor methods such as `distributed.ones`, `distributed.zeros`, etc.

If the input argument is already a distributed array, the result is the same as the input.

Examples

Create a small array and distribute it:

```
Nsmall = 50;
D1 = distributed(magic(Nsmall));
```

Create a large distributed array using a static build method:

```
Nlarge = 1000;
D2 = distributed.rand(Nlarge);
```

distributed.cell

Purpose

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`

Purpose

Create distributed identity matrix

Syntax

```
D = distributed.eye(n)
D = distributed.eye(m, n)
D = distributed.eye([m, n])
D = distributed.eye(..., classname)
```

Description

`D = distributed.eye(n)` creates an `n`-by-`n` distributed identity matrix of underlying class double.

`D = distributed.eye(m, n)` or `D = distributed.eye([m, n])` creates an `m`-by-`n` distributed matrix of underlying class double with 1's on the diagonal and 0's elsewhere.

`D = distributed.eye(..., *classname*)` specifies the class of the distributed array `D`. Valid choices are the same as for the regular `eye` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.

Examples

Create a 1000-by-1000 distributed identity matrix of class double:

```
D = distributed.eye(1000)
```

See Also

`eye` | `codistributed.eye` | `distributed.ones` | `distributed.speye` | `distributed.zeros`

distributed.false

Purpose

Create distributed false array

Syntax

```
F = distributed.false(n)
F = distributed.false(m, n, ...)
F = distributed.false([m, n, ...])
```

Description

`F = distributed.false(n)` creates an `n`-by-`n` distributed array of logical zeros.

`F = distributed.false(m, n, ...)` or `F = distributed.false([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of logical zeros.

Examples

Create a 1000-by-1000 distributed false array.

```
F = distributed.false(1000);
```

See Also

`false` | `codistributed.false` | `distributed.true`

distributed.Inf

Purpose

Create distributed array of `Inf` values

Syntax

```
D = distributed.Inf(n)
D = distributed.Inf(m, n, ...)
D = distributed.Inf([m, n, ...])
D = distributed.Inf(..., classname)
```

Description

`D = distributed.Inf(n)` creates an `n`-by-`n` distributed matrix of `Inf` values.

`D = distributed.Inf(m, n, ...)` or `D = distributed.Inf([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of `Inf` values.

`D = distributed.Inf(..., *classname*)` specifies the class of the distributed array `D`. Valid choices are the same as for the regular `Inf` function: `'double'` (the default), or `'single'`.

Examples

Create a 1000-by-1000 distributed matrix of `Inf` values:

```
D = distributed.Inf(1000)
```

See Also

`Inf` | `codistributed.Inf` | `distributed.NaN`

distributed.NaN

Purpose

Create distributed array of Not-a-Number values

Syntax

```
D = distributed.NaN(n)
D = distributed.NaN(m, n, ...)
D = distributed.NaN([m, n, ...])
D = distributed.NaN(..., classname)
```

Description

`D = distributed.NaN(n)` creates an `n`-by-`n` distributed matrix of `NaN` values.

`D = distributed.NaN(m, n, ...)` or `D = distributed.NaN([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of `NaN` values.

`D = distributed.NaN(..., *classname*)` specifies the class of the distributed array `D`. Valid choices are the same as for the regular `NaN` function: `'double'` (the default), or `'single'`.

Examples

Create a 1000-by-1000 distributed matrix of `NaN` values of class double:

```
D = distributed.NaN(1000)
```

See Also

`Inf` | `codistributed.NaN` | `distributed.Inf`

Purpose

Create distributed array of ones

Syntax

```
D = distributed.ones(n)
D = distributed.ones(m, n, ...)
D = distributed.ones([m, n, ...])
D = distributed.ones(..., classname)
```

Description

`D = distributed.ones(n)` creates an `n`-by-`n` distributed matrix of ones of class double.

`D = distributed.ones(m, n, ...)` or `D = distributed.ones([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of ones.

`D = distributed.ones(..., *classname*)` specifies the class of the distributed array `D`. Valid choices are the same as for the regular `ones` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.

Examples

Create a 1000-by-1000 distributed matrix of ones of class double:

```
D = distributed.ones(1000);
```

See Also

`ones` | `codistributed.ones` | `distributed.eye` | `distributed.zeros`

distributed.rand

Purpose

Create distributed array of uniformly distributed pseudo-random numbers

Syntax

```
R = distributed.rand(n)
R = distributed.rand(m, n, ...)
R = distributed.rand([m, n, ...])
R = distributed.rand(..., classname)
```

Description

`R = distributed.rand(n)` creates an `n`-by-`n` distributed array of underlying class double.

`R = distributed.rand(m, n, ...)` or `R = distributed.rand([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of underlying class double.

`R = distributed.rand(..., classname)` specifies the class of the distributed array `R`. Valid choices are the same as for the regular `rand` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.

Tips

When you use `rand` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

Create a 1000-by-1000 distributed matrix of random values of class double:

```
R = distributed.rand(1000);
```

See Also

`rand` | `codistributed.rand` | `distributed.randn` | `distributed.sprand` | `distributed.sprandn`

Purpose

Create distributed array of normally distributed random values

Syntax

```
RN = distributed.randn(n)
RN = distributed.randn(m, n, ...)
RN = distributed.randn([m, n, ...])
RN = distributed.randn(..., classname)
```

Description

`RN = distributed.randn(n)` creates an `n`-by-`n` distributed array of normally distributed random values with underlying class double.

`RN = distributed.randn(m, n, ...)` and `RN = distributed.randn([m, n, ...])` create an `m`-by-`n`-by-... distributed array of normally distributed random values.

`RN = distributed.randn(..., *classname*)` specifies the class of the distributed array `D`. Valid choices are the same as for the regular `randn` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.

Tips

When you use `randn` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

Create a 1000-by-1000 distributed matrix of normally distributed random values of class double:

```
RN = distributed.randn(1000);
```

See Also

`randn` | `codistributed.randn` | `distributed.rand` | `distributed.speye` | `distributed.sprand` | `distributed.sprandn`

distributed.spalloc

Purpose 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 `spalloc` | `codistributed.spalloc` | `sparse`

Purpose

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

`speye` | `codistributed.speye` | `distributed.eye`

distributed.sprand

Purpose

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.

Tips

When you use `sprand` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

Create a 1000-by-1000 sparse distributed double array `DS` with approximately 1000 nonzeros.

```
DS = distributed.sprand(1000, 1000, .001);
```

See Also

`sprand | codistributed.sprand | distributed.rand | distributed.randn | sparse | distributed.speye | distributed.sprandn`

Purpose

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.

Tips

When you use `sprandn` on the workers in the MATLAB pool, or in a distributed or parallel job (including pmode), each worker or lab sets its random generator seed to a value that depends only on the lab index or task ID. Therefore, the array on each lab is unique for that job. However, if you repeat the job, you get the same random data.

Examples

Create a 1000-by-1000 sparse distributed double array `DS` with approximately 1000 nonzeros.

```
DS = distributed.sprandn(1000, 1000, .001);
```

See Also

`sprandn` | `codistributed.sprandn` | `distributed.rand` | `distributed.randn` | `sparse` | `distributed.speye` | `distributed.sprand`

distributed.true

Purpose

Create distributed true array

Syntax

```
T = distributed.true(n)
T = distributed.true(m, n, ...)
T = distributed.true([m, n, ...])
```

Description

`T = distributed.true(n)` creates an `n`-by-`n` distributed array of logical ones.

`T = distributed.true(m, n, ...)` or `T = distributed.true([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of logical ones.

Examples

Create a 1000-by-1000 distributed true array.

```
T = distributed.true(1000);
```

See Also

`true` | `codistributed.true` | `distributed.false`

Purpose

Create distributed array of zeros

Syntax

```
D = distributed.zeros(n)
D = distributed.zeros(m, n, ...)
D = distributed.zeros([m, n, ...])
D = distributed.zeros(..., classname)
```

Description

`D = distributed.zeros(n)` creates an `n`-by-`n` distributed matrix of zeros of class double.

`D = distributed.zeros(m, n, ...)` or `D = distributed.zeros([m, n, ...])` creates an `m`-by-`n`-by-... distributed array of zeros.

`D = distributed.zeros(..., *classname*)` specifies the class of the distributed array `D`. Valid choices are the same as for the regular `zeros` function: `'double'` (the default), `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, and `'uint64'`.

Examples

Create a 1000-by-1000 distributed matrix of zeros using default class:

```
D = distributed.zeros(1000);
```

See Also

`zeros` | `codistributed.zeros` | `distributed.eye` | `distributed.ones`

dload

Purpose

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 MATLAB 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 MATLAB pool workers using the default distribution scheme. It is not necessary to have the same size MATLAB pool open when loading as when saving using `dsave`.

When loading Composite objects, the data is sent to the available MATLAB pool workers. If the Composite is too large to fit on the current

MATLAB pool, the data is not loaded. If the Composite is smaller than the current MATLAB 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

load | Composite | distributed | dsave | matlabpool

Purpose

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 MATLAB pool open, create and save several variables to `mydatafile.mat`:

```
D = distributed.rand(1000); % Distributed array
C = Composite();            %
C{1} = magic(20);           % Data on lab 1 only
X = rand(40);               % Client workspace only
dsave mydatafile D C X      % Save all three variables
```

See Also

`save` | `Composite` | `distributed` | `dload` | `matlabpool`

Purpose

Check whether Composite is defined on labs

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 lab 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 labs are serializable. The function throws an error if the lab indices are invalid.

Examples

Define a variable on a random number of labs. Check on which labs 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`

existsOnGPU

Purpose

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 the index of the current 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

gpuDevice | gpuArray | parallel.gpu.CUDAKernel | reset

fetchOutputs

Purpose

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}
```

Purpose

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`

Purpose

Find job objects stored in scheduler

Syntax

```
out = findJob(sched)
[pending queued running completed] = findJob(sched)
out = findJob(sched,'p1',v1,'p2',v2,...)
```

Arguments

`sched`	Scheduler object in which to find the job.
`pending`	Array of jobs whose `State` is `pending` in scheduler `sched`.
`queued`	Array of jobs whose `State` is `queued` in scheduler `sched`.
`running`	Array of jobs whose `State` is `running` in scheduler `sched`.
`completed`	Array of jobs that have completed running, i.e., whose `State` is `finished` or `failed` in scheduler `sched`.
`out`	Array of jobs found in scheduler `sched`.
`p1`, *`p2`*	Job object properties to match.
`v1`, `v2`	Values for corresponding object properties.

Description

`out = findJob(sched)` returns an array, `out`, of all job objects stored in the scheduler `sched`. 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(sched)` returns arrays of all job objects stored in the scheduler `sched`, 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 destroyed or whose status is unavailable are not returned by this function.

`out = findJob(sched,'p1',v1,'p2',v2,...)` returns an array, `out`, of job objects whose property names and property values match those passed as parameter-value pairs, *`p1`*, `v1`, *`p2`*, `v2`.

Note that the property value pairs can be in any format supported by the `set` function, i.e., param-value string pairs, structures, and param-value cell array pairs. If a structure is used, the structure field names are job object property names and the field values are the appropriate property values to match.

When a property value is specified, it must use the same exact value that the `get` function returns, including letter case. For example, if `get` returns the `Name` property value as `MyJob`, then `findJob` will not find that object while searching for a `Name` property value of `myjob`.

See Also

`createJob` | `findResource` | `findTask` | `parcluster` | `submit`

Purpose

Find available parallel computing resources

Syntax

```
out = findResource()
out = findResource('scheduler', ... 'configuration',
   'ConfigurationName')
out = findResource('scheduler', 'type', 'SchedType')
out = findResource('scheduler', 'type', 'jobmanager',
   'LookupURL', 'host:port')
out = findResource('scheduler', 'type', 'hpcserver')
out = findResource('scheduler', 'type', 'hpcserver',
   'SchedulerHostname', 'headNode')
out = findResource('worker')
out = findResource('worker', 'LookupURL', 'host:port')
out = findResource(... ,'p1', v1, 'p2', v2,...)
```

Arguments

`out`	Object or array of objects returned.
`'configuration'`	Literal string to indicate usage of a configuration.
`'ConfigurationName'`	Name of configuration to use.
`'scheduler'`	Literal string specifying that you are finding a scheduler, which can be a job manager or a third-party scheduler.
`'SchedType'`	Specifies the type of scheduler: `'jobmanager'`, `'local'`, `'hpcserver'`, `'LSF'`, `'pbspro'`, `'torque'`, `'mpiexec'`, or any string that starts with `'generic'`.
`'worker'`	Literal string specifying that you are finding a worker.
`'LookupURL'`	Literal string to indicate usage of a remote lookup service.
`'host:port'`	Host name and (optionally) port of remote lookup service to use.

`'SchedulerHostname'`	Literal string to indicate that the next argument identifies the head node for the HPC Server scheduler. Ignored if scheduler type is not `'hpcserver'`.
`'headNode'`	String specifying the HPC Server head node name.
`p1`, *`p2`*	Object properties to match.
`v1`, `v2`	Values for corresponding object properties.

Description

`out = findResource()` returns a scheduler object , `out`, representing the scheduler identified by the default parallel configuration, with the scheduler object properties set to the values defined in that configuration.

`out = findResource('`**`scheduler`**`', ...  '`**`configuration`**`', 'ConfigurationName')` returns a scheduler object , `out`, representing the scheduler identified by the parallel configuration `ConfigurationName`, with the scheduler object properties set to the values defined in that configuration. For details about defining and applying parallel configurations, see "Cluster Profiles" on page 6-12.

Note If you specify the **`'scheduler'`** option without the **`'configuration'`** option, no configuration is used, so no configuration properties are applied to the object.

`out = findResource('`**`scheduler`**`', '`**`type`**`', '`*`SchedType`*`')` returns an array, `out`, containing objects representing all available parallel computing schedulers of the given type. *`SchedType`* can be `'jobmanager'`, `'local'`, `'hpcserver'`, `'LSF'`, `'pbspro'`, `'torque'`, `'mpiexec'`, or any string starting with `'generic'`. A `'local'` scheduler queues jobs for running on workers that it will start on your local client machine. You can use different scheduler types starting with `'generic'` to identify one generic scheduler or configuration from another. You can have multiple scheduler objects to simultaneously support several job managers or generic schedulers, but you cannot create more than one object for each type of fully supported third-party scheduler or the local scheduler. For third-party and generic schedulers, job data is stored in the location specified by the scheduler object's `DataLocation` property.

`out = findResource('`**`scheduler`**`', '`**`type`**`', '`**`jobmanager`**`', '`**`LookupURL`**`', 'host:port')` uses the lookup process of the job manager running at a specific location. The lookup process is part of a job manager. By default, `findResource` uses all the lookup processes that are available to the local machine via multicast. If you specify `'`**`LookupURL`**`'` with a host, `findResource` uses the job manager lookup process running at that location. The port is optional, and is necessary only if the lookup process was configured to use a port other than the default `BASEPORT` setting of the `mdce_def` file. This URL is where the lookup is performed from, it is not necessarily the host running the job manager or worker. This unicast call is useful when you want to find resources that might not be available via multicast or in a network that does not support multicast.

Notes Although Version 5 of the Parallel Computing Toolbox and MATLAB Distributed Computing Server products continue to support multicast communications between their processes, multicast is not recommended and might not be supported in future releases.

`findResource` ignores **`LookupURL`** when finding third-party schedulers.

`out = findResource('`**`scheduler`**`', '`**`type`**`', '`**`hpcserver`**`')` searches your environment variables and Active Directory to find the HPC Server cluster head node. If more than one HPC Server cluster head node is found in the Active Directory, an error is thrown listing the names of all the head nodes found.

`out = findResource('`**`scheduler`**`', '`**`type`**`', '`**`hpcserver`**`', '`**`SchedulerHostname`**`', 'headNode')` uses the HPC Server scheduler with the specified head node. **`SchedulerHostname`** is ignored for all scheduler types other than `'hpcserver'`.

`out = findResource('`**`worker`**`')` or `out = findResource('`**`worker`**`', '`**`LookupURL`**`', 'host:port')` returns an array of worker objects representing all found workers or those that match the search criteria.

`out = findResource(...  ,'`*`p1`*`', v1, '`*`p2`*`', v2,...)` returns an array, `out`, of resources whose property names and property values match those passed as parameter-value pairs, *`p1`*, `v1`, *`p2`*, `v2`.

Note that the property value pairs can be in any format supported by the `set` function.

When a property value is specified, it must use the same exact value that the `get` function returns, including letter case. For example, if `get` returns the `Name` property value as `'MyJobManager'`, then `findResource` will *not* find that object if searching for a `Name` property value of `'myjobmanager'`.

Tips

You are allowed to use parameter-value string pairs, structures, parameter-value cell array pairs, and configurations in the same call to `findResource`.

Examples

Find a particular job manager by its name.

```
jm1 = findResource('scheduler','type','jobmanager', ...
        'Name', 'ClusterQueue1');
```

Find all job managers. In this example, there are four.

```
all_job_managers = findResource('scheduler','type','jobmanager')
all_job_managers =
    distcomp.jobmanager: 1-by-4
```

Find all job managers accessible from the lookup service on a particular host.

```
jms = findResource('scheduler','type','jobmanager', ...
        'LookupURL','host234');
```

Find a particular job manager accessible from the lookup service on a particular host. In this example, `subnet2.hostalpha` port `6789` is where the lookup is performed, but the job manager named `SN2Jmgr` might be running on another machine.

```
jm = findResource('scheduler','type','jobmanager', ...
       'LookupURL', 'subnet2.hostalpha:6789', 'Name', 'SN2JMgr');
```

Find the Platform LSF scheduler on the network.

```
lsf_sched = findResource('scheduler','type','LSF')
```

Create a local scheduler that will start workers on the client machine for running your job.

```
local_sched = findResource('scheduler','type','local')
```

Find the scheduler identified by the default parallel configuration, with the scheduler object properties set to the values defined in that configuration.

```
sched = findResource();
```

See Also

`findJob` | `findTask`

findTask

Purpose

Task objects belonging to job object

Syntax

```
tasks = findTask(obj)
[pending running completed] = findTask(obj)
tasks = findTask(obj,'p1',v1,'p2',v2,...)
```

Arguments

`obj`	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(obj)` gets a 1-by-N array of task objects belonging to a job object `obj` Tasks in the array are ordered by the `ID` property of the tasks, indicating the sequence in which they were created.

`[pending running completed] = findTask(obj)` returns arrays of all task objects stored in the job object `obj`, sorted by state. Within each array (`pending`, `running`, and `completed`), the tasks are returned in sequence of creation.

`tasks = findTask(obj,'p1',v1,'p2',v2,...)` gets a 1-by-N array of task objects belonging to a job object `obj`. The returned task objects will be only those having the specified property-value pairs.

Note that the property value pairs can be in any format supported by the `set` function, i.e., param-value string pairs, structures, and param-value cell array pairs. If a structure is used, the structure field names are object property names and the field values are the appropriate property values to match.

When a property value is specified, it must use the same exact value that the `get` function returns, including letter case. For example, if `get` returns the `Name` property value as `MyTask`, then `findTask` will not find that object while searching for a `Name` property value of `mytask`.

Tips

If `obj` 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.

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)
```

See Also

`createJob` | `createTask` | `findJob`

Purpose

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 lab.

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 labs 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 lab, 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 lab 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`

Purpose

Transfer distributed array data or GPUArray to local workspace

Syntax

```
X = gather(A)
X = gather(C, lab)
```

Description

`X = gather(A)` can operate inside an `spmd` statement, pmode, or parallel job to gather together the data of a codistributed array, or outside an `spmd` statement to gather the data of a distributed array. If you execute this inside an `spmd` statement, pmode, or parallel job, `X` is replicated array with all the data of the array on every lab. If you execute this outside an `spmd` statement, `X` is an array in the local workspace, with the data transferred from the multiple labs.

`X = gather(distributed(X))` or `X = gather(codistributed(X))` returns the original array `X`.

`X = gather(C, lab)` converts a codistributed array `C` to a variant array `X`, such that all of the data is contained on lab `lab`, and `X` is a 0-by-0 empty double on all other labs.

For a GPUArray input, `X = gather(A)` transfers the data from the GPU to the local workspace.

If the input argument to `gather` is not a distributed, a codistributed, or a GPUArray, the output is the same as the input.

Tips

Note that `gather` assembles the codistributed or distributed array in the workspaces of all the labs 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 “Accessing Data with Composites” on page 3-7. If you are running `gather` in a parallel 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 labs, you cannot gather data from all the labs onto a single lab by placing the function inside a conditional statement such as `if labindex == 1`.

Examples

Distribute a magic square across your labs, then gather the whole matrix onto every lab and then onto the client. This code results in the equivalent of `M = magic(n)` on all labs and the client.

```
n = 10;
spmd
  C = codistributed(magic(n));
  M = gather(C) % Gather data on all labs
end
S = gather(C) % Gather data on client
```

Gather all of the data in `C` onto lab 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
```

Gather all of the data from a distributed array into `D` on the client.

```
n = 10;
D = distributed(magic(n)); % Distribute data to labs
M = gather(D) % Return data to client
```

Gather the results of a GPU operation to the local workspace.

```
G = gpuArray(rand(1024,1));
F = sqrt(G); %input and output both GPUArray
W = gather(G); % Return data to client
whos
Name          Size            Bytes  Class

F         1024x1                108  parallel.gpu.GPUArray
G         1024x1                108  parallel.gpu.GPUArray
W         1024x1               8192  double
```

See Also

arrayfun | codistributed | distributed | gpuArray | pmode

Purpose

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 lab in the second dimension. The result is replicated on all labs.

`Xs = gcat(X, dim)` concatenates the variant array `X` from each lab in the dimension indicated by `dim`.

`Xs = gcat(X, dim, targetlab)` performs the reduction, and places the result into `res` only on the lab indicated by `targetlab`. `res` is set to `[]` on all other labs.

Examples

With four labs,

```
Xs = gcat(labindex)
```

returns `Xs = [1 2 3 4]` on all four labs.

See Also

`cat` | `gop` | `labindex` | `numlabs`

Purpose

Object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Arguments

`obj`	An object or an array of objects.
`'PropertyName'`	A property name or a cell array of property names.
`out`	A single property value, a structure of property values, or a cell array of property values.

Description

`get(obj)` returns all property names and their current values to the command line for `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-`n` or `n`-by-1 cell array of strings containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of objects, then `out` will be an `m`-by-`n` cell array of property values where `m` is equal to the length of `obj` and `n` is equal to the number of properties specified.

Tips

When specifying a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `jm` is a job manager object, then these commands are all valid and return the same result.

```
out = get(jm,'HostAddress');
out = get(jm,'hostaddress');
out = get(jm,'HostAddr');
```

Examples

This example illustrates some of the ways you can use `get` to return property values for the job object `j1`.

```
get(j1,'State')
ans =
pending

get(j1,'Name')
ans =
MyJobManager_job

out = get(j1);
out.State
ans =
pending

out.Name
ans =
MyJobManager_job

two_props = {'State' 'Name'};
get(j1, two_props)
ans =
    'pending'    'MyJobManager_job'
```

See Also

`inspect` | `set`

Purpose

Output arguments from evaluation of all tasks in job object

Syntax

```
data = getAllOutputArguments(obj)
```

Arguments

`obj`	Job object whose tasks generate output arguments.
`data`	`M`-by-`N` cell array of job results.

Description

`data = getAllOutputArguments(obj)` returns `data`, the output data 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` columns, 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` will be the same as the order of the tasks contained in the job.

Tips

If you are using a job manager, `getAllOutputArguments` results in a call to a remote service, which could take a long time to complete, depending on the amount of data being retrieved and the network speed. Also, if the remote service is no longer available, an error will be thrown.

Note that issuing a call to `getAllOutputArguments` will not remove the output data from the location where it is stored. To remove the output data, use the `destroy` function to remove the individual task or their parent job object.

The same information returned by `getAllOutputArguments` can be obtained by accessing the `OutputArguments` property of each task in the job.

Examples

Create a job to generate a random matrix.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
```

```
j = createJob(jm, 'Name', 'myjob');
t = createTask(j, @rand, 1, {10});
submit(j);
data = getAllOutputArguments(j);
```

Display the 10-by-10 random matrix.

```
disp(data{1});
destroy(j);
```

See Also

`submit`

getAttachedFilesFolder

Purpose

Folder into which AttachedFiles are written

Syntax

```
folder = getAttachedFilesFolder
```

Arguments

`folder`	String indicating location where files from job's `AttachedFiles` property are placed

Description

`folder = getAttachedFilesFolder` returns a string, which is the path to the local folder into which `AttachedFiles` are written. This function returns an empty array if it is not called on a MATLAB worker.

Examples

Find the current `AttachedFiles` folder.

```
folder = getAttachedFilesFolder;
```

Change to that folder to invoke an executable that was included in `AttachedFiles`.

```
oldFolder = cd(folder);
```

Invoke the executable.

```
[OK, output] = system('myexecutable');
```

Change back to the original folder.

```
cd(oldfolder);
```

See Also

`getCurrentCluster` | `getCurrentJob` | `getCurrentTask` | `getCurrentWorker`

Purpose

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 labs. 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 labs:

```
spmd (4)
    I1 = codistributed.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 labs:

```
spmd (4)
    I2 = codistributed.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`

Purpose

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 cluster object that has sent the task currently being evaluated by the worker session. `c` is the `Parent` of the task's parent job.

Tips

If this function is executed in a MATLAB session that is not a worker, you get an empty result.

Examples

Find the current cluster.

```
myCluster = getCurrentCluster;
```

Get the host on which the cluster is running.

```
host = myCluster.Host;
```

See Also

`getAttachedFilesFolder` | `getCurrentJob` | `getCurrentTask` | `getCurrentWorker`

getCurrentJob

Purpose 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 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 `getCurrentJobmanager` | `getCurrentTask` | `getCurrentWorker` | `getFileDependencyDir`

Purpose

Job manager object that scheduled current task

Syntax

```
jm = getCurrentJobmanager
```

Arguments

`jm`	The job manager object that scheduled the task currently being evaluated by the worker session.

Description

`jm = getCurrentJobmanager` returns the job manager object that has sent the task currently being evaluated by the worker session. `jm` is the `Parent` of the task's parent job.

Tips

If the function is executed in a MATLAB session that is not a worker, you get an empty result.

If your tasks are scheduled by a third-party scheduler instead of a job manager, `getCurrentJobmanager` returns a `distcomp.taskrunner` object.

See Also

`getCurrentJob` | `getCurrentTask` | `getCurrentWorker` | `getFileDependencyDir`

Purpose

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 task object that is 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

`getCurrentJob` | `getCurrentJobmanager` | `getCurrentWorker` | `getFileDependencyDir`

Purpose

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 worker object representing the session that is currently evaluating the task that calls this function.

Tips

If the function runs in a MATLAB session that is not a worker, it returns an empty result.

Examples

Create a job with one task, and have the task return the worker that evaluates it. Then view the `Host` property of the worker:

```
c = parcluster();
j = createJob(c);
t = createTask(j, @getCurrentWorker, 1, {});
submit(j)
wait(j)
w = t.OutputArguments{1};
h = w.Host
```

The task `t` executes `getCurrentWorker` to get an object representing the worker that is evaluating the task. The result is placed in the `OutputArguments` property of the task.

Create a task to return only the `Host` property value of its worker:

```
c = parcluster();
j = createJob(c);
t = createTask(j, @() get(getCurrentWorker,'Host'), 1, {});
submit(j)
wait(j)
h = t.OutputArguments{1}
```

This code defines a task to run an anonymous function, which uses `get` to view the `Host` property of the worker object returned by `getCurrentWorker`. So only the `Host` property value is available in the `OutputArguments` property.

See Also

`getCurrentJob` | `getCurrentJobmanager` | `getCurrentTask` | `getFileDependencyDir`

Purpose

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 string expression.
`cluster`	Cluster object referring to mpiexec, Microsoft Windows HPC Server (or CCS), Platform LSF, PBS Pro, or TORQUE scheduler, created by `findResource`.
`job_or_task`	Object identifying job, parallel 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`. You cannot use this function to retrieve messages from a task in an mpiexec cluster.

Examples

Construct a cluster object so you can create a communicating job. Assume that you have already defined a profile called `mpiexec` to define the properties of the cluster.

```
mpiexecObj = parcluster('mpiexec');
```

Create and submit a parallel job.

```
job = createCommunicatingJob(mpiexecObj);
createTask(job, @labindex, 1, {});
submit(job);
```

Look at the debug log.

```
getDebugLog(mpiexecObj, job);
```

See Also createCommunicatingJob | createJob | createTask | parcluster

Purpose

Directory where `FileDependencies` are written on worker machine

Syntax

```
depdir = getFileDependencyDir
```

Arguments

`depdir`	String indicating directory where `FileDependencies` are placed.

Description

`depdir = getFileDependencyDir` returns a string, which is the path to the local directory into which `FileDependencies` are written. This function will return an empty array if it is not called on a MATLAB worker.

Examples

Find the current directory for `FileDependencies`.

```
ddir = getFileDependencyDir;
```

Change to that directory to invoke an executable.

```
cdir = cd(ddir);
```

Invoke the executable.

```
[OK, output] = system('myexecutable');
```

Change back to the original directory.

```
cd(cdir);
```

See Also

`getCurrentJob` | `getCurrentJobmanager` | `getCurrentTask` | `getCurrentWorker` | `FileDependencies`

getJobClusterData

Purpose

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.

To use this feature, you should call the function `setJobClusterData` in the submit function (identified by the `IndependentSubmitFcn` or `CommunicatingSubmitFcn` property) and call `getJobSchedulerData` in any of the functions identified by the properties `GetJobStateFcn`, `DeleteJobFcn`, `DeleteTaskFcn`, `CancelJobFcn`, or `CancelTaskFcn`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

See Also

`setJobClusterData`

Purpose

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`

getJobFolderOnCluster

Purpose 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`

Purpose

Get specific user data for job on generic scheduler

Syntax

```
userdata = getJobSchedulerData(sched, job)
```

Arguments

`userdata`	Information that was previously stored for this job.
`sched`	Scheduler object identifying the generic third-party scheduler running the job.
`job`	Job object identifying the job for which to retrieve data.

Description

`userdata = getJobSchedulerData(sched, job)` returns data stored for the job `job` that was derived from the generic scheduler `sched`. The information was originally stored with the function `setJobSchedulerData`. 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.

To use this feature, you should call the function `setJobSchedulerData` in the submit function (identified by the `SubmitFcn` property) and call `getJobSchedulerData` in any of the functions identified by the properties `GetJobStateFcn`, `DestroyJobFcn`, `DestroyTaskFcn`, `CancelJobFcn`, or `CancelTaskFcn`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

See Also

`setJobSchedulerData`

Purpose

Local portion of codistributed array

Syntax

```
L = getLocalPart(A)
```

Description

`L = getLocalPart(A)` returns the local portion of a codistributed array.

Examples

With four labs,

```
A = magic(4);   %replicated on all labs
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`

Purpose

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`

globalIndices

Purpose

Global indices for local part of codistributed array

Syntax

```
K = globalIndices(R, dim)
K = globalIndices(R, dim, lab)
[E,F] = globalIndices(R, dim)
[E,F] = globalIndices(R, dim, lab)
K = codist.globalIndices(dim, lab)
[E,F] = codist.globalIndices(dim, lab)
```

Description

`globalIndices` tell you the relationship between indices on a local part and the corresponding index range in a given dimension on the distributed array. The `globalIndices` method on a codistributor object allows you to get this relationship without actually creating the array.

`K = globalIndices(R, dim)` or `K = globalIndices(R, dim, lab)` returns a vector `K` so that `getLocalPart(R) = R(...,K,...)` in the specified dimension `dim` on the specified lab. If the lab argument is omitted, the default is `labindex`.

`[E,F] = globalIndices(R, dim)` or `[E,F] = globalIndices(R, dim, lab)` returns two integers `E` and `F` so that `getLocalPart(R) = R(...,E:F,...)` in the specified dimension `dim` on the specified lab. If the lab argument is omitted, the default is `labindex`.

`K = codist.globalIndices(dim, lab)` is the same as `K = globalIndices(R, dim, lab)`, where `codist` is the codistributor for `R`, or `codist = getCodistributor(R)`. This allows you to get the global indices for a codistributed array without having to create the array itself.

`[E,F] = codist.globalIndices(dim, lab)` is the same as `[E,F] = globalIndices(R, dim, lab)`, where `codist` is the codistributor for `R`, or `codist = getCodistributor(R)`. 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 labs, and view the global indices on each lab:

```
spmd
    C = codistributed.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 labs and alleviates you from calculating offsets or partitions.

```
spmd
    siz = [1000, 1000];
    codistr = codistributor1d(2, [], siz);

    % Use globalIndices to figure out which columns
    % each lab should load.
    [firstCol, lastCol] = codistr.globalIndices(2);

    % Call user-defined function readRectangleFromFile to
    % load all the values that should go into
    % the local part for this lab.
    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`

Purpose

Global operation across all labs

Syntax

```
res = gop(@F, x)
res = gop(@F, x, targetlab)
```

Arguments

`F`	Function to operate across labs.
`x`	Argument to function `F`, should be same variable on all labs, but can have different values.
`res`	Variable to hold reduction result.
`targetlab`	Lab to which reduction results are returned.

Description

`res = gop(@F, x)` is the reduction via the function `F` of the quantities `x` from each lab. The result is duplicated on all labs.

The function `F(x,y)` should accept two arguments of the same type and produce one result of that type, so it can be used iteratively, that is,

```
  F(F(x1,x2),F(x3,x4))
```

The function `F` should be associative, that is,

```
F(F(x1, x2), x3) = F(x1, F(x2, x3))
```

`res = gop(@F, x, targetlab)` performs the reduction, and places the result into `res` only on the lab indicated by `targetlab`. `res` is set to `[]` on all other labs.

Examples

Calculate the sum of all labs' value for `x`.

```
res = gop(@plus,x)
```

Find the maximum value of `x` among all the labs.

```
res = gop(@max,x)
```

Perform the horizontal concatenation of x from all labs.

```
res = gop(@horzcat,x)
```

Calculate the 2-norm of x from all labs.

```
res = gop(@(a1,a2)norm([a1 a2]),x)
```

See Also

labBarrier | numlabs

gplus

Purpose

Global addition

Syntax

```
S = gplus(X)
S = gplus(X, targetlab)
```

Description

`S = gplus(X)` returns the addition of the variant array `X` from each lab. The result `S` is replicated on all labs.

`S = gplus(X, targetlab)` performs the addition, and places the result into `S` only on the lab indicated by `targetlab`. `S` is set to `[]` on all other labs.

Examples

With four labs,

```
S = gplus(labindex)
```

returns `S = 1 + 2 + 3 + 4 = 10` on all four labs.

See Also

`gop` | `labindex`

Purpose

Create array on GPU

Syntax

```
G = gpuArray(X)
```

Description

`G = gpuArray(X)` copies the numeric data `X` to the GPU, and returns a GPUArray object. You can operate on this data by passing it to the `feval` method of a CUDA kernel object, or by using one of the methods defined for GPUArray objects in "Using GPUArray" on page 10-4.

The MATLAB data `X` must be numeric (for example: `single`, `double`, `int8`, etc.) or logical, and the GPU device must have sufficient free memory to store the data. `X` must be a full matrix, not sparse.

If the input argument is already a GPUArray, the output is the same as the input.

Examples

Transfer a 10-by-10 matrix of random single-precision values to the GPU, then use the GPU to square each element.

```
X = rand(10, 'single');
G = gpuArray(X);
isequal(gather(G), X)  % Returns true
classUnderlying(G)     % Returns 'single'
G2 = G .* G            % Uses times method defined for
                       % GPUArray objects
```

See Also

`arrayfun` | `bsxfun` | `existsOnGPU` | `feval` | `gather` | `parallel.gpu.CUDAKernel` | `reset`

gpuDevice

Purpose

Query or select GPU device

Syntax

```
D = gpuDevice
D = gpuDevice()
D = gpuDevice(IDX)
gpuDevice([ ])
```

Description

`D = gpuDevice` or `D = gpuDevice()`, if no device is already selected, selects the default GPU device and returns an object representing that device. If a GPU device is already selected, this returns an object representing that device without clearing it.

`D = gpuDevice(IDX)` selects the GPU device specified by index `IDX`. `IDX` must be in the range of 1 to `gpuDeviceCount`. A warning or error might occur if the specified GPU device is not supported. This form of the command with a specified index resets the device and clears its memory (even if this device is already currently selected, equivalent to `reset`); so all workspace variables representing GPUArray or CUDAKernel data are now invalid, and you should clear them from the workspace or redefine them.

`gpuDevice([ ])`, with an empty argument (as opposed to no argument), deselects the GPU device and clears its memory of GPUArray and CUDAKernel data. This leaves no GPU device selected as the current device.

Examples

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
```

See Also

`arrayfun` | `feval` | `gpuDeviceCount` | `parallel.gpu.CUDAKernel` | `reset`

gpuDeviceCount

Purpose

Number of GPU devices present

Syntax

```
n = gpuDeviceCount
```

Description

`n = gpuDeviceCount` returns the number of GPU devices present 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`

Purpose

Help for toolbox functions in Command Window

Syntax

`help class/function`

Arguments

class	A Parallel Computing Toolbox object class: `distcomp.jobmanager`, `distcomp.job`, or `distcomp.task`.
function	A function for the specified class. To see what functions are available for a class, see the `methods` 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 from each of the Parallel Computing Toolbox object classes.

```
help distcomp.jobmanager/createJob
help distcomp.job/cancel
help distcomp.task/waitForState

class(j1)
ans =
distcomp.job
help distcomp.job/createTask
```

See Also

`methods`

importParallelConfig

Purpose

Import parallel configuration .mat file

Syntax

```
configname = importParallelConfig(filename)
```

Description

The `importParallelConfig` function allows you to import a configuration that was stored in a `.mat` file.

`configname = importParallelConfig(filename)` imports the configuration stored in the specified file and returns the name of the imported configuration as a string assigned to `configname`. If a configuration with the same name already exists in your MATLAB session, an extension is added to the name of the imported configuration. If `filename` has no extension, `.mat` is assumed. Each configuration `.mat` file contains only one configuration.

You can use the imported configuration with any functions that support configurations. `importParallelConfig` does not set the imported configuration as the default; you can set it as the default configuration with the `defaultParallelConfig` function.

To export a configuration, use the Configurations Manager, which you can open by selecting **Parallel > Manage Configurations**. Configurations exported from earlier versions of the product are upgraded during the import.

Configurations that you import with `importParallelConfig` are saved as a part of your MATLAB preferences, so these configurations are available in your subsequent MATLAB sessions without importing them again.

Examples

Import a configuration from the file `Config01.mat` and use it to open a pool of MATLAB workers:

```
conf_1 = importParallelConfig('Config01')
matlabpool('open', conf_1)
```

Import a configuration from the file `ConfigMaster.mat` and set it as the default parallel configuration:

```
def_config = importParallelConfig('ConfigMaster')
defaultParallelConfig(def_config)
```

See Also

`defaultParallelConfig`

Purpose

Open Property Inspector

Syntax

`inspect(obj)`

Arguments

`obj`	An object or an array of objects.

Description

`inspect(obj)` opens the Property Inspector and allows you to inspect and set properties for the object `obj`.

Tips

You can also open the Property Inspector via the Workspace browser by double-clicking an object.

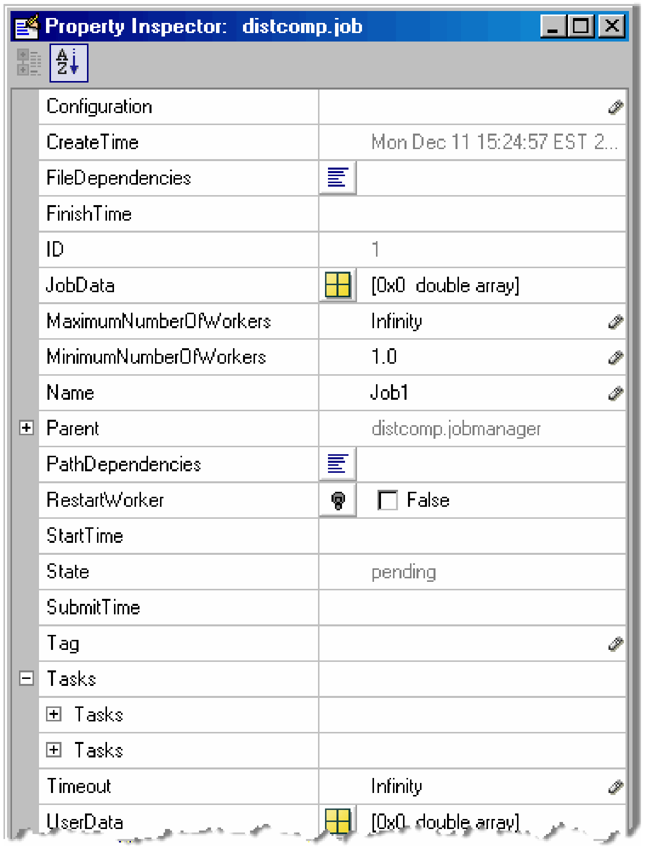
The Property Inspector does not automatically update its display. To refresh the Property Inspector, open it again.

Note that properties that are arrays of objects are expandable. In the figure of the example below, the `Tasks` property is expanded to enumerate the individual task objects that make up this property. These individual task objects can also be expanded to display their own properties.

Examples

Open the Property Inspector for the job object `j1`.

```
inspect(j1)
```


Property Inspector: distcomp.job

Configuration	
CreateTime	Mon Dec 11 15:24:57 EST 2...
FileDependencies	
FinishTime	
ID	1
JobData	[0x0 double array]
MaximumNumberOfWorkers	Infinity
MinimumNumberOfWorkers	1.0
Name	Job1
Parent	distcomp.jobmanager
PathDependencies	
RestartWorker	False
StartTime	
State	pending
SubmitTime	
Tag	
Tasks	
Tasks	
Tasks	
Timeout	Infinity
UserData	[0x0 double array]

See Also

`get` | `set`

Purpose

True if distributed array's underlying elements are of specified class

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  = distributed.ones(1, N, 'uint8');
D_cell   = distributed.cell(1, N);
isUint8  = isaUnderlying(D_uint8, 'uint8') % returns true
isDouble = isaUnderlying(D_cell, 'double') % returns false
```

See Also

`isa`

iscodistributed

Purpose

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 5-2.

Examples

With an open MATLAB pool,

```
spmd
    L = ones(100, 1);
    D = codistributed.ones(100, 1);
    iscodistributed(L) % returns false
    iscodistributed(D) % returns true
end
```

See Also

`isdistributed`

isComplete

Purpose

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 5-2.

See Also

`codistributed` | `codistributor`

Purpose

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 5-2.

Examples

With an open MATLAB pool,

```
L = ones(100, 1);
D = distributed.ones(100, 1);
isdistributed(L) % returns false
isdistributed(D) % returns true
```

See Also

`iscodistributed`

isequal

Purpose

True if clusters have same property values

Syntax

```
isequal(C1,C2)
isequal(C1,C2,C3,...)
```

Description

`isequal(C1,C2)` returns logical 1 () 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`

Purpose

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 5-2. `isreplicated` also returns true for a Composite `X` if all its elements are identical.

Tips

`isreplicated(X)` requires checking for equality of the array `X` across all labs. This might require extensive communication and time. `isreplicated` is most useful for debugging or error checking small arrays. A codistributed array is not replicated.

Examples

With an open MATLAB pool,

```
spmd
    A = magic(3);
    t = isreplicated(A) % returns t = true
    B = magic(labindex);
    f = isreplicated(B) % returns f = false
end
```

See Also

`iscodistributed` | `isdistributed`

Purpose

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 `FileDependencies` property.
2. In a folder included in the job's `PathDependencies` property.
3. In the worker's MATLAB installation at the location

   `matlabroot``/toolbox/distcomp/user/jobStartup.m`

To create a version of `jobStartup.m` for `FileDependencies` or `PathDependencies`, 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` | `FileDependencies` | `PathDependencies`

labBarrier

Purpose

Block execution until all labs reach this call

Syntax

```
labBarrier
```

Description

`labBarrier` blocks execution of a parallel algorithm until all labs have reached the call to `labBarrier`. This is useful for coordinating access to shared resources such as file I/O.

For a demonstration that uses `labSend`, `labReceive`, `labBarrier`, and `labSendReceive`, see the demo Profiling Explicit Parallel Communication.

Examples

In this example, all labs know the shared data filename.

```
fname = 'c:\data\datafile.mat';
```

Lab 1 writes some data to the file, which all other labs will read.

```
if labindex == 1
    data = randn(100, 1);
    save(fname, 'data');
    pause(5) %allow time for file to become available to other labs
end
```

All labs wait until all have reached the barrier; this ensures that no lab attempts to load the file until lab 1 writes to it.

```
labBarrier;
load(fname);
```

See Also

`labBroadcast` | `labReceive` | `labSend` | `labSendReceive`

labBroadcast

Purpose

Send data to all labs or receive data sent to all labs

Syntax

```
shared_data = labBroadcast(senderlab, data)
shared_data = labBroadcast(senderlab)
```

Arguments

`senderlab`	The `labindex` of the lab sending the broadcast.
`data`	The data being broadcast. This argument is required only for the lab that is broadcasting. The absence of this argument indicates that a lab is receiving.
`shared_data`	The broadcast data as it is received on all other labs.

Description

`shared_data = labBroadcast(senderlab, data)` sends the specified `data` to all executing labs. The data is broadcast from the lab with `labindex == senderlab`, and received by all other labs.

`shared_data = labBroadcast(senderlab)` receives on each executing lab the specified `shared_data` that was sent from the lab whose `labindex` is `senderlab`.

If `labindex` is not `senderlab`, 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 labs.

This function blocks execution until the lab's involvement in the collective broadcast operation is complete. Because some labs may complete their call to `labBroadcast` before others have started, use `labBarrier` to guarantee that all labs are at the same point in a program.

Examples

In this case, the broadcaster is the lab whose `labindex` is `1`.

```
broadcast_id = 1;
if labindex == broadcast_id
  data = randn(10);
  shared_data = labBroadcast(broadcast_id, data);
else
  shared_data = labBroadcast(broadcast_id);
end
```

See Also

labBarrier | labindex | labSendReceive

labindex

Purpose Index of this lab

Syntax

```
id = labindex
```

Description `id = labindex` returns the index of the lab currently executing the function. `labindex` is assigned to each lab 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 labs running the current job, defined by `numlabs`.

See Also `numlabs`

Purpose

Test to see if messages are ready to be received from other lab

Syntax

```
is_data_available = labProbe
is_data_available = labProbe(source)
is_data_available = labProbe('any',tag)
is_data_available = labProbe(source,tag)
[is_data_available, source, tag] = labProbe
```

Arguments

`source`	`labindex` of a particular lab from which to test for a message.
`tag`	Tag defined by the sending lab's `labSend` function to identify particular data.
`'any'`	String to indicate that all labs should be tested for a message.
`is_data_available`	Boolean indicating if a message is ready to be received.

Description

`is_data_available = labProbe` returns a logical value indicating whether any data is available for this lab to receive with the `labReceive` function.

`is_data_available = labProbe(source)` tests for a message only from the specified lab.

`is_data_available = labProbe('any',tag)` tests only for a message with the specified tag, from any lab.

`is_data_available = labProbe(source,tag)` tests for a message from the specified lab and tag.

`[is_data_available, source, tag] = labProbe` returns `labindex` and tag of ready messages. If no data is available, `source` and `tag` are returned as `[]`.

See Also

`labindex` | `labReceive` | `labSend` | `labSendReceive`

labReceive

Purpose

Receive data from another lab

Syntax

```
data = labReceive
data = labReceive(source)
data = labReceive('any',tag)
data = labReceive(source,tag)
[data, source, tag] = labReceive
```

Arguments

`source`	`labindex` of a particular lab from which to receive data.
`tag`	Tag defined by the sending lab's `labSend` function to identify particular data.
`'any'`	String to indicate that data can come from any lab.
`data`	Data sent by the sending lab's `labSend` function.

Description

`data = labReceive` receives data from any lab with any tag.

`data = labReceive(source)` receives data from the specified lab with any tag

`data = labReceive('any',tag)` receives data from any lab with the specified tag.

`data = labReceive(source,tag)` receives data from only the specified lab with the specified tag.

`[data, source, tag] = labReceive` returns the source and tag with the data.

Tips

This function blocks execution in the lab until the corresponding call to `labSend` occurs in the sending lab.

For a demonstration that uses `labSend`, `labReceive`, `labBarrier`, and `labSendReceive`, see the demo Profiling Explicit Parallel Communication.

See Also

`labBarrier` | `labindex` | `labProbe` | `labSend` | `labSendReceive`

labSend

Purpose

Send data to another lab

Syntax

```
labSend(data, destination)
labSend(data, destination, tag)
```

Arguments

`data`	Data sent to the other lab; any MATLAB data type.
`destination`	`labindex` of receiving lab.
`tag`	Nonnegative integer to identify data.

Description

`labSend(data, destination)` sends the data to the specified destination, with a tag of `0`.

`labSend(data, destination, tag)` sends the data to the specified destination with the specified `tag`. `data` can be any MATLAB data type. `destination` identifies the `labindex` of the receiving lab, and must be either a scalar or a vector of integers between `1` and `numlabs`; it cannot be `labindex` (i.e., the current lab). `tag` can be any integer from `0` to `32767`.

Tips

This function might or might not return before the corresponding `labReceive` completes in the receiving lab.

For a demonstration that uses `labSend`, `labReceive`, `labBarrier`, and `labSendReceive`, see the demo Profiling Explicit Parallel Communication.

See Also

`labBarrier` | `labindex` | `labProbe` | `labReceive` | `labSendReceive` | `numlabs`

Purpose

Simultaneously send data to and receive data from another lab

Syntax

```
received = labSendReceive(labTo, labFrom, data)
received = labSendReceive(labTo, labFrom, data, tag)
```

Arguments

`data`	Data on the sending lab that is sent to the receiving lab; any MATLAB data type.
`received`	Data accepted on the receiving lab.
`labTo`	`labindex` of the lab to which data is sent.
`labFrom`	`labindex` of the lab from which data is received.
`tag`	Nonnegative integer to identify data.

Description

`received = labSendReceive(labTo, labFrom, data)` sends `data` to the lab whose `labindex` is `labTo`, and receives `received` from the lab whose `labindex` is `labFrom`. `labTo` and `labFrom` must be scalars. This function is conceptually equivalent to the following sequence of calls:

```
labSend(data, labTo);
received = labReceive(labFrom);
```

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 `labTo` is an empty array, `labSendReceive` does not send data, but only receives. If `labFrom` is an empty array, `labSendReceive` does not receive data, but only sends.

`received = labSendReceive(labTo, labFrom, data, tag)` uses the specified tag for the communication. `tag` can be any integer from `0` to `32767`.

For a demonstration that uses `labSend`, `labReceive`, `labBarrier`, and `labSendReceive`, see the demo Profiling Explicit Parallel Communication.

Examples

Create a unique set of data on each lab, and transfer each lab's data one lab to the right (to the next higher `labindex`).

First use `magic` to create a unique value for the variant array `mydata` on each lab.

```
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 lab on either side, so that each lab will receive data from the lab on the "left" while sending data to the lab on the "right," cycling data from the end lab back to the beginning lab.

```
labTo = mod(labindex, numlabs) + 1; % one lab to the right
labFrom = mod(labindex - 2, numlabs) + 1; % one lab to the left
```

Transfer the data, sending each lab's `mydata` into the next lab's `otherdata` variable, wrapping the third lab's data back to the first lab.

```
otherdata = labSendReceive(labTo, labFrom, 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 lab without wrapping data from the last lab to the first lab.

```
if labindex < numlabs; labTo = labindex + 1; else labTo = []; end;
if labindex > 1; labFrom = labindex - 1; else labFrom = []; end;
otherdata = labSendReceive(labTo, labFrom, mydata)
Lab 1:
  otherdata =
       []
Lab 2:
  otherdata =
       1
Lab 3:
  otherdata =
       1     3
       4     2
```

See Also

`labBarrier` | `labindex` | `labProbe` | `labReceive` | `labSend` | `numlabs`

Purpose

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
```

See Also

`size`

Purpose

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 | getAllOutputArguments

Purpose

Log out of MJS cluster

Syntax

```
logout(c)
```

Description

`logout(c)` logs the you out of the MJS 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.

Examples

See Also

`changePassword`

matlabpool

Purpose

Open or close pool of MATLAB sessions for parallel computation

Syntax

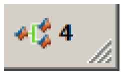
```
matlabpool
matlabpool open
matlabpool open poolsize
matlabpool open profilename
matlabpool open profilename poolsize
matlabpool poolsize
matlabpool(poolsize)
matlabpool profilename
matlabpool profilename poolsize
matlabpool(clusterobj)
matlabpool(clusterobj, 'open')
matlabpool(clusterobj, 'open', ...)
matlabpool(clusterobj, poolsize)
matlabpool close
matlabpool close force
matlabpool close force profilename
matlabpool size
matlabpool('open', ...)
matlabpool('close', ...)
matlabpool('open',..., 'AttachedFiles', filecell)
matlabpool('addattachedfiles', filecell)
matlabpool updateattachedfiles
```

Description

`matlabpool` enables the parallel language features in the MATLAB language (e.g., `parfor`) by starting a parallel job that connects this MATLAB client with a number of workers.

`matlabpool` or `matlabpool` **`open`** starts a worker pool using the default cluster profile, with the pool size specified by that profile. (For information about setting up and selecting profiles, see "Cluster Profiles" on page 6-12.) You can also specify the pool size using `matlabpool` **`open`** `poolsize`, but most clusters have a maximum number of processes that they can start (12 for a local scheduler). If the profile specifies an MJS as the cluster, `matlabpool` reserves its workers from among those already running and available under that MJS. If the profile specifies a third-party scheduler, `matlabpool` instructs the scheduler to start the workers.

Note When you first open a MATLAB pool from your desktop session, an indicator appears in the lower-right corner of the desktop to show that this desktop session is connected to an open pool. The number indicates how many workers are in the pool.



`matlabpool` **`open`** `profilename` or `matlabpool` **`open`** `profilename poolsize` starts a worker pool using the Parallel Computing Toolbox cluster profile identified by `profilename` rather than the default cluster profile to locate a cluster. If the pool size is specified, it overrides the number of workers range specified in the profile, and starts a pool of exactly that number of workers, even if it has to wait for them to be available.

Without specifying **`open`** or **`close`**, the command default is **`open`**. So, `matlabpool poolsize`, `matlabpool(poolsize)`, `matlabpool profilename`, and `matlabpool profilename poolsize` operate as `matlabpool open ...`, and are provided for convenience.

`matlabpool(clusterobj)` or `matlabpool(clusterobj, '`**`open`**`')` is the same as `matlabpool open`, except that the worker pool is started on the cluster specified by the object `clusterobj`.

`matlabpool(clusterobj, '`**`open`**`', ...)` is the same as `matlabpool('`**`open`**`', ...)` except that the worker pool is started on the cluster specified by the object `clusterobj`.

`matlabpool(clusterobj, poolsize)` is the same as `matlabpool poolsize` except that the worker pool is started on the cluster specified by the object `clusterobj`.

`matlabpool` **`close`** stops the worker pool, deletes the pool job, and makes all parallel language features revert to using the MATLAB client for computing their results.

`matlabpool` **`close force`** deletes all pool jobs created by `matlabpool` for the current user under the cluster specified by the default profile, including any jobs currently running.

`matlabpool` **`close force`** `profilename` deletes all pool jobs being run under the cluster specified in the profile `profilename`.

`matlabpool` **`size`** returns the size of the worker pool if it is open, or 0 if the pool is closed.

`matlabpool('`**`open`**`', ...)` and `matlabpool('`**`close`**`', ...)` can be invoked as functions with optional arguments, such as profile name and pool size. The default is `'`**`open`**`'`. For example, the following are equivalent:

```
matlabpool open MyProfile 4
matlabpool('MyProfile', 4)
```

`matlabpool('`**`open`**`',..., '`**`AttachedFiles`**`', filecell)` starts a worker pool and allows you to specify attached files so that you can pass necessary files to the workers in the pool. The cell array `filecell` is appended to the `AttachedFiles` property specified in the profile used for startup. The `'AttachedFiles'` property name is case sensitive, and must appear as shown. (Note: This form of the command does not allow you to directly specify any other job property-value pairs when opening a pool.)

`matlabpool('`**`addattachedfiles`**`', filecell)` allows you to add extra attached files to an already running pool. `filecell` is a cell array of strings, identical in form to those you use when adding attached files to a job or when you open a MATLAB pool. Each string can specify either absolute or relative files, folders, or a file on the MATLAB path. The command transfers the files to each worker, placing the files in the attached files folder, exactly the same as if you sent them at the time the pool was opened.

`matlabpool` **`updateattachedfiles`** checks all the attached files of the current pool to see if they have changed, and replicates any changes to each of the workers in the pool. In this way, you can send code changes out to remote workers. This checks files that you added with the `matlabpool addattachedfiles` command, as well as those you specified when the pool was started (by a profile or command-line argument).

Tips

When a pool of workers is open, the following commands entered in the client's Command Window also execute on all the workers:

- `cd`
- `addpath`
- `rmpath`

This enables you to set the working directory and the path on all the workers, so that a subsequent `parfor`-loop executes in the proper context.

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 see or access, the `addpath` command is not executed on the workers. However, if the working directory or path 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.

This slight difference in behavior might be an issue 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 MATLAB 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
```

Examples

Start a pool using the default profile to define the number of workers:

```
matlabpool
```

Start a pool of 16 workers using a profile called `myProf`:

```
matlabpool open myProf 16
```

Start a pool of 2 workers using the local profile:

```
matlabpool local 2
```

Run `matlabpool` as a function to check whether the worker pool is currently open:

```
isOpen = matlabpool('size') > 0
```

Start a pool with the default profile, and pass two code files to the workers:

```
matlabpool('open', 'AttachedFiles', {'mod1.m', 'mod2.m'})
```

Start a MATLAB pool with the cluster and pool size determined by the default profile:

```
c = parcluster
matlabpool(c)
```

See Also

`defaultParallelConfig` | `pctRunOnAll` | `parfor`

methods

Purpose

List functions of object class

Syntax

```
methods(obj)
out = methods(obj)
```

Arguments

`obj`	An object or an array of objects.
`out`	Cell array of strings.

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 strings.

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` | `get`

Purpose

Location of MPI implementation

Syntax

```
[primaryLib, extras] = mpiLibConf
```

Arguments

`primaryLib`	MPI implementation library used by a parallel job.
`extras`	Cell array of other required library names.

Description

`[primaryLib, extras] = mpiLibConf` returns the MPI implementation library to be used by a parallel 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/distcomp/user`. Your `mpiLibConf.m` file must be higher on the cluster workers' path than *`matlabroot`*`/toolbox/distcomp/mpi`. (Sending `mpiLibConf.m` as a file dependency for this purpose does not work.)

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 MathWorks job manager or the local scheduler, 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.

Examples

Use the `mpiLibConf` function to view the current MPI implementation library:

```
mpiLibConf
    mpich2.dll
```

Purpose

Profile parallel communication and execution times

Syntax

```
mpiprofile
mpiprofile on <options>
mpiprofile off
mpiprofile resume
mpiprofile clear
mpiprofile status
mpiprofile reset
mpiprofile info
mpiprofile viewer
mpiprofile('viewer', <profinfoarray>)
```

Description

`mpiprofile` enables or disables the parallel profiler data collection on a MATLAB worker running a parallel job. `mpiprofile` aggregates statistics on execution time and communication times. The statistics are collected 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 labs. This command in general should be executed in pmode or as part of a task in a parallel job.

`mpiprofile on <options>` starts the parallel profiler and clears previously recorded profile statistics.

`mpiprofile` takes the following options.

Option	Description

`-detail mmex`
`-detail builtin`
	This option specifies the set of functions for which profiling statistics are gathered. `-detail mmex` (the default) records information about functions, subfunctions, and MEX-functions. `-detail builtin` additionally

Option	Description
	records information about built-in functions such as `eig` or `labReceive`.

`-messagedetail default`
 	
`-messagedetail simplified`
	This option specifies the detail at which communication information is stored.
	
	`-messagedetail default` collects information on a per-lab 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-lab communication plots in the viewer.
	
	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, you should also call `mpiprofile reset`.

`mpiprofile resume` restarts the profiler without clearing previously recorded function statistics. This works only in pmode or in the same MATLAB worker session.

`mpiprofile clear` clears the profile information.

`mpiprofile status` returns a valid status when it runs on the worker.

`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 info` returns a profiling data structure with additional fields to the one provided by the standard `profile info` 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 lab
`TimeWastedPerLab`	Vector of communication waiting time for each lab
`BytesReceivedPerLab`	Vector of data received from each lab

The three `*PerLab` fields are collected only on a per-function basis, and can be turned off by typing the following command in pmode:

```
mpiprofile on -messagedetail simplified
```

`mpiprofile viewer` is used in pmode after running user code with `mpiprofile on`. Calling the 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('viewer', <profinfoarray>)` in function form can be used from the client. A structure `<profinfoarray>` needs be passed in as the second argument, which is an array of `mpiprofile info` structures. See `pInfoVector` in the Examples section below.

`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 6-31.

Examples

In pmode, turn on the parallel profiler, run your function in parallel, and call the viewer:

```
mpiprofile on;
% call your function;
mpiprofile viewer;
```

If you want to obtain the profiler information from a parallel job outside of pmode (i.e., in the MATLAB client), you need to return output arguments of `mpiprofile info` by using the functional form of the command. Define your function `foo()`, and make it the task function in a parallel job:

```
function [pInfo, yourResults] = foo
mpiprofile on
initData = (rand(100, codistributor()) ...
                   * rand(100, codistributor()));
pInfo = mpiprofile('info');
```

```
yourResults = gather(initData,1)
```

After the job runs and `foo()` is evaluated on your cluster, get the data on the client:

```
A = getAllOutputArguments(yourJob);
```

Then view parallel profile information:

```
pInfoVector = [A{:, 1}];
mpiprofile('viewer', pInfoVector);
```

See Also

`profile` | `mpiSettings` | `pmode`

mpiSettings

Purpose

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 labs, this is the most useful option. The default value is `'off'` for parallel jobs, and `'on'` inside `pmode` sessions or `spmd` statements. Once the setting has been changed within a `pmode` session or an `spmd` statement, the setting stays in effect until either the `pmode` session ends or the MATLAB pool is closed.

`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 parallel 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 job manager, this is the mdce service log file; if you are using an mpiexec scheduler, this is the mpiexec debug log, which you can read with `getDebugLog`.

`mpiSettings('MessageLoggingDestination','File','filename')` sends MPI logging information to the specified file.

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 lab, not the client. That is, it should be called within the task function, within `jobStartup.m`, or within `taskStartup.m`.

Examples

Set deadlock detection for a parallel job inside the `jobStartup.m` file for that job:

```
% Inside jobStartup.m for the parallel 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 MATLAB pool:

```
spmd; mpiSettings('DeadlockDetection', 'off'); end
```

numlabs

Purpose

Total number of labs operating in parallel on current job

Syntax

```
n = numlabs
```

Description

`n = numlabs` returns the total number of labs currently operating on the current job. This value is the maximum value that can be used with `labSend` and `labReceive`.

See Also

`labindex` | `labReceive` | `labSend`

Purpose

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()
myCluster = parcluster(allNames{end});
```

See Also

`parallel.defaultClusterProfile` | `parallel.exportProfile` | `parallel.importProfile`

parallel.defaultClusterProfile

Purpose

Examine or set default cluster profile

Syntax

```
p = parallel.defaultClusterProfile
oldprofile = parallel.defaultClusterProfile(newprofile)
```

Description

`p = parallel.defaultClusterProfile` returns the name of the current 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 by selecting **Parallel > Manage Cluster Profiles**.

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`

Purpose

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 `.settings` 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.settings`.

```
parallel.exportProfile('MyProfile','MyExportedProfile')
```

Export the default profile to the file `MyDefaultProfile.settings`.

```
def_profile = parallel.defaultClusterProfile();
parallel.exportProfile(def_profile,'MyDefaultProfile')
```

Export all profiles except for `local` to the file `AllProfiles.settings`.

```
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`

parallel.gpu.CUDAKernel

Purpose

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 kernel 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 string; and `CPROTO` is the C prototype for the kernel call that `KERN` represents. If specified, `FUNC` must be a string 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)` 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`.

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

parallel.importProfile

Purpose

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, `.settings` is assumed; configuration files must be specified with the `.mat` extension. Configuration `.mat` files contain only one profile, but profile `.settings` files can contain one or more profiles. If only one profile is defined in the file, then `prof` is a string reflecting the name of the profile; if multiple profiles are defined in the file, then `prof` is a cell array of strings. 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.settings` 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.settings`, and use the first one to open a MATLAB pool.

```
profs = parallel.importProfile('ManyProfiles');
matlabpool('open',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`

parcluster

Purpose

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.

`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.

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 by selecting **Parallel > Manage Cluster Profiles**.

Examples

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 MATLAB pool on the cluster.

```
defaultProfile = parallel.defaultClusterProfile
myCluster = parcluster(defaultProfile);
matlabpool(myCluster, 'open');
```

Find a particular cluster using the profile named `'MyProfile'`, and create an independent job on the cluster.

```
myCluster = parcluster('MyProfile');
j = createJob(myCluster);
```

Purpose

Execute code loop in parallel

Syntax

```
parfor loopvar = initval:endval, statements, end
parfor (loopvar = initval:endval, M), statements, end
```

Description

`parfor loopvar = initval:endval, statements, end` allows you to write a loops for a statement or block of code that executes in parallel on a cluster of workers, which are identified and reserved with the `matlabpool` command. `initval` and `endval` must evaluate to finite integer values, or the range must evaluate to a value that can be obtained by such an expression, that is, an ascending row vector of consecutive integers.

The following table lists some ranges that are not valid.

Invalid parfor Range	Reason Range Not Valid
`parfor i = 1:2:25`	`1, 3, 5,...` are not consecutive.
`parfor i = -7.5:7.5`	`-7.5, -6.5,...` are not integers.

`A = [3 7 -2 6 4 -4 9 3 7];`
`parfor i = find(A>0)`
	The resulting range, `1, 2, 4,...`, has nonconsecutive integers.

`parfor i = [5;6;7;8]`	`[5;6;7;8]` is a column vector, not a row vector.

You can enter a `parfor`-loop on multiple lines, but if you put more than one segment of the loop statement on the same line, separate the segments with commas or semicolons:

```
parfor i = range; <loop body>; end
```

`parfor (loopvar = initval:endval, M), statements, end` uses `M` to specify the maximum number of MATLAB workers that will evaluate statements in the body of the `parfor`-loop. `M` must be a nonnegative integer. By default, MATLAB uses as many workers as it finds available. If you specify an upper limit, MATLAB employs no more than that number, even if additional workers are available. If you request more resources than are available, MATLAB uses the maximum number available at the time of the call.

If the `parfor`-loop cannot run on workers in a MATLAB pool (for example, if no workers are available or `M` is `0`), MATLAB executes the loop on the client in a serial manner. In this situation, the `parfor` semantics are preserved in that the loop iterations can execute in any order.

Note Because of independence of iteration order, execution of `parfor` does not guarantee deterministic results.

The maximum amount of data that can be transferred in a single chunk between client and workers in the execution of a `parfor`-loop is determined by the JVM memory allocation limit. For details, see “Object Data Size Limitations” on page 6-44.

For a detailed description of `parfor`-loops, see Chapter 2, “Parallel for-Loops (parfor)”.

Examples

Suppose that `f` is a time-consuming function to compute, and that you want to compute its value on each element of array `A` and place the corresponding results in array `B`:

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

Because the loop iteration occurs in parallel, this evaluation can complete much faster than it would in an analogous `for`-loop.

Next assume that `A`, `B`, and `C` are variables and that `f`, `g`, and `h` are functions:

```
parfor i = 1:n
   t = f(A(i));
   u = g(B(i));
   C(i) = h(t, u);
end
```

If the time to compute f, g, and h is large, parfor will be significantly faster than the corresponding for statement, even if n is relatively small. Although the form of this statement is similar to a for statement, the behavior can be significantly different. Notably, the assignments to the variables i, t, and u do *not* affect variables with the same name in the context of the parfor statement. The rationale is that the body of the parfor is executed in parallel for all values of i, and there is no deterministic way to say what the "final" values of these variables are. Thus, parfor is defined to leave these variables unaffected in the context of the parfor statement. By contrast, the variable C has a different element set for each value of i, and these assignments *do* affect the variable C in the context of the parfor statement.

Another important use of parfor has the following form:

```
s = 0;
parfor i = 1:n
   if p(i)   % assume p is a function
      s = s + 1;
   end
end
```

The key point of this example is that the conditional adding of 1 to s can be done in any order. After the parfor statement has finished executing, the value of s depends only on the number of iterations for which p(i) is true. As long as p(i) depends only upon i, the value of s is deterministic. This technique generalizes to functions other than plus (+).

Note that the variable s does refer to the variable in the context of the parfor statement. The general rule is that the only variables in the context of a parfor statement that can be affected by it are those like s (combined by a suitable function like +) or those like C in the previous example (set by indexed assignment).

See Also `for` | `matlabpool` | `pmode` | `numlabs`

Purpose

Pause job manager queue

Syntax

```
pause(jm)
```

Arguments

`jm`	Job manager object whose queue is paused.

Description

`pause(jm)` pauses the job manager'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 job manager.

The pause function does nothing if the job manager is already paused.

See Also

`resume` | `waitForState`

pctconfig

Purpose

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 string 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 Distributed Computing 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.

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.

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');
```

pctRunDeployedCleanup

Purpose 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.

Purpose

Run command on client and all workers in matlabpool

Syntax

```
pctRunOnAll command
```

Description

`pctRunOnAll command` runs the specified `command` on all the workers of the matlabpool 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 labs 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 labs. For example, if your labs 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 labs:

```
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 labs:

```
pctRunOnAll addpath({'/usr/share/path1' '/usr/share/path2'})
```

See Also

`matlabpool`

Purpose

Load file into parallel session

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 labs running a parallel job. The files should have been created by the `psave` command. The number of labs should be the same as the number of files. The files should be accessible to all the labs. 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 lab 1, `abc2.mat` on lab 2, and so on.

Examples

Create three variables — one replicated, one variant, and one codistributed. Then save the data.

```
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 labs and confirm there are no variables.

```
clear all
whos
```

Load the previously saved data into the labs. Confirm its presence.

```
pload('threeThings');
whos
isreplicated(rep)
iscodistributed(D)
```

See Also

`load` | `save` | `labindex` | `numlabs` | `pmode` | `psave`

pmode

Purpose

Interactive Parallel Command Window

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 parallel 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 scheduler 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`, but note that the local cluster allows for only up to twelve workers.

`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 parallel 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
```

See Also

createCommunicatingJob | parallel.defaultClusterProfile | parcluster

Purpose

File for user-defined options to run on each worker when MATLAB pool starts

Syntax

`poolStartup`

Description

`poolStartup` runs automatically on a worker each time the worker forms part of a MATLAB 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 `FileDependencies` property.
2. In a folder included in the job's `PathDependencies` property.
3. In the worker's MATLAB installation at the location

   `matlabroot``/toolbox/distcomp/user/poolStartup.m`

To create a version of `poolStartup.m` for `FileDependencies` or `PathDependencies`, copy the provided file and modify it as required. .

`poolStartup` is the ideal location for startup code required for parallel execution on the MATLAB 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 a distributed job, parallel job, or using a MATLAB 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` | `FileDependencies` | `PathDependencies`

promote

Purpose

Promote job in MJS cluster queue

Syntax

```
promote(c,job)
```

Arguments

`c`	The MJS 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 MJS cluster `c`.

If `job` is not the first job in the queue, `promote` exchanges the position of `job` and the previous job.

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 MJS 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)`.

Examples

Create and submit multiple jobs to the cluster identified by the default cluster profile, assuming that the default cluster profile uses an MJS:

```
c = parcluster();
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'
```

See Also

`createJob` | `demote` | `findJob` | `submit`

Purpose

Save data from parallel job session

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

This example works in a communicating (parallel) job or in pmode, but not in a parfor or spmd block. Create three arrays — one replicated, one variant, and one codistributed. Then save the data.

```
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 labs. Confirm its presence.

```
pload('threeThings');
whos
isreplicated(rep)
iscodistributed(D)
```

See Also

load | save | labindex | numlabs | pmode | pload

redistribute

Purpose

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`

Purpose

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.

Arguments

`gpudev`	GPUDevice object representing 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.

Examples

Reset GPU Device

Create a GPUArray on the selected GPU device, then reset the 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(g);
g   % Show that the device is still selected

  parallel.gpu.CUDADevice handle
  Package: parallel.gpu

  Properties:
```

```
                  Name: 'Tesla C1060'
                 Index: 1
     ComputeCapability: '1.3'
        SupportsDouble: 1
         DriverVersion: 4
    MaxThreadsPerBlock: 512
      MaxShmemPerBlock: 16384
    MaxThreadBlockSize: [512 512 64]
           MaxGridSize: [65535 65535]
             SIMDWidth: 32
           TotalMemory: 4.2948e+09
            FreeMemory: 4.2091e+09
   MultiprocessorCount: 30
          ClockRateKHz: 1296000
           ComputeMode: 'Default'
  GPUOverlapsTransfers: 1
KernelExecutionTimeout: 0
      CanMapHostMemory: 1
       DeviceSupported: 1
        DeviceSelected: 1
```

```
whos

  Name      Size      Bytes  Class
  g         1x1         112  parallel.gpu.CUDADevice
  M         1x1         108  parallel.gpu.GPUArray

M  % Try to display GPUArray

Data no longer exists on the GPU.

clear M
```

See Also gpuDevice | gpuArray | parallel.gpu.CUDAKernel

Purpose

Resume processing queue in job manager

Syntax

```
resume(jm)
```

Arguments

`jm`	Job manager object whose queue is resumed.

Description

`resume(jm)` resumes processing of the job manager's queue so that jobs waiting in the queued state will be run. This call will do nothing if the job manager is not paused.

See Also

`pause` | `waitForState`

saveAsProfile

Purpose

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');
```

See Also

`parcluster` | `saveProfile`

Purpose

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.NumWorkers = 3;  % 'Modified' property now TRUE
saveProfile(myCluster);    % 'local' profile now updated,
                           % 'Modified' property now FALSE
```

See Also

`parcluster` | `saveAsProfile`

Purpose

Configure or display object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
set(obj,'configuration', 'ConfigurationName',...)
```

Arguments

`obj`	An object or an array of objects.
`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *`PropertyName`*.
`PN`	A cell array of property names.
`PV`	A cell array of property values.
`props`	A structure array whose field names are the property names for `obj`.
`S`	A structure with property names and property values.
`'`**`configuration`**`'`	Literal string to indicate usage of a configuration.
`'ConfigurationName'`	Name of the configuration to use.

Description

`set(obj)` displays all configurable properties for `obj`. If a property has a finite list of possible string values, these values are also displayed.

`props = set(obj)` returns all configurable properties for `obj` and their possible values to the structure `props`. The field names of `props` are the property names of `obj`, and the field values are cell arrays of possible property values. If a property does not have a finite set of possible values, its cell array is empty.

`set(obj,'`*`PropertyName`*`')` displays the valid values for *`PropertyName`* if it possesses a finite list of string values.

`props = set(obj,'`*`PropertyName`*`')` returns the valid values for *`PropertyName`* `to props`. `props` is a cell array of possible string values or an empty cell array if *`PropertyName`* does not have a finite list of possible values.

`set(obj,'`*`PropertyName`*`',PropertyValue,...)` configures one or more property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be `m`-by-`n`, where `m` is equal to the number of objects in `obj` and `n` is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are object properties, and whose field values are the values for the corresponding properties.

`set(obj,'`**`configuration`**`', 'ConfigurationName',...)` sets the object properties with values specified in the configuration `ConfigurationName`. For details about defining and applying configurations, see "Cluster Profiles" on page 6-12.

Tips

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `j1` is a job object, the following commands are all valid and have the same result:

```
set(j1,'Timeout',20)
set(j1,'timeout',20)
set(j1,'timeo',20)
```

Examples

This example illustrates some of the ways you can use `set` to configure property values for the job object `j1`.

```
set(j1,'Name','Job_PT109','Timeout',60);

props1 = {'Name' 'Timeout'};
values1 = {'Job_PT109' 60};
set(j1, props1, values1);

S.Name = 'Job_PT109';
S.Timeout = 60;
set(j1,S);
```

See Also

get | inspect

Purpose

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 defines the following constants:

```
__constant__ int N;
__constant__ double CONST_DATA[256];
```

you can fill these with MATLAB data as follows:

```
KERN = parallel.gpu.CUDAKernel(ptxFile, cudaFile);
setConstantMemory(KERN,'N',int16(10));
setConstantMemory(KERN,'CONST_DATA',1:10);
```

or

```
setConstantMemory(KERN,'N',int16(10),'CONST_DATA',1:10);
```

See Also

`GPUArray` | `parallel.gpu.CUDAKernel`

setJobClusterData

Purpose

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.

You should call the function `setJobClusterData` in the submit function (identified by the `IndependentSubmitFcn` or `CommunicatingSubmitFcn` property) and call `getJobSchedulerData` in any of the functions identified by the properties `GetJobStateFcn`, `DeleteJobFcn`, `DeleteTaskFcn`, `CancelJobFcn`, or `CancelTaskFcn`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

See Also

`getJobClusterData`

Purpose

Set specific user data for job on generic scheduler

Syntax

```
setJobSchedulerData(sched, job, userdata)
```

Arguments

`sched`	Scheduler object identifying the generic third-party scheduler running the job.
`job`	Job object identifying the job for which to store data.
`userdata`	Information to store for this job.

Description

`setJobSchedulerData(sched, job, userdata)` stores data for the job `job` that is running under the generic scheduler `sched`. You can later retrieve the information with the function `getJobSchedulerData`. 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.

You should call the function `setJobSchedulerData` in the submit function (identified by the `SubmitFcn` property) and call `getJobSchedulerData` in any of the functions identified by the properties `GetJobStateFcn`, `DestroyJobFcn`, `DestroyTaskFcn`, `CancelJobFcn`, or `CancelTaskFcn`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

See Also

`getJobSchedulerData`

setupForParallelExecution

Purpose

Set options for submitting parallel jobs to scheduler

Syntax

```
setupForParallelExecution(sched, 'pc')
setupForParallelExecution(sched, 'pcNoDelegate')
setupForParallelExecution(sched, 'unix')
```

Arguments

`sched`	Platform LSF, PBS Pro, or TORQUE scheduler object.
`'pc'`, `'pcNoDelegate'`, `'unix'`	Setting for parallel execution.

Description

`setupForParallelExecution(sched, 'pc')` sets up the scheduler to expect workers running on Microsoft Windows operating systems, and selects the wrapper script which expects to be able to call "mpiexec -delegate" on the workers. Note that you still need to supply `SubmitArguments` that ensure that the LSF or PBS Pro scheduler runs your job only on PC-based workers. For example, for LSF, including `'-R type==NTX86'` in your `SubmitArguments` causes the scheduler to select only workers on 32-bit Windows operating systems.

`setupForParallelExecution(sched, 'pcNoDelegate')` is similar to the `'pc'` mode, except that the wrapper script does not attempt to call `"mpiexec -delegate"`, and so assumes that you have installed some other means of achieving authentication without passwords.

`setupForParallelExecution(sched, 'unix')` sets up the scheduler to expect workers running on UNIX operating systems, and selects the default wrapper script for UNIX-based workers. You still need to supply `SubmitArguments` to ensure that the LSF, PBS Pro, or TORQUE scheduler runs your job only on UNIX-based workers. For example, for LSF, including `'-R type==LINUX64'` in your `SubmitArguments` causes the scheduler to select only 64-bit Linux-based workers.

This function sets the values for the properties `ParallelSubmissionWrapperScript` and `ClusterOsType`.

Examples

From any client, set up the scheduler to run parallel jobs only on Windows-based (PC) workers.

```
lsf_sched = findResource('scheduler', 'Type', 'lsf');
setupForParallelExecution(lsf_sched, 'pc');
set(lsf_sched, 'SubmitArguments', '-R type==NTX86');
```

From any client, set up the scheduler to run parallel jobs only on UNIX-based workers.

```
lsf_sched = findResource('scheduler', 'Type', 'lsf');
setupForParallelExecution(lsf_sched, 'unix');
set(lsf_sched, 'SubmitArguments', '-R type==LINUX64');
```

See Also

`createParallelJob` | `findResource`

Purpose

Size of object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

`obj`	An object or an array of objects.
`dim`	The dimension of `obj`.
`d`	The number of rows and columns in `obj`.
`m`	The number of rows in `obj`, or the length of the dimension specified by `dim`.
`n`	The number of columns in `obj`.
`m1,m2,m3,...,mn`	The lengths of the first `n` dimensions of `obj`.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

`length`

Purpose

Create sparse distributed or codistributed matrix

Syntax

```
SD = sparse(FD)
SC = sparse(m, n, codist)
SC = sparse(m, n, codist, 'noCommunication')
```

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`, pmode, or a parallel 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`, pmode, or a parallel job.

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 labs,

```
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 lab 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 lab contains a 10-by-`labindex` local piece of `C`.

Convert a distributed array into a sparse distributed array:

```
R = distributed.rand(1000);
D = floor(2*R); % D also is distributed
SD = sparse(D); % SD is sparse distributed
```

See Also

`sparse` | `distributed.spalloc` | `codistributed.spalloc`

Purpose

Execute code in parallel on MATLAB pool

Syntax

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

Description

The general form of an `spmd` (single program, multiple data) statement is:

```
spmd
    statements
end
```

`spmd, statements, end` defines an `spmd` statement on a single line. MATLAB executes the `spmd` body denoted by `statements` on several MATLAB workers simultaneously. The `spmd` statement can be used only if you have Parallel Computing Toolbox. To execute the statements in parallel, you must first open a pool of MATLAB workers using `matlabpool`.

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 parallel 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 MATLAB pool remains open.

By default, MATLAB uses as many workers as it finds available in the pool. When there are no MATLAB workers available, MATLAB executes the block body locally and creates Composite objects as necessary.

`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 MATLAB 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.

For more information about `spmd` and Composite objects, see Chapter 3, "Single Program Multiple Data (spmd)".

Tips

For information about restrictions and limitations when using `spmd`, see "Limitations" on page 3-15.

Examples

Perform a simple calculation in parallel, and plot the results:

```
matlabpool(3)
spmd
  % build magic squares in parallel
  q = magic(labindex + 2);
end
for ii=1:length(q)
  % plot each magic square
  figure, imagesc(q{ii});
end
matlabpool close
```

See Also

`batch` | `Composite` | `labindex` | `matlabpool` | `numlabs` | `parfor`

Purpose

Queue job in scheduler

Syntax

```
submit(j)
```

Arguments

`j`	Job object to be queued.

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.

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`.

Examples

Find the MJS cluster identified by the cluster profile `Profile1`.

```
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);
```

See Also

`createCommunicatingJob` | `createJob` | `findJob` | `parcluster` | `promote`

subsasgn

Purpose 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 MATLAB pool.

`C(i) = {B}` sets the entry of `C` on lab `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 labs `i1` and `i2`.

`C{i} = B` sets the entry of `C` on lab `i` to the value `B`.

See Also `subsasgn` | `Composite` | `subsref`

Purpose

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 MATLAB pool.

`B = C(i)` returns the entry of Composite `C` from lab `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 lab `i` as a single entry.

`[B1, B2, ...]  = C{[i1, i2, ...]}` returns multiple entries.

See Also

`subsref` | `Composite` | `subsasgn`

taskFinish

Purpose

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` (or `FileDependencies`) property.
2. In a folder included in the job's `AdditionalPaths` (or `PathDependencies`) property.
3. In the worker's MATLAB installation at the location

   `matlabroot``/toolbox/distcomp/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`

Purpose

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` (or `FileDependencies`) property.
2. In a folder included in the job's `AdditionalPaths` (or `PathDependencies`) property.
3. In the worker's MATLAB installation at the location

   `matlabroot``/toolbox/distcomp/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`

Purpose

Wait for job to change state or for GPU calculation to complete

Syntax

```
wait(j)
wait(j,'state')
wait(j,'state',timeout)
wait(gpudev)
```

Arguments

`job`	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.
`gpudev`	GPU device object whose calculations to wait for.

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.

`wait(j,'state',timeout)` blocks execution until either the job reaches the specified `'state'`, or `timeout` seconds elapse, whichever happens first.

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`.

`wait(gpudev)` blocks execution in MATLAB until the GPU device identified by the object `gpudev` completes its calculations. 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. `wait` can be useful when you are timing GPU calculations to profile your code.

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` | `waitForState`

waitForState

Purpose

Wait for object to change state

Syntax

```
waitForState(obj)
waitForState(obj,'state')
waitForState(obj,'state',timeout)
OK = waitForState(..., timeout)
```

Arguments

`obj`	Job or task object whose change in state to wait for.
`'state'`	Value of the object's `State` property to wait for.
`timeout`	Maximum time to wait, in seconds.
`OK`	Boolean `true` if wait succeeds, `false` if times out.

Description

`waitForState(obj)` blocks execution in the client session until the job or task identified by the object `obj` reaches the `'finished'` state or fails. For a job object, this occurs when all its tasks are finished processing on remote workers.

`waitForState(obj,'state')` blocks execution in the client session until the specified object changes state to the value of `'state'`. For a job object, the valid states to wait for are `'queued'`, `'running'`, and `'finished'`. For a task object, the valid states are `'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 `waitForState(job, 'queued')` for a job already in the `'finished'` state, the call returns immediately.

`waitForState(obj,'state',timeout)` blocks execution until either the object reaches the specified `'state'`, or `timeout` seconds elapse, whichever happens first.

`OK = waitForState(..., timeout)` returns a value of `true` to `OK` if the awaited state occurs, or `false` if the wait times out.

Notes `waitForState` is available only for objects using the scheduler interface resulting from `findResource` or parallel configurations. It is not available on objects using the cluster interface of `parcluster` or cluster profiles.

Simulink models cannot run while a MATLAB session is blocked by `waitForState`. If you must run Simulink from the MATLAB client while also running distributed or parallel jobs, you cannot use `waitForState`.

Examples

Submit a job to the queue, and wait for it to finish running before retrieving its results.

```
submit(job)
waitForState(job,'finished')
results = getAllOutputArguments(job)
```

See Also

`pause` | `resume` | `wait`

Property Reference

Job Manager

`BusyWorkers`	Workers currently running tasks
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`Configuration`	Specify configuration to apply to object or toolbox function
`HostAddress`	IP address of host running job manager or worker session
`Hostname`	Name of host running job manager or worker session
`IdleWorkers`	Idle workers available to run tasks
`IsUsingSecureCommunication`	True if job manager and workers use secure communication
`Jobs`	Jobs contained in job manager service or in scheduler's data location
`Name`	Name of job manager, job, or worker object
`NumberOfBusyWorkers`	Number of workers currently running tasks
`NumberOfIdleWorkers`	Number of idle workers available to run tasks
`PromptForPassword`	Specify if system should prompt for password when authenticating user
`SecurityLevel`	Security level controlling access to job manager and its jobs
`State`	Current state of task, job, job manager, or worker
`Type`	Type of scheduler object

`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

Schedulers

`CancelJobFcn`	Specify function to run when canceling job on generic scheduler
`CancelTaskFcn`	Specify function to run when canceling task on generic scheduler
`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterName`	Name of Platform LSF cluster
`ClusterOsType`	Specify operating system of nodes on which scheduler will start workers
`ClusterSize`	Number of workers available to scheduler
`ClusterVersion`	Version of HPC Server scheduler
`Configuration`	Specify configuration to apply to object or toolbox function
`DataLocation`	Specify folder where job data is stored
`DestroyJobFcn`	Specify function to run when destroying job on generic scheduler
`DestroyTaskFcn`	Specify function to run when destroying task on generic scheduler
`EnvironmentSetMethod`	Specify means of setting environment variables for mpiexec scheduler
`GetJobStateFcn`	Specify function to run when querying job state on generic scheduler
`HasSharedFilesystem`	Specify whether nodes share data location
`JobDescriptionFile`	Name of XML job description file for Microsoft Windows HPC Server scheduler

`Jobs`	Jobs contained in job manager service or in scheduler's data location
`JobTemplate`	Name of job template for HPC Server 2008 scheduler
`MasterName`	Name of Platform LSF master node
`MatlabCommandToRun`	MATLAB command that generic scheduler runs to start lab
`MpiexecFileName`	Specify pathname of executable mpiexec command
`ParallelSubmissionWrapperScript`	Script that scheduler runs to start labs
`ParallelSubmitFcn`	Specify function to run when parallel job submitted to generic scheduler
`RcpCommand`	Command to copy files from client
`ResourceTemplate`	Resource definition for PBS Pro or TORQUE scheduler
`RshCommand`	Remote execution command used on worker nodes during parallel job
`SchedulerHostname`	Name of host running Microsoft Windows HPC Server scheduler
`ServerName`	Name of current PBS Pro or TORQUE server machine
`SubmitArguments`	Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler
`SubmitFcn`	Specify function to run when job submitted to generic scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

`UseSOAJobSubmission`	Allow service-oriented architecture (SOA) submission on HPC Server 2008 cluster
`WorkerMachineOsType`	Specify operating system of nodes on which mpiexec scheduler will start labs

Jobs

`AuthorizedUsers`	Specify users authorized to access job
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`FileDependencies`	Directories and files that worker can access
`FinishedFcn`	Specify callback to execute after task or job runs
`FinishTime`	When task or job finished
`ID`	Object identifier
`JobData`	Data made available to all workers for job's tasks
`MaximumNumberOfWorkers`	Specify maximum number of workers to perform job tasks
`MinimumNumberOfWorkers`	Specify minimum number of workers to perform job tasks
`Name`	Name of job manager, job, or worker object
`Parent`	Parent object of job or task
`PathDependencies`	Specify directories to add to MATLAB worker path
`QueuedFcn`	Specify function file to execute when job is submitted to job manager queue
`RestartWorker`	Specify whether to restart MATLAB workers before evaluating job tasks
`RunningFcn`	Specify function file to execute when job or task starts running
`StartTime`	When job or task started

`State`	Current state of task, job, job manager, or worker
`SubmitArguments`	Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler
`SubmitTime`	When job was submitted to queue
`Tag`	Specify label to associate with job object
`Task`	First task contained in MATLAB pool job object
`Tasks`	Tasks contained in job object
`Timeout`	Specify time limit to complete task or job
`UserData`	Specify data to associate with object
`UserName`	User who created job or job manager object

Tasks

`AttemptedNumberOfRetries`	Number of times failed task was rerun
`CaptureCommandWindowOutput`	Specify whether to return Command Window output
`CommandWindowOutput`	Text produced by execution of task object's function
`Configuration`	Specify configuration to apply to object or toolbox function
`CreateTime`	When task or job was created
`Error`	Task error information
`ErrorIdentifier`	Task error identifier
`ErrorMessage`	Message from task error
`FailedAttemptInformation`	Information returned from failed task
`FinishedFcn`	Specify callback to execute after task or job runs
`FinishTime`	When task or job finished
`Function`	Function called when evaluating task
`ID`	Object identifier
`InputArguments`	Input arguments to task object
`MaximumNumberOfRetries`	Specify maximum number of times to rerun failed task
`NumberOfOutputArguments`	Number of arguments returned by task function
`OutputArguments`	Data returned from execution of task
`Parent`	Parent object of job or task
`RunningFcn`	Specify function file to execute when job or task starts running

`StartTime`	When job or task started
`State`	Current state of task, job, job manager, or worker
`Timeout`	Specify time limit to complete task or job
`UserData`	Specify data to associate with object
`Worker`	Worker session that performed task

Workers

`Computer`	Information about computer on which worker is running
`CurrentJob`	Job whose task this worker session is currently evaluating
`CurrentTask`	Task that worker is currently running
`HostAddress`	IP address of host running job manager or worker session
`Hostname`	Name of host running job manager or worker session
`JobManager`	Job manager that this worker is registered with
`Name`	Name of job manager, job, or worker object
`PreviousJob`	Job whose task this worker previously ran
`PreviousTask`	Task that this worker previously ran
`State`	Current state of task, job, job manager, or worker

Properties — Alphabetical List

AttemptedNumberOfRetries property

Purpose

Number of times failed task was rerun

Description

If a task reruns because of certain system failures, the task property `AttemptedNumberOfRetries` stores a count of the number of attempted reruns.

Note The `AttemptedNumberOfRetries` property is available only when using the MathWorks job manager as your scheduler.

Characteristics

Usage	Task object
Read-only	Always
Data type	Double

See Also

Properties

`FailedAttemptInformation`, `MaximumNumberOfRetries`

Purpose

Specify users authorized to access job

Description

The `AuthorizedUsers` property value is a cell array of strings which identify the users who are allowed to access the job. This controls who can set properties on the job, add tasks, destroy the job, etc. The person identified as the owner by the job's `UserName` property does not have to be listed in the `AuthorizedUsers` property value.

The following table explains the effect of `AuthorizedUsers` at different security levels.

Security Level	Effect of AuthorizedUsers
0	No effect. All users can access the job without hindrance.
1	For users included in the property value, the system suppresses the dialog box that requires acknowledgment that the job belongs to another user. All other users must acknowledge job ownership every time they access the job.
2 and 3	Only users who are authenticated in this session and are listed in `AuthorizedUsers` can access the job.

Note The `AuthorizedUsers` property is available only when using the MathWorks job manager as your scheduler.

Characteristics

Usage	Job object
Read-only	Never
Data type	Cell array of strings

Values

You can populate `AuthorizedUsers` with the names of any users. At security levels 1–3, the users must be recognized by the job manager as authenticated in the session in which you are setting the property.

Examples

This example creates a job named `Job33`, then adds the users `sammy` and `bob` to the job's `AuthorizedUsers`.

```
jm = findResource('scheduler','Configuration',defaultparallelconfig);
j = createJob(jm,'Name','Job33');
set(j,'AuthorizedUsers',{'sammy','bob'})
```

See Also

Properties

`SecurityLevel`, `UserName`

Purpose

Block size of codistributor2dbc object

Description

`blksz = dist.BlockSize` returns the block size of codistributor2dbc object `dist`. The default value is 64. You can read this property only by using dot-notation; not the `get` function.

For more information on 2dbc distribution and the block size of distributed arrays, see "2-Dimensional Distribution" on page 5-17.

Characteristics

Usage	codistributor2dbc object
Read-only	Always
Data type	Double

See Also

Functions

`codistributor2dbc`

Properties

`LabGrid`, `Orientation`

BusyWorkers property

Purpose

Workers currently running tasks

Description

The `BusyWorkers` property value indicates which workers are currently running tasks for the job manager.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Array of worker objects

Values

As workers complete tasks and assume new ones, the lists of workers in `BusyWorkers` and `IdleWorkers` can change rapidly. If you examine these two properties at different times, you might see the same worker on both lists if that worker has changed its status between those times.

If a worker stops unexpectedly, the job manager's knowledge of that as a busy or idle worker does not get updated until the job manager runs the next job and tries to send a task to that worker.

Examples

Examine the workers currently running tasks for a particular job manager.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
workers_running_tasks = get(jm, 'BusyWorkers')
```

See Also

Properties

`ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

Purpose

Specify function to run when canceling job on generic scheduler

Description

`CancelJobFcn` specifies a function to run when you call `cancel` for a job running on a generic scheduler. This function lets you communicate with the scheduler, to provide any instructions beyond the normal toolbox action of changing the state of the job. To identify the job for the scheduler, the function should include a call to `getJobSchedulerData`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

You can set `CancelJobFcn` to any valid function handle or a cell array whose first element is a function handle.

See Also

Functions

`cancel`, `getJobSchedulerData`, `setJobSchedulerData`

Properties

`CancelTaskFcn`, `DestroyJobFcn`, `DestroyTaskFcn`

CancelTaskFcn property

Purpose

Specify function to run when canceling task on generic scheduler

Description

`CancelTaskFcn` specifies a function to run when you call `cancel` for a task running on a generic scheduler. This function lets you communicate with the scheduler, to provide any instructions beyond the normal toolbox action of changing the state of the task. To identify the task for the scheduler, the function should include a call to `getJobSchedulerData`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

You can set `CancelTaskFcn` to any valid function handle or a cell array whose first element is a function handle.

See Also

Functions

`cancel`, `getJobSchedulerData`, `setJobSchedulerData`

Properties

`CancelJobFcn`, `DestroyJobFcn`, `DestroyTaskFcn`

Purpose

Specify whether to return Command Window output

Description

`CaptureCommandWindowOutput` specifies whether to return command window output for the evaluation of a task object's `Function` property.

If `CaptureCommandWindowOutput` is set `true` (or logical `1`), the command window output will be stored in the `CommandWindowOutput` property of the task object. If the value is set false (or logical 0), the task does not retain command window output.

Characteristics

Usage	Task object
Read-only	While task is running or finished
Data type	Logical

Values

The value of `CaptureCommandWindowOutput` can be set to `true` (or logical `1`) or `false` (or logical `0`). When you perform `get` on the property, the value returned is logical `1` or logical `0`. The default value is logical `0` to save network bandwidth in situations where the output is not needed; except for batch jobs, whose default is `1` (`true`).

Examples

Set all tasks in a job to retain any command window output generated during task evaluation.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
createTask(j, @myfun, 1, {x});
createTask(j, @myfun, 1, {x});
.
.
.
alltasks = get(j, 'Tasks');
set(alltasks, 'CaptureCommandWindowOutput', true)
```

See Also

Properties

`Function`, `CommandWindowOutput`

Purpose

Specify MATLAB root for cluster

Description

`ClusterMatlabRoot` specifies the path name to MATLAB for the cluster to use for starting MATLAB worker processes. The path must be available from all nodes on which worker sessions will run. When using the generic scheduler interface, your scheduler script can construct a path to the executable by concatenating the values of `ClusterMatlabRoot` and `MatlabCommandToRun` into a single string.

Characteristics

Usage	Scheduler object
Read-only	Never
Data type	String

Values

`ClusterMatlabRoot` is a string. It must be structured appropriately for the file system of the cluster nodes. The folder must be accessible as expressed in this string, from all cluster nodes on which MATLAB workers will run. If the value is empty, the MATLAB executable must be on the path of the worker.

See Also

Properties

`DataLocation`, `MasterName`, `MatlabCommandToRun`, `PathDependencies`

ClusterName property

Purpose

Name of Platform LSF cluster

Description

`ClusterName` indicates the name of the LSF cluster on which this scheduler will run your jobs.

Characteristics

Usage	LSF scheduler object
Read-only	Always
Data type	String

See Also

Properties

`DataLocation`, `MasterName`, `PathDependencies`

Purpose

Specify operating system of nodes on which scheduler will start workers

Description

`ClusterOsType` specifies the operating system of the nodes on which a scheduler will start workers, or whose workers are already registered with a job manager.

Characteristics

Usage	Scheduler object
Read-only	For job manager or Microsoft Windows HPC Server (or CCS) scheduler object
Data type	String

Values

The valid values for this property are `'pc'`, `'unix'`, and`'mixed'`.

- For Windows HPC Server, the setting is always `'pc'`.
- A value of `'mixed'` is valid only for distributed jobs with Platform LSF or generic schedulers; or for distributed or parallel jobs with a job manager. Otherwise, the nodes of the labs running a parallel job with LSF, Windows HPC Server, PBS Pro, TORQUE, mpiexec, or generic scheduler must all be the same platform.
- For parallel jobs with an LSF, PBS Pro, or TORQUE scheduler, this property value is set when you execute the function `setupForParallelExecution`, so you do not need to set the value directly.

See Also

Functions

`createParallelJob`, `findResource`, `setupForParallelExecution`

Properties

`ClusterName`, `MasterName`, `SchedulerHostname`

ClusterSize property

Purpose

Number of workers available to scheduler

Description

`ClusterSize` indicates the number of workers available to the scheduler for running your jobs.

Characteristics

Usage	Scheduler object
Read-only	For job manager object
Data type	Double

Values

For job managers this property is read-only. The value for a job manager represents the number of workers registered with that job manager.

For local or third-party schedulers, this property is settable, and its value specifies the maximum number of workers or labs that this scheduler can start for running a job. A parallel job's `MaximumNumberOfWorkers` property value must not exceed the value of `ClusterSize`.

Remarks

If you change the value of `ClusterVersion` or `SchedulerHostname`, this resets the values of `ClusterSize`, `JobTemplate`, and `UseSOAJobSubmission`.

See Also

Properties

`BusyWorkers`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

Purpose

Version of HPC Server scheduler

Description

`ClusterVersion` specifies which version of MicrosoftWindows HPC Server scheduler you submit your jobs to.

Characteristics

Usage	Windows HPC Server scheduler object
Read-only	Never
Data type	String

Values

This property can have the value `'CCS'` (for CCS) or `'HPCServer2008'` (for HPC Server 2008).

Remarks

If you change the value of `ClusterVersion`, this resets the values of `ClusterSize`, `JobTemplate`, and `UseSOAJobSubmission`.

See Also

Properties

`JobDescriptionFile`, `JobTemplate`, `UseSOAJobSubmission`

codistributor2dbc.defaultBlockSize property

Purpose

Default block size for codistributor2dbc distribution scheme

Description

`dbs = codistributor2dbc.defaultBlockSize` returns the default block size for a codistributor2dbc distribution scheme. Currently this returns the value 64. You can read this property only by using dot-notation; not with the `get` function on the codistributor2dbc object.

Characteristics

Usage	codistributor2dbc object
Read-only	Always
Data type	Double

See Also

Functions

`codistributor2dbc`, `codistributor2dbc.defaultLabGrid`

Properties

`BlockSize`, `LabGrid`

Purpose

Text produced by execution of task object's function

Description

CommandWindowOutput contains the text produced during the execution of a task object's Function property that would normally be printed to the MATLAB Command Window.

For example, if the function specified in the Function property makes calls to the disp command, the output that would normally be printed to the Command Window on the worker is captured in the CommandWindowOutput property.

Whether to store the CommandWindowOutput is specified using the CaptureCommandWindowOutput property. The CaptureCommandWindowOutput property by default is logical 0 to save network bandwidth in situations when the CommandWindowOutput is not needed.

Characteristics

Usage	Task object
Read-only	Always
Data type	String

Values

Before a task is evaluated, the default value of CommandWindowOutput is an empty string.

Examples

Get the Command Window output from all tasks in a job.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
createTask(j, @myfun, 1, {x});
createTask(j, @myfun, 1, {x});
.
.
alltasks = get(j, 'Tasks')
set(alltasks, 'CaptureCommandWindowOutput', true)
```

```
submit(j)
outputmessages = get(alltasks, 'CommandWindowOutput')
```

See Also

Properties

`Function`, `CaptureCommandWindowOutput`

Purpose

Information about computer on which worker is running

Description

The `Computer` property of a worker is set to the string that would be returned from running the `computer` function on that worker.

Characteristics

Usage	Worker object
Read-only	Always
Data type	String

Values

Some possible values for the `Computer` property are `GLNX86`, `GLNXA64`, `MACI`, `PCWIN`, and `PCWIN64`. For more information about specific values, see the `computer` function reference page.

See Also

Functions

`computer` MATLAB function reference page

Properties

`HostAddress`, `Hostname`, `WorkerMachineOsType`

Configuration property

Purpose

Specify configuration to apply to object or toolbox function

Description

You use the `Configuration` property to apply a configuration to an object. For details about writing and applying configurations, see "Cluster Profiles" on page 6-12.

Setting the `Configuration` property causes all the applicable properties defined in the configuration to be set on the object.

Characteristics

Usage	Scheduler, job, or task object
Read-only	Never
Data type	String

Values

The value of `Configuration` is a string that matches the name of a configuration. If a configuration was never applied to the object, or if any of the settable object properties have been changed since a configuration was applied, the `Configuration` property is set to an empty string.

Examples

Use a configuration to find a scheduler.

```
jm = findResource('scheduler','configuration','myConfig')
```

Use a configuration when creating a job object.

```
job1 = createJob(jm,'Configuration','jobmanager')
```

Apply a configuration to an existing job object.

```
job2 = createJob(jm)
set(job2,'Configuration','myjobconfig')
```

See Also

Functions

`createJob`, `createParallelJob`, `createTask`, `dfeval`, `dfevalasync`, `findResource`

Purpose

When task or job was created

Description

`CreateTime` holds a date number specifying the time when a task or job was created, in the format `'day mon dd hh:mm:ss tz yyyy'`.

Characteristics

Usage	Task object or job object
Read-only	Always
Data type	String

Values

`CreateTime` is assigned the job manager's system time when a task or job is created.

Examples

Create a job, then get its Create`Time`.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
get(j,'CreateTime')
ans =
Mon Jun 28 10:13:47 EDT 2004
```

See Also

Functions

`createJob`, `createTask`

Properties

`FinishTime`, `StartTime`, `SubmitTime`

Purpose

Job whose task this worker session is currently evaluating

Description

`CurrentJob` indicates the job whose task the worker is evaluating at the present time.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Job object

Values

`CurrentJob` is an empty vector while the worker is not evaluating a task.

See Also

Properties

`CurrentTask`, `PreviousJob`, `PreviousTask`, `Worker`

CurrentTask property

Purpose

Task that worker is currently running

Description

`CurrentTask` indicates the task that the worker is evaluating at the present time.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Task object

Values

`CurrentTask` is an empty vector while the worker is not evaluating a task.

See Also

Properties

`CurrentJob`, `PreviousJob`, `PreviousTask`, `Worker`

Purpose

Specify folder where job data is stored

Description

`DataLocation` identifies where the job data is located.

Characteristics

Usage	Scheduler object
Read-only	Never
Data type	String or struct

Values

`DataLocation` is a string or structure specifying a pathname for the job data. In a shared file system, the client, scheduler, and all worker nodes must have access to this location. In a nonshared file system, only the MATLAB client and scheduler access job data in this location.

If `DataLocation` is not set, the default location for job data is the current working directory of the MATLAB client the first time you use `findResource` to create an object for this type of scheduler. All settable property values on a scheduler object are local to the MATLAB client, and are lost when you close the client session or when you remove the object from the client workspace with `delete` or `clear all`.

Use a structure to specify the `DataLocation` in an environment of mixed platforms. The fields for the structure are named `pc` and `unix`. Each node then uses the field appropriate for its platform. See the examples below. When you examine a `DataLocation` property that was set by a structure in this way, the value returned is the string appropriate for the platform on which you are examining it.

The job data stored in the folder identified by `DataLocation` might not be compatible between different versions of MATLAB Distributed Computing Server. Therefore, `DataLocation` should not be shared by parallel computing products running different versions, and each version on your cluster should have its own `DataLocation`.

Examples

Set the `DataLocation` property for a UNIX-based cluster.

```
sch = findResource('scheduler','name','LSF')
```

```
set(sch, 'DataLocation','/depot/jobdata')
```

Use a structure to set the `DataLocation` property for a mixed platform cluster.

```
d = struct('pc',   '\\ourdomain\depot\jobdata', ...
           'unix', '/depot/jobdata')
set(sch, 'DataLocation', d)
```

See Also

Properties

`HasSharedFilesystem`, `PathDependencies`

Purpose

Specify function to run when destroying job on generic scheduler

Description

`DestroyJobFcn` specifies a function to run when you call `destroy` for a job running on a generic scheduler. This function lets you communicate with the scheduler, to provide any instructions beyond the normal toolbox action of deleting the job data from disk. To identify the job for the scheduler, the function should include a call to `getJobSchedulerData`.

For more information and examples on using these functions and properties, see “Manage Jobs with Generic Scheduler” on page 8-35.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

You can set `DestroyJobFcn` to any valid function handle, string of a function name, or a cell array whose first element is one of these.

See Also

Functions

`destroy`, `getJobSchedulerData`, `setJobSchedulerData`

Properties

`CancelJobFcn`, `CancelTaskFcn`, `DestroyTaskFcn`

DestroyTaskFcn property

Purpose

Specify function to run when destroying task on generic scheduler

Description

`DestroyTaskFcn` specifies a function to run when you call `destroy` for a task running on a generic scheduler. This function lets you communicate with the scheduler, to provide any instructions beyond the normal toolbox action of deleting the task data from disk. To identify the task for the scheduler, the function should include a call to `getJobSchedulerData`.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

You can set `DestroyTaskFcn` to any valid function handle or a cell array whose first element is a function handle.

See Also

Functions

`destroy`, `getJobSchedulerData`, `setJobSchedulerData`

Properties

`CancelJobFcn`, `CancelTaskFcn`, `DestroyJobFcn`

Purpose

Distributed dimension of codistributor1d object

Description

`dim = dist.Dimension` returns the distribution dimension of the codistributor object `dist`. The default value is the last nonsingleton dimension, indicated by a value of 0. You can read this property only by using dot-notation; not the `get` function.

Characteristics

Usage	codistributor1d object
Read-only	Always
Data type	Double

See Also

Functions

`codistributor1d`

Properties

`Partition`

EnvironmentSetMethod property

Purpose

Specify means of setting environment variables for mpiexec scheduler

Description

The mpiexec scheduler needs to supply environment variables to the MATLAB processes (labs) that it launches. There are two means by which it can do this, determined by the `EnvironmentSetMethod` property.

Characteristics

Usage	mpiexec scheduler object
Read-only	Never
Data type	String

Values

A value of `'-env'` instructs the mpiexec scheduler to insert into the mpiexec command line additional directives of the form `-env VARNAME value`.

A value of `'setenv'` instructs the mpiexec scheduler to set the environment variables in the environment that launches mpiexec.

Purpose

Task error information

Description

If an error occurs during the task evaluation, `Error` contains the `MException` object thrown. See the `MException` reference page for more about the returned information.

Characteristics

Usage	Task object
Read-only	Always
Data type	`MException` object

Values

The `Error` property is empty before an attempt to run a task.

When using a job manager, the `Error` property indicates if a worker has crashed during task execution, but when using a different scheduler `Error` does not indicate a crash.

The `Error`'s `message` and `identifier` contain the same information as the task object properties `ErrorMessage` and `ErrorIdentifier`, respectively.

Examples

Examine a task's `Error` property to determine if there was an error during execution:

```
jm = findResource('scheduler','type','local');
j = createJob(jm);
t = createTask(j, @myfun, 1, argcell);
submit(j)
waitForState(j)
has_error = ~isempty(t.Error)
```

Retrieve the error MException object from a task.

```
jm = findResource('scheduler','type','local');
j = createJob(jm);
a = [1 2 3 4]; % Note: matrix not square.
t = createTask(j, @inv, 1, {a});
```

```
submit(j)
waitForState(j)
e = get(t,'Error')
e =
  MException

  Properties:
    identifier: 'MATLAB:square'
       message: 'Matrix must be square.'
         cause: {0x1 cell}
         stack: [7x1 struct]
```

See Also

Properties

`ErrorIdentifier`, `ErrorMessage`, `Function`

Purpose

Task error identifier

Description

If an error occurs during the task evaluation, `ErrorIdentifier` contains the `identifier` property of the `MException` thrown.

Characteristics

Usage	Task object
Read-only	Always
Data type	String

Values

`ErrorIdentifier` is empty before an attempt to run a task, and remains empty if the evaluation of a task function does not produce an error or if the error did not provide an identifier. `ErrorIdentifier` has the same value as the `identifier` property of the task's `Error`.

When using a job manager, `ErrorIdentifier` indicates if a worker has crashed during task execution, but when using a different scheduler `ErrorIdentifier` does not indicate a crash.

Examples

Error Identifier Retrieval

Retrieve the error identifier from a task object.

```
jm = findResource('scheduler','type','local');
j = createJob(jm);
a = [1 2 3 4]; % Note: matrix not square.
t = createTask(j, @inv, 1, {a});
submit(j)
waitForState(j)
e_id = get(t,'ErrorIdentifer')

e_id =
MATLAB:square
```

See Also

Properties

`Error`, `ErrorMessage`, `Function`

ErrorMessage property

Purpose

Message from task error

Description

If an error occurs during the task evaluation, `ErrorMessage` contains the `message` property of the `MException` thrown.

Characteristics

Usage	Task object
Read-only	Always
Data type	String

Values

`ErrorMessage` is empty before an attempt to run a task, and remains empty if the evaluation of a task object function does not produce an error or if the error did not provide a message. `ErrorMessage` has the same value as the `message` property of the task's `Error`.

When using a job manager, `ErrorMessage` indicates if a worker has crashed during task execution, but when using a different scheduler `ErrorMessage` does not indicate a crash.

Examples

Retrieve the error message from a task object.

```
jm = findResource('scheduler','type','local');
j = createJob(jm);
a = [1 2 3 4]; % Note: matrix not square.
t = createTask(j, @inv, 1, {a});
submit(j)
waitForState(j)
e_msg = get(t,'ErrorMessage')
e_msg =
Matrix must be square.
```

See Also

Properties

`Error`, `ErrorIdentifier`, `Function`

FailedAttemptInformation property

Purpose

Information returned from failed task

Description

If a task reruns because of certain system failures, the task property `FailedAttemptInformation` stores information related to the failure and rerun attempts.

Note The `FailedAttemptInformation` property is available only when using the MathWorks job manager as your scheduler.

Characteristics

Usage	Task object
Read-only	Always
Data type	Array of objects

Values

The data type of `FailedAttemptInformation` is an array of objects, one object for each rerun of the task. The property values of each resulting object contain information about when the task was rerun and the error that caused it.

See Also

Properties

`AttemptedNumberOfRetries`, `MaximumNumberOfRetries`

Purpose

Directories and files that worker can access

Description

`FileDependencies` contains a list of directories and files that the worker will need to access for evaluating a job's tasks.

The value of the property is defined by the client session. You set the value for the property as a cell array of strings. Each string is an absolute or relative pathname to a folder or file. The toolbox makes a zip file of all the files and directories referenced in the property. (Note: If the files or directories change while they are being zipped, or if any of the directories are empty, a failure or error can result.)

The first time a worker evaluates a task for a particular job, the scheduler passes to the worker the zip file of the files and directories in the `FileDependencies` property. On the worker machine, the file is unzipped, and 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 path in the MATLAB worker session. (The subdirectories of the entries are not added to the path, even though they are included in the folder structure.) To find out where the unzipping occurs on the worker machine, use the function `getFileDependencyDir` in code that runs on the worker. See Example 2, below.

When the worker runs subsequent tasks for the same job, it uses the folder structure already set up by the job's `FileDependencies` property for the first task it ran for that job.

When you specify `FileDependencies` at the time of creating a job, the settings are combined with those specified in the applicable configuration, if any. (Setting `FileDependencies` on a job object after it is created does not combine the new setting with the configuration settings, but overwrites existing settings for that job.)

The transfer and unzipping of `FileDependencies` 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 file dependencies are deleted from the worker machine when the job is completed, or when the next job begins.

Characteristics

Usage	Job object
Read-only	After job is submitted
Data type	Cell array of strings

Values

The value of `FileDependencies` is empty by default. If a pathname that does not exist is specified for the property value, an error is generated.

Remarks

There is a default limitation on the size of data transfers via the `AttachedFiles` property. For more information on this limit, see "Object Data Size Limitations" on page 6-44. For alternative means of making data available to workers, see "Share Code" on page 8-15.

Examples

Example 1

Make available to a job's workers the contents of the directories `fd1` and `fd2`, and the file `fdfile1.m`.

```
set(job1,'FileDependencies',{'fd1' 'fd2' 'fdfile1.m'})
get(job1,'FileDependencies')
ans =
    'fd1'
    'fd2'
    'fdfile1.m'
```

Example 2

Suppose in your client MATLAB session you have the following folders on your MATLAB path:

```
dirA
dirA\subdir1
dirA\subdir2
dirB
```

Transfer the content of these folders to the worker machines, and add all these folders to the paths of the worker MATLAB sessions. On the client, execute the following code:

```
j = createJob(FileDependencies, {'dirA', 'dirB'})
% This includes the subfolders of dirA.
```

In the task function that executes on the workers, include the following code:

```
%First find where FileDependencies are unzipped:
  DepDir = getFileDependencyDir
%The top folders are already on the path, so add subfolders:
  addpath(fullfile(DepDir,'dirA','subdir1'),...
          fullfile(DepDir,'dirA','subdir2'))
```

See Also

Functions

`getFileDependencyDir`, `jobStartup`, `taskFinish`, `taskStartup`

Properties

`PathDependencies`

FinishedFcn property

Purpose

Specify callback to execute after task or job runs

Description

`FinishedFcn` specifies the function file to execute when a job or task completes its execution.

The callback executes in the local MATLAB session, that is, the session that sets the property, the MATLAB client.

Notes The `FinishedFcn` property is available only when using the MathWorks job manager as your scheduler.

The `FinishedFcn` property applies only in the client MATLAB session in which it is set. Later sessions that access the same job or task object do not inherit the setting from previous sessions.

Characteristics

Usage	Task object or job object
Read-only	Never
Data type	Function handle

Values

`FinishedFcn` can be set to any valid MATLAB callback value.

The callback follows the same model as callbacks for Handle Graphics®, passing to the callback function the object (job or task) that makes the call and an empty argument of event data.

Examples

Create a job and set its `FinishedFcn` property using a function handle to an anonymous function that sends information to the display.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm, 'Name', 'Job_52a');

set(j, 'FinishedFcn', ...
    @(job,eventdata) disp([job.Name ' ' job.State]));
```

Create a task whose FinishFcn is a function handle to a separate function.

```
createTask(j, @rand, 1, {2,4}, ...
    'FinishedFcn', @clientTaskCompleted);
```

Create the function clientTaskCompleted.m on the path of the MATLAB client.

```
function clientTaskCompleted(task,eventdata)
   disp(['Finished task: ' num2str(task.ID)])
```

Run the job and note the output messages from the job and task FinishedFcn callbacks.

```
submit(j)
Finished task: 1
Job_52a finished
```

See Also

Properties

QueuedFcn, RunningFcn

FinishTime property

Purpose When task or job finished

Description `FinishTime` holds a date number specifying the time when a task or job finished executing, in the format `'day mon dd hh:mm:ss tz yyyy'`.

If a task or job is stopped or is aborted due to an error condition, `FinishTime` will hold the time when the task or job was stopped or aborted.

Characteristics

Usage	Task object or job object
Read-only	Always
Data type	String

Values `FinishTime` is assigned the job manager's system time when the task or job has finished. If the job or task is in the failed state, its `FinishTime` property value is empty.

Examples Create and submit a job, then get its `StartTime` and `FinishTime`.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t1 = createTask(j, @rand, 1, {12,12});
t2 = createTask(j, @rand, 1, {12,12});
t3 = createTask(j, @rand, 1, {12,12});
t4 = createTask(j, @rand, 1, {12,12});
submit(j)
waitForState(j,'finished')
get(j,'StartTime')
ans =
Mon Jun 21 10:02:17 EDT 2004
get(j,'FinishTime')
ans =
Mon Jun 21 10:02:52 EDT 2004
```

See Also

Functions

`cancel`, `submit`

Properties

`CreateTime`, `StartTime`, `SubmitTime`

Function property

Purpose

Function called when evaluating task

Description

`Function` indicates the function performed in the evaluation of a task. You set the function when you create the task using `createTask`.

Characteristics

Usage	Task object
Read-only	While task is running or finished
Data type	String or function handle

See Also

Functions

`createTask`

Properties

`InputArguments`, `NumberOfOutputArguments`, `OutputArguments`

Purpose

Specify function to run when querying job state on generic scheduler

Description

`GetJobStateFcn` specifies a function to run when you call `get`, `waitForState`, or any other function that queries the state of a job running on a generic scheduler. This function lets you communicate with the scheduler, to provide any instructions beyond the normal toolbox action of retrieving the job state from disk. To identify the job for the scheduler, the function should include a call to `getJobSchedulerData`.

The value returned from the function must be a valid `State` for a job, and replaces the value ordinarily returned from the original call to `get`, etc. This might be useful when the scheduler has more up-to-date information about the state of a job than what is stored by the toolbox. For example, the scheduler might be aware of a failure before the toolbox is aware.

For more information and examples on using these functions and properties, see "Manage Jobs with Generic Scheduler" on page 8-35.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

You can set `GetJobStateFcn` to any valid function handle or a cell array whose first element is a function handle.

See Also

Functions

`get`, `getJobSchedulerData`, `setJobSchedulerData`

Properties

`State`, `SubmitFcn`

HasSharedFilesystem property

Purpose

Specify whether nodes share data location

Description

`HasSharedFilesystem` determines whether the job data stored in the location identified by the `DataLocation` property can be accessed from all nodes in the cluster. If `HasSharedFilesystem` is `false` (`0`), the scheduler handles data transfers to and from the worker nodes. If `HasSharedFilesystem` is `true` (`1`), the workers access the job data directly.

Characteristics

Usage	Scheduler object
Read-only	For Windows HPC Server scheduler object
Data type	Logical

Values

The value of `HasSharedFilesystem` can be set to `true` (or logical `1`) or `false` (or logical `0`). When you perform `get` on the property, the value returned is logical `1` or logical `0`.

See Also

Properties

`DataLocation`, `FileDependencies`, `PathDependencies`

HostAddress property

Purpose

IP address of host running job manager or worker session

Description

`HostAddress` indicates the numerical IP address of the computer running the job manager or worker session to which the job manager object or worker object refers. You can match the `HostAddress` property to find a desired job manager or worker when creating an object with `findResource`.

Characteristics

Usage	Job manager object or worker object
Read-only	Always
Data type	Cell array of strings

Examples

Create a job manager object and examine its `HostAddress` property.

```
jm = findResource('scheduler','type','jobmanager', ...
         'name','MyJobManager','LookupURL','JobMgrHost');
get(jm, 'HostAddress')
ans =
    123.123.123.123
```

See Also

Functions

`findResource`

Properties

`Computer`, `Hostname`, `WorkerMachineOsType`

Hostname property

Purpose

Name of host running job manager or worker session

Description

You can match the `Hostname` property to find a desired job manager or worker when creating the job manager or worker object with `findResource`.

Characteristics

Usage	Job manager object or worker object
Read-only	Always
Data type	String

Examples

Create a job manager object and examine its `Hostname` property.

```
jm = findResource('scheduler','type','jobmanager', ...
                                      'Name', 'MyJobManager')
get(jm, 'Hostname')
ans =
JobMgrHost
```

See Also

Functions

`findResource`

Properties

`Computer`, `HostAddress`, `WorkerMachineOsType`

Purpose

Object identifier

Description

Each object has a unique identifier within its parent object. The ID value is assigned at the time of object creation. You can use the ID property value to distinguish one object from another, such as different tasks in the same job.

Characteristics

Usage	Job object or task object
Read-only	Always
Data type	Double

Values

The first job created in a job manager has the ID value of 1, and jobs are assigned ID values in numerical sequence as they are created after that.

The first task created in a job has the ID value of 1, and tasks are assigned ID values in numerical sequence as they are created after that.

Examples

Examine the ID property of different objects.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm)
createTask(j, @rand, 1, {2,4});
createTask(j, @rand, 1, {2,4});
tasks = get(j, 'Tasks');
get(tasks, 'ID')
ans =
    [1]
    [2]
```

The ID values are the only unique properties distinguishing these two tasks.

See Also

Functions

`createJob`, `createTask`

Properties

`Jobs`, `Tasks`

Purpose

Idle workers available to run tasks

Description

The `IdleWorkers` property value indicates which workers are currently available to the job manager for the performance of job tasks.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Array of worker objects

Values

As workers complete tasks and assume new ones, the lists of workers in `BusyWorkers` and `IdleWorkers` can change rapidly. If you examine these two properties at different times, you might see the same worker on both lists if that worker has changed its status between those times.

If a worker stops unexpectedly, the job manager's knowledge of that as a busy or idle worker does not get updated until the job manager runs the next job and tries to send a task to that worker.

Examples

Examine which workers are available to a job manager for immediate use to perform tasks.

```
jm = findResource('scheduler','type','jobmanager', ...
           'name','MyJobManager','LookupURL','JobMgrHost');
get(jm, 'NumberOfIdleWorkers')
```

See Also

Properties

`BusyWorkers`, `ClusterSize`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

InputArguments property

Purpose

Input arguments to task object

Description

InputArguments is a 1-by-N cell array in which each element is an expected input argument to the task function. You specify the input arguments when you create a task with the createTask function.

Characteristics

Usage	Task object
Read-only	While task is running or finished
Data type	Cell array

Values

The forms and values of the input arguments are totally dependent on the task function.

Examples

Create a task requiring two input arguments, then examine the task's InputArguments property.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
get(t, 'InputArguments')
ans =
    [2]    [4]
```

See Also

Functions

createTask

Properties

Function, OutputArguments

Purpose

True if job manager and workers use secure communication

Description

The `IsUsingSecureCommunication` property indicates whether secure communication is being used between the job manager and the workers. The `mdce_def` file sets the parameter that controls secure communication when the mdce process starts on the cluster nodes.

Secure communication is required when running with `SecurityLevel` set to 3. It is optional at other security levels.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Boolean

See Also

Functions

`changePassword`, `clearLocalPassword`

Properties

`PromptForPassword`, `SecurityLevel`, `UserName`

JobData property

Purpose

Data made available to all workers for job's tasks

Description

The `JobData` property holds data that eventually gets stored in the local memory of the worker machines, so that it does not have to be passed to the worker for each task in a job that the worker evaluates. Passing the data only once per job to each worker is more efficient than passing data with each task.

Note, that to access the data contained in a job's `JobData` property, the worker session evaluating the task needs to have access to the job, which it gets from a call to the function `getCurrentJob`, as discussed in the example below.

Characteristics

Usage	Job object
Read-only	After job is submitted
Data type	Any type

Values

`JobData` is an empty vector by default.

Examples

Create `job1` and set its `JobData` property value to the contents of `array1`.

```
job1 = createJob(jm)
set(job1, 'JobData', array1)
createTask(job1, @myfunction, 1, {task_data})
```

Now the contents of `array1` are available to all the tasks in the job. Because the job itself must be accessible to the tasks, `myfunction` must include a call to the function `getCurrentJob`. That is, the task function `myfunction` needs to call `getCurrentJob` to get the job object through which it can get the `JobData` property. So `myfunction` should contain lines like the following:

```
cj = getCurrentJob
array1 = get(cj, 'JobData')
```

See Also

Functions

`createJob`, `createTask`

JobDescriptionFile property

Purpose

Name of XML job description file for Microsoft Windows HPC Server scheduler

Description

The XML file you specify by the `JobDescriptionFile` property defines the base state from which the job is created. The file must exist on the MATLAB path or the property must specify the full path name to the file.

Any job properties that are specified as part of MATLAB job objects (e.g., `MinimumNumberOfWorkers`, `MaximumNumberOfWorkers`, etc., for parallel or MATLAB pool jobs) override the values specified in the job description file. Scheduler properties (e.g., nonempty `JobTemplate` property) also override the values specified in the job description file.

For SOA jobs the values in the job description file are ignored.

For version 2 of Windows HPC Server 2008, the values for HPC Server job properties specified in the job description file must be compatible with the values in the job template that is applied to the job (either the default job template or the job template specified by the `JobTemplate` property). Incompatibilities between property values specified by the job description file and the job template might result in an error when you submit a job. For example, if the job template imposes property restrictions that you violate in your job description file, you get an error.

For information about job description files, consult Microsoft online documentation at:

http://technet.microsoft.com/en-us/library/cc972801(WS.10).aspx

Characteristics

Usage	Windows HPC Server scheduler object
Read-only	Never
Data type	String

See Also

Properties

`ClusterVersion`, `JobTemplate`, `UseSOAJobSubmission`

Purpose Job manager that this worker is registered with

Description `JobManager` indicates the job manager that the worker that the worker is registered with.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Job manager object

Values The value of `JobManager` is always a single job manager object.

See Also

Properties

`BusyWorkers`, `IdleWorkers`

Jobs property

Purpose

Jobs contained in job manager service or in scheduler's data location

Description

The `Jobs` property contains an array of all the job objects in a scheduler. Job objects will be in the order indicated by their `ID` property, consistent with the sequence in which they were created, regardless of their `State`. (To see the jobs categorized by state or the scheduled execution sequence for jobs in the queue, use the `findJob` function.)

Characteristics

Usage	Job manager or scheduler object
Read-only	Always
Data type	Array of job objects

Examples

Examine the `Jobs` property for a job manager, and use the resulting array of objects to set property values.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j1 = createJob(jm);
j2 = createJob(jm);
j3 = createJob(jm);
j4 = createJob(jm);
.
.
.
all_jobs = get(jm, 'Jobs')
set(all_jobs, 'MaximumNumberOfWorkers', 10);
```

The last line of code sets the `MaximumNumberOfWorkers` property value to `10` for each of the job objects in the array `all_jobs`.

See Also

Functions

`createJob`, `destroy`, `findJob`, `submit`

Properties

`Tasks`

JobTemplate property

Purpose

Name of job template for HPC Server 2008 scheduler

Description

JobTemplate identifies the name of a job template to use with your HPC Server scheduler. The property value is not case-sensitive.

With HPC Server 2008, if you do not specify a value for the `JobTemplate` property, the scheduler uses the default job template to run the job. Ask your system administrator which job template you should use.

For SOA jobs, the specified job template used for submitting SOA jobs must not impose any restrictions on the name of the job, otherwise these jobs fail.

Characteristics

Usage	Windows HPC Server scheduler object
Read-only	Never
Data type	String

Values

`JobTemplate` is an empty string by default. Job templates apply only for HPC Server 2008 clusters, and your scheduler `ClusterVersion` property must be set to `'HPCServer2008'`. If `ClusterVersion` is set to any other value, and you attempt to set `JobTemplate` to a nonempty string, an error is generated and the property value remains as a nonempty string.

Remarks

If you change the value of `ClusterVersion` or `SchedulerHostname`, this resets the values of `ClusterSize`, `JobTemplate`, and `UseSOAJobSubmission`.

See Also

Properties

`ClusterVersion`, `JobDescriptionFile`, `UseSOAJobSubmission`

Purpose

Lab grid of codistributor2dbc object

Description

`lbgrd = dist.LabGrid` returns the lab grid associated with a codistributor2dbc object `dist`. The lab grid is the row vector of length 2, `[nprow, npcol]`, used by the ScaLAPACK library to represent the `nprow`-by-`npcol` layout of the labs for array distribution. `nprow` times `npcol` must equal `numlabs`.

For more information on 2dbc distribution and lab grids of distributed arrays, see “2-Dimensional Distribution” on page 5-17.

Characteristics

Usage	codistributor2dbc object
Read-only	Always
Data type	Array of doubles

See Also

Functions

`codistributor2dbc`, `numlabs`

Properties

`BlockSize`, `Orientation`

MasterName property

Purpose	Name of Platform LSF master node
Description	`MasterName` indicates the name of the LSF cluster master node.

Characteristics

Usage	LSF scheduler object
Read-only	Always
Data type	String

Values `MasterName` is a string of the full name of the master node.

See Also

Properties

`ClusterName`

Purpose

MATLAB command that generic scheduler runs to start lab

Description

`MatlabCommandToRun` indicates the command that the scheduler uses to start a MATLAB worker on a cluster node for a task evaluation. To ensure that the correct MATLAB runs, your scheduler script can construct a path to the executable by concatenating the values of `ClusterMatlabRoot` and `MatlabCommandToRun` into a single string.

Characteristics

Usage	Generic scheduler object
Read-only	Always
Data type	String

Values

`MatlabCommandToRun` is set by the toolbox when the scheduler object is created.

See Also

Properties

`ClusterMatlabRoot`, `SubmitFcn`

MaximumNumberOfRetries property

Purpose

Specify maximum number of times to rerun failed task

Description

If a task cannot complete because of certain system failures, the job manager can attempt to rerun the task. `MaximumNumberOfRetries` specifies how many times to try to run the task after such failures. The task reruns until it succeeds or until it reaches the specified maximum number of attempts.

Note The `MaximumNumberOfRetries` property is available only when using the MathWorks job manager as your scheduler.

Characteristics

Usage	Task object
Read-only	Never
Data type	Double

Values

The default value for `MaximumNumberOfRetries` is 1.

See Also

Properties

`AttemptedNumberOfRetries`, `FailedAttemptInformation`

MaximumNumberOfWorkers property

Purpose

Specify maximum number of workers to perform job tasks

Description

With `MaximumNumberOfWorkers` you specify the greatest number of workers to be used to perform the evaluation of the job's tasks at any one time. Tasks may be distributed to different workers at different times during execution of the job, so that more than `MaximumNumberOfWorkers` might be used for the whole job, but this property limits the portion of the cluster used for the job at any one time.

Characteristics

Usage	Job object
Read-only	After job is submitted
Data type	Double

Values

You can set the value to anything equal to or greater than the value of the `MinimumNumberOfWorkers` property.

Examples

Set the maximum number of workers to perform a job.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
set(j, 'MaximumNumberOfWorkers', 12);
```

In this example, the job will use no more than 12 workers, regardless of how many tasks are in the job and how many workers are available on the cluster.

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

MinimumNumberOfWorkers property

Purpose

Specify minimum number of workers to perform job tasks

Description

With `MinimumNumberOfWorkers` you specify the minimum number of workers to perform the evaluation of the job's tasks. When the job is queued, it will not run until at least this many workers are simultaneously available.

If `MinimumNumberOfWorkers` workers are available to the job manager, but some of the task dispatches fail due to network or node failures, such that the number of tasks actually dispatched is less than `MinimumNumberOfWorkers`, the job will be canceled.

Characteristics

Usage	Job object
Read-only	After job is submitted
Data type	Double

Values

The default value is `1`. You can set the value anywhere from `1` up to or equal to the value of the `MaximumNumberOfWorkers` property.

Examples

Set the minimum number of workers to perform a job.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
set(j, 'MinimumNumberOfWorkers', 6);
```

In this example, when the job is queued, it will not begin running tasks until at least six workers are available to perform task evaluations.

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

Purpose Specify pathname of executable mpiexec command

Description `MpiexecFileName` specifies which `mpiexec` command is executed to run your jobs.

Characteristics

Usage	mpiexec scheduler object
Read-only	Never
Data type	String

Remarks See your network administrator to find out which `mpiexec` you should run. The default value of the property points the `mpiexec` included in your MATLAB installation.

See Also

Functions

`mpiLibConf`, `mpiSettings`

Properties

`SubmitArguments`

Name property

Purpose

Name of job manager, job, or worker object

Description

The descriptive name of a job manager or worker is set when its service is started, as described in "Customizing Engine Services" in the MATLAB Distributed Computing Server System Administrator's Guide. This is reflected in the `Name` property of the object that represents the service. You can use the name of the job manager or worker service to search for the particular service when creating an object with the `findResource` function.

You can configure `Name` as a descriptive name for a job object at any time before the job is submitted to the queue.

Characteristics

Usage	Job manager object, job object, or worker object
Read-only	Always for a job manager or worker object; after job object is submitted
Data type	String

Values

By default, a job object is constructed with a `Name` created by concatenating the `Name` of the job manager with `_job`.

Examples

Construct a job manager object by searching for the name of the service you want to use.

```
jm = findResource('scheduler','type','jobmanager', ...
          'Name','MyJobManager');
```

Construct a job and note its default `Name`.

```
j = createJob(jm);
get(j, 'Name')
ans =
    MyJobManager_job
```

Change the job's `Name` property and verify the new setting.

```
set(j,'Name','MyJob')
get(j,'Name')
ans =
    MyJob
```

See Also

Functions

`findResource`, `createJob`

NumberOfBusyWorkers property

Purpose

Number of workers currently running tasks

Description

The `NumberOfBusyWorkers` property value indicates how many workers are currently running tasks for the job manager.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Double

Values

The value of `NumberOfBusyWorkers` can range from 0 up to the total number of workers registered with the job manager.

Examples

Examine the number of workers currently running tasks for a job manager.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
get(jm, 'NumberOfBusyWorkers')
```

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfIdleWorkers`

Purpose

Number of idle workers available to run tasks

Description

The `NumberOfIdleWorkers` property value indicates how many workers are currently available to the job manager for the performance of job tasks.

If the `NumberOfIdleWorkers` is equal to or greater than the `MinimumNumberOfWorkers` of the job at the top of the queue, that job can start running.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Double

Values

The value of `NumberOfIdleWorkers` can range from 0 up to the total number of workers registered with the job manager.

Examples

Examine the number of workers available to a job manager.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
get(jm, 'NumberOfIdleWorkers')
```

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`

NumberOfOutputArguments property

Purpose

Number of arguments returned by task function

Description

When you create a task with the `createTask` function, you define how many output arguments are expected from the task function.

Characteristics

Usage	Task object
Read-only	While task is running or finished
Data type	Double

Values

A matrix is considered one argument.

Examples

Create a task and examine its `NumberOfOutputArguments` property.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
get(t,'NumberOfOutputArguments')
ans =
    1
```

This example returns a 2-by-4 matrix, which is a single argument. The `NumberOfOutputArguments` value is set by the `createTask` function, as the argument immediately after the task function definition; in this case, the `1` following the `@rand` argument.

See Also

Functions

`createTask`

Properties

`OutputArguments`

Purpose

Orientation of codistributor2dbc object

Description

`DIST.Orientation` returns the orientation associated with the `LabGrid` of the codistributor2dbc object `DIST`. This orientation refers to how the labs are organized within the lab grid. Supported orientation values are `'row'` and `'col'`. You can read this property only by using dot-notation; not the `get` function.

For more information on 2dbc distribution of arrays, see "2-Dimensional Distribution" on page 5-17.

Characteristics

Usage	codistributor2dbc object
Read-only	Always
Data type	String

See Also

Functions

`codistributor2dbc`

Properties

`BlockSize`, `LabGrid`

OutputArguments property

Purpose

Data returned from execution of task

Description

`OutputArguments` is a 1-by-N cell array in which each element corresponds to each output argument requested from task evaluation. If the task's `NumberOfOutputArguments` property value is `0`, or if the evaluation of the task produced an error, `the` cell array is empty.

Characteristics

Usage	Task object
Read-only	Always
Data type	Cell array

Values

The forms and values of the output arguments are totally dependent on the task function.

Examples

Create a job with a task and examine its result after running the job.

```
jm = findResource('scheduler','type','jobmanager', ...
         'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
submit(j)
```

When the job is finished, retrieve the results as a cell array.

```
result = get(t, 'OutputArguments')
```

Retrieve the results from all the tasks of a job.

```
alltasks = get(j, 'Tasks')
allresults = get(alltasks, 'OutputArguments')
```

Because each task returns a cell array, `allresults` is a cell array of cell arrays.

See Also

Functions

`createTask`, `getAllOutputArguments`

Properties

`Function`, `InputArguments`, `NumberOfOutputArguments`

ParallelSubmissionWrapperScript property

Purpose

Script that scheduler runs to start labs

Description

`ParallelSubmissionWrapperScript` identifies the script for the LSF, PBS Pro, or TORQUE scheduler to run when starting labs for a parallel job.

Characteristics

Usage	LSF, PBS Pro, or TORQUE scheduler object
Read-only	Never
Data type	String

Values

`ParallelSubmissionWrapperScript` is a string specifying the full path to the script. This property value is set when you execute the function `setupForParallelExecution`, so you do not need to set the value directly. The property value then points to the appropriate wrapper script in `*matlabroot*/toolbox/distcomp/bin/util`.

See Also

Functions

`createParallelJob`, `setupForParallelExecution`, `submit`

Properties

`ClusterName`, `ClusterMatlabRoot`, `MasterName`, `SubmitArguments`

Purpose

Specify function to run when parallel job submitted to generic scheduler

Description

`ParallelSubmitFcn` identifies the function to run when you submit a parallel job to the generic scheduler. The function runs in the MATLAB client. This user-defined parallel submit function provides certain job and task data for the MATLAB worker, and identifies a corresponding decode function for the MATLAB worker to run.

For more information, see "MATLAB Client Submit Function" on page 8-22.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

`ParallelSubmitFcn` can be set to any valid MATLAB callback value that uses the user-defined parallel submit function.

For more information about parallel submit functions and where to find example templates you can use, see "Use the Generic Scheduler Interface" on page 9-8.

See Also

Functions

`createParallelJob`, `submit`

Properties

`MatlabCommandToRun`, `SubmitFcn`

Parent property

Purpose

Parent object of job or task

Description

A job's `Parent` property indicates the job manager or scheduler object that contains the job. A task's `Parent` property indicates the job object that contains the task.

Characteristics

Usage	Job object or task object
Read-only	Always
Data type	Job manager, scheduler, or job object

See Also

Properties

`Jobs`, `Tasks`

Purpose

Partition scheme of codistributor1d object

Description

`par = dist.Partition` returns the partition scheme of the codistributor1d object `dist`, describing how the object would distribute an array among the labs. You can read this property only by using dot-notation; not the `get` function.

Characteristics

Usage	codistributor1d object
Read-only	Always
Data type	Array of doubles

Examples

```
dist = codistributor1d(2, [3 3 2 2])
dist.Partition
```

returns `[3 3 2 2]` .

See Also

Functions

`codistributor1d`

Properties

`Dimension`

PathDependencies property

Purpose

Specify directories to add to MATLAB worker path

Description

`PathDependencies` identifies directories to be added to the top of the path of MATLAB worker sessions for this job. If `FileDependencies` are also used, `FileDependencies` are above `PathDependencies` on the worker's path.

When you specify `PathDependencies` at the time of creating a job, the settings are combined with those specified in the applicable configuration, if any. (Setting `PathDependencies` on a job object after it is created does not combine the new setting with the configuration settings, but overwrites existing settings for that job.)

Characteristics

Usage	Scheduler job object
Read-only	Never
Data type	Cell array of strings

Values

`PathDependencies` is empty by default. For a mixed-platform environment, the strings can specify both UNIX-based and Microsoft Windows-based paths; those not appropriate or not found for a particular node generate warnings and are ignored.

Remarks

For alternative means of making data available to workers, see "Share Code" on page 8-15.

Examples

Set 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`).

```
sch = findResource('scheduler','name','LSF')
job1 = createJob(sch)
p = {'/central/funcs','/dept1/funcs', ...
     '\\OurDomain\central\funcs','\\OurDomain\dept1\funcs'}
set(job1, 'PathDependencies', p)
```

See Also

Properties

`ClusterMatlabRoot`, `FileDependencies`

PreviousJob property

Purpose

Job whose task this worker previously ran

Description

`PreviousJob` indicates the job whose task the worker most recently evaluated.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Job object

Values

`PreviousJob` is an empty vector until the worker finishes evaluating its first task.

See Also

Properties

`CurrentJob`, `CurrentTask`, `PreviousTask`, `Worker`

Purpose

Task that this worker previously ran

Description

`PreviousTask` indicates the task that the worker most recently evaluated.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Task object

Values

`PreviousTask` is an empty vector until the worker finishes evaluating its first task.

See Also

Properties

`CurrentJob`, `CurrentTask`, `PreviousJob`, `Worker`

PromptForPassword property

Purpose

Specify if system should prompt for password when authenticating user

Description

The `PromptForPassword` property is `true` by default, so that when you access a job manager object, if you do not already have a password stored, the system prompts you to enter it.

Setting `PromptForPassword` to `false` causes the system to generate an error when a password is required. This can be useful when you have a noninteractive script or function that programmatically accesses the job manager, and you might prefer an error rather than a password prompt.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Boolean

See Also

Functions

`changePassword`, `clearLocalPassword`

Properties

`IsUsingSecureCommunication`, `SecurityLevel`, `UserName`

Purpose

Specify function file to execute when job is submitted to job manager queue

Description

`QueuedFcn` specifies the function file to execute when a job is submitted to a job manager queue.

The callback executes in the local MATLAB session, that is, the session that sets the property.

Notes The `QueuedFcn` property is available only when using the MathWorks job manager as your scheduler.

The `QueuedFcn` property applies only in the client MATLAB session in which it is set. Later sessions that access the same job object do not inherit the setting from previous sessions.

Characteristics

Usage	Job object
Read-only	Never
Data type	Function handle

Values

`QueuedFcn` can be set to any valid MATLAB callback value.

Examples

Create a job and set its `QueuedFcn` property, using a function handle to an anonymous function that sends information to the display.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm, 'Name', 'Job_52a');
set(j, 'QueuedFcn', ...
  @(job,eventdata) disp([job.Name ' now queued for execution.']))
.
.
.
```

```
submit(j)
Job_52a now queued for execution.
```

See Also

Functions

`submit`

Properties

`FinishedFcn`, `RunningFcn`

Purpose

Command to copy files from client

Description

When using a nonshared file system, the command specified by this property's value is used on the cluster to copy files from the client machine. The syntax of the command must be compatible with standard `rcp`. On MicrosoftWindows operating systems, the cluster machines must have a suitable installation of `rcp`.

Characteristics

Usage	PBS Pro or TORQUE scheduler object
Read-only	Never
Data type	String

ResourceTemplate property

Purpose

Resource definition for PBS Pro or TORQUE scheduler

Description

The value of this property is used to build the resource selection portion of the `qsub` command, generally identified by the `-l` flag. The toolbox uses this to identify the number of tasks in a parallel job, and you might want to fill out other selection subclauses (such as the OS type of the workers). You should specify a value for this property that includes the literal string `^N^` , which the toolbox will replace with the number of workers in the parallel job prior to submission.

Characteristics

Usage	PBS Pro or TORQUE scheduler object
Read-only	Never
Data type	String

Values

You might set the property value as follows, to accommodate your cluster size and to set the “wall time” limit of the job (i.e., how long it is allowed to run in real time) to one hour:

- `'-l select=^N^,walltime=1:00:00'` (for a PBS Pro scheduler)
- `'-l nodes=^N^,walltime=1:00:00'` (for a TORQUE scheduler)

Purpose

Specify whether to restart MATLAB workers before evaluating job tasks

Description

In some cases, you might want to restart MATLAB on the workers before they evaluate any tasks in a job. This action resets defaults, clears the workspace, frees available memory, and so on.

Characteristics

Usage	Job object
Read-only	After job is submitted
Data type	Logical

Values

Set `RestartWorker` to `true` (or logical `1`) if you want the job to restart the MATLAB session on any workers before they evaluate their first task for that job. The workers are not reset between tasks of the same job. Set `RestartWorker` to `false` (or logical `0`) if you do not want MATLAB restarted on any workers. When you perform `get` on the property, the value returned is logical 1 or logical 0. The default value is `0`, which does not restart the workers.

Examples

Create a job and set it so that MATLAB workers are restarted before evaluating tasks in a job.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
set(j, 'RestartWorker', true)
.
.
.
submit(j)
```

See Also

Functions

`submit`

RshCommand property

Purpose

Remote execution command used on worker nodes during parallel job

Description

Used on only UNIX operating systems, the value of this property is the command used at the beginning of running parallel jobs, typically to start MPI daemon processes on the nodes allocated to run MATLAB workers. The remote execution must be able to proceed without user interaction, for example, without prompting for user credentials.

Characteristics

Usage	PBS Pro or TORQUE scheduler object
Read-only	Never
Data type	String

Purpose

Specify function file to execute when job or task starts running

Description

`RunningFcn` specifies the function file to execute when a job or task begins its execution.

The callback executes in the local MATLAB client session, that is, the session that sets the property.

Notes The `RunningFcn` property is available only when using the MathWorks job manager as your scheduler.

The `RunningFcn` property applies only in the client MATLAB session in which it is set. Later sessions that access the same job or task object do not inherit the setting from previous sessions.

Characteristics

Usage	Task object or job object
Read-only	Never
Data type	Function handle

Values

`RunningFcn` can be set to any valid MATLAB callback value.

Examples

Create a job and set its `QueuedFcn` property, using a function handle to an anonymous function that sends information to the display.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm, 'Name', 'Job_52a');
set(j, 'RunningFcn', ...
    @(job,eventdata) disp([job.Name ' now running.']))
.
.
.
submit(j)
```

```
Job_52a now running.
```

See Also

Functions

`submit`

Properties

`FinishedFcn`, `QueuedFcn`

Purpose

Name of host running Microsoft Windows HPC Server scheduler

Description

`SchedulerHostname` indicates the name of the node on which the Windows HPC Server (or CCS) scheduler is running.

Characteristics

Usage	Windows HPC Server scheduler object
Read-only	Never
Data type	String

Values

`SchedulerHostname` is a string of the full name of the scheduler node.

Remarks

If you change the value of `SchedulerHostname`, this resets the values of `ClusterSize`, `JobTemplate`, and `UseSOAJobSubmission`.

See Also

Properties

`ClusterOsType`

SecurityLevel property

Purpose

Security level controlling access to job manager and its jobs

Description

The `SecurityLevel` property indicates the degree of security applied to the job manager and its jobs. The `mdce_def` file sets the parameter that controls security level when the mdce process starts on the cluster nodes.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Double

Values

The property values indicating security level and their effects are shown in the following table.

Security Level	Effect
0	No security. All users can access all jobs; the `AuthorizedUsers` property of the job is ignored.
1	You are warned when you try to access other users' jobs and tasks, but can still perform all actions. You can suppress the warning by adding your user name to the `AuthorizedUsers` property of the job.

Security Level	Effect
2	Authentication required. You must enter a password to access any jobs and tasks. You cannot access other users' jobs unless your user name is included in the job's `AuthorizedUsers` property.
3	Same as level 2, but in addition, tasks run on the workers as the user to whom the job belongs. The user name and password for authentication in the client session need to be the same as the system password used to log on to a worker machine. NOTE: This level requires secure communication between job manager and workers. Secure communication is also set in the `mdce_def` file, and is indicated by a job manager's `IsUsingSecureCommunication` property.

The job manager and the workers should run at the same security level. A worker running at too low a security level will fail to register with the job manager, because the job manager does not trust it.

See Also

Functions

`changePassword`, `clearLocalPassword`

Properties

`AuthorizedUsers`, `IsUsingSecureCommunication`, `PromptForPassword`, `UserName`

ServerName property

Purpose

Name of current PBS Pro or TORQUE server machine

Description

`ServerName` indicates the name of the node on which the PBS Pro or TORQUE scheduler is running.

Characteristics

Usage	PBS Pro or TORQUE scheduler object
Read-only	Always
Data type	String

See Also

Properties

`ClusterOsType`

StartTime property

Purpose

When job or task started

Description

`StartTime` holds a date number specifying the time when a job or task starts running, in the format `'day mon dd hh:mm:ss tz yyyy'`.

Characteristics

Usage	Job object or task object
Read-only	Always
Data type	String

Values

`StartTime` is assigned the job manager's system time when the task or job has started running.

Examples

Create and submit a job, then get its `StartTime` and `FinishTime`.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t1 = createTask(j, @rand, 1, {12,12});
t2 = createTask(j, @rand, 1, {12,12});
t3 = createTask(j, @rand, 1, {12,12});
t4 = createTask(j, @rand, 1, {12,12});
submit(j)
waitForState(j, 'finished')
get(j, 'StartTime')
ans =
Mon Jun 21 10:02:17 EDT 2004
get(j, 'FinishTime')
ans =
Mon Jun 21 10:02:52 EDT 2004
```

See Also

Functions

`submit`

Properties

`CreateTime`, `FinishTime`, `SubmitTime`

Purpose

Current state of task, job, job manager, or worker

Description

The `State` property reflects the stage of an object in its life cycle, indicating primarily whether or not it has yet been executed. The possible `State` values for all Parallel Computing Toolbox objects are discussed below in the “Values” section.

Note The `State` property of the task object is different than the `State` property of the job object. For example, a task that is finished may be part of a job that is running if other tasks in the job have not finished.

Characteristics

Usage	Task, job, job manager, or worker object
Read-only	Always
Data type	String

Values

Task Object

For a task object, possible values for `State` are

- `pending` — Tasks that have not yet started to evaluate the task object’s `Function` property are in the `pending` state.
- `running` — Task objects that are currently in the process of evaluating the `Function` property are in the `running` state.
- `finished` — Task objects that have finished evaluating the task object’s `Function` property are in the `finished` state.
- `unavailable` — Communication cannot be established with the job manager.

Job Object

For a job object, possible values for `State` are

- `pending` — Job objects that have not yet been submitted to a job queue are in the `pending` state.
- `queued` — Job objects that have been submitted to a job queue but have not yet started to run are in the `queued` state.
- `running` — Job objects that are currently in the process of running are in the `running` state.
- `finished` — Job objects that have completed running all their tasks are in the `finished` state.
- `failed` — Job objects when using a third-party scheduler and the job could not run because of unexpected or missing information.
- `destroyed` — Job objects whose data has been permanently removed from the data location or job manager.
- `unavailable` — Communication cannot be established with the job manager.

Job Manager

For a job manager, possible values for `State` are

- `running` — A started job queue will execute jobs normally.
- `paused` — The job queue is paused.
- `unavailable` — Communication cannot be established with the job manager.

When a job manager first starts up, the default value for `State` is `running`.

Worker

For a worker, possible values for `State` are

- `running` — A started job queue will execute jobs normally.
- `unavailable` — Communication cannot be established with the worker.

Examples

Create a job manager object representing a job manager service, and create a job object; then examine each object's `State` property.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
get(jm, 'State')
ans =
    running
j = createJob(jm);
get(j, 'State')
ans =
    pending
```

See Also

Functions

`createJob`, `createTask`, `findResource`, `pause`, `resume`, `submit`

SubmitArguments property

Purpose

Specify additional arguments to use when submitting job to Platform LSF, PBS Pro, TORQUE, or mpiexec scheduler

Description

`SubmitArguments` is simply a string that is passed via the `bsub` or `qsub` command to the LSF, PBS Pro, or TORQUE scheduler at submit time, or passed to the `mpiexec` command if using an mpiexec scheduler.

Characteristics

Usage	LSF, PBS Pro, TORQUE, or mpiexec scheduler object
Read-only	Never
Data type	String

Values

LSF Scheduler

Useful `SubmitArguments` values might be `'-m "machine1 machine2"'` to indicate that your scheduler should use only the named machines to run the job, or `'-R "type==LINUX64"'` to use only workers running on 64-bit machines with a Linux operating system. Note that by default the scheduler will attempt to run your job on only nodes with an architecture similar to the local machine's unless you specify `'-R "type==any"'`.

PBS Pro or TORQUE Scheduler

A value of `'-q queuename'` submits the job to the queue specified by `queuename`. A value of `'-p 10'` runs the job at priority level 10.

mpiexec Scheduler

The following `SubmitArguments` values might be useful when using an mpiexec scheduler. They can be combined to form a single string when separated by spaces.

Value	Description
`-phrase MATLAB`	Use `MATLAB` as passphrase to connect with smpd.
`-noprompt`	Suppress prompting for any user information.
`-localonly`	Run only on the local computer.
`-host <hostname>`	Run only on the identified host.
`-machinefile <filename>`	Run only on the nodes listed in the specified file (one hostname per line).

For a complete list, see the command-line help for the `mpiexec` command:

```
mpiexec -help
mpiexec -help2
```

See Also

Functions

`submit`

Properties

`MatlabCommandToRun`, `MpiexecFileName`

SubmitFcn property

Purpose

Specify function to run when job submitted to generic scheduler

Description

`SubmitFcn` identifies the function to run when you submit a job to the generic scheduler. The function runs in the MATLAB client. This user-defined submit function provides certain job and task data for the MATLAB worker, and identifies a corresponding decode function for the MATLAB worker to run.

For further information, see "MATLAB Client Submit Function" on page 8-22.

Characteristics

Usage	Generic scheduler object
Read-only	Never
Data type	Function handle

Values

`SubmitFcn` can be set to any valid MATLAB callback value that uses the user-defined submit function.

For a description of the user-defined submit function, how it is used, and its relationship to the worker decode function, see "Use the Generic Scheduler Interface" on page 8-21.

See Also

Functions

`submit`

Properties

`MatlabCommandToRun`

Purpose

When job was submitted to queue

Description

`SubmitTime` holds a date number specifying the time when a job was submitted to the job queue, in the format `'day mon dd hh:mm:ss tz yyyy'`.

Characteristics

Usage	Job object
Read-only	Always
Data type	String

Values

`SubmitTime` is assigned the job manager's system time when the job is submitted.

Examples

Create and submit a job, then get its `SubmitTime`.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
createTask(j, @rand, 1, {12,12});
submit(j)
get(j, 'SubmitTime')
ans =
Wed Jun 30 11:33:21 EDT 2004
```

See Also

Functions

`submit`

Properties

`CreateTime`, `FinishTime`, `StartTime`

Tag property

Purpose

Specify label to associate with job object

Description

You configure `Tag` to be a string value that uniquely identifies a job object.

`Tag` is particularly useful in programs that would otherwise need to define the job object as a global variable, or pass the object as an argument between callback routines.

You can return the job object with the `findJob` function by specifying the `Tag` property value.

Characteristics

Usage	Job object
Read-only	Never
Data type	String

Values

The default value is an empty string.

Examples

Suppose you create a job object in the job manager `jm`.

```
job1 = createJob(jm);
```

You can assign `job1` a unique label using `Tag`.

```
set(job1,'Tag','MyFirstJob')
```

You can identify and access `job1` using the `findJob` function and the `Tag` property value.

```
job_one = findJob(jm,'Tag','MyFirstJob');
```

See Also

Functions

`findJob`

Purpose

First task contained in MATLAB pool job object

Description

The `Task` property contains the task object for the MATLAB pool job, which has only this one task. This is the same as the first task contained in the `Tasks` property.

Characteristics

Usage	MATLAB pool job object
Read-only	Always
Data type	Task object

See Also

Functions

`createMatlabPoolJob`, `createTask`

Properties

`Tasks`

Tasks property

Purpose

Tasks contained in job object

Description

The Tasks property contains an array of all the task objects in a job, whether the tasks are pending, running, or finished. Tasks are always returned in the order in which they were created.

Characteristics

Usage	Job object
Read-only	Always
Data type	Array of task objects

Examples

Examine the Tasks property for a job object, and use the resulting array of objects to set property values.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
createTask(j, ...)
.
.
.
createTask(j, ...)
alltasks = get(j, 'Tasks')
alltasks =
     distcomp.task: 10-by-1
set(alltasks, 'Timeout', 20);
```

The last line of code sets the Timeout property value to 20 seconds for each task in the job.

See Also

Functions

`createTask`, `destroy`, `findTask`

Properties

`Jobs`

Purpose

Specify time limit to complete task or job

Description

Timeout holds a double value specifying the number of seconds to wait before giving up on a task or job.

The time for timeout begins counting when the task `State` property value changes from the `Pending` to `Running`, or when the job object `State` property value changes from `Queued` to `Running`.

When a task times out, the behavior of the task is the same as if the task were stopped with the `cancel` function, except a different message is placed in the task object's `ErrorMessage` property.

When a job times out, the behavior of the job is the same as if the job were stopped using the `cancel` function, except all pending and running tasks are treated as having timed out.

Characteristics

Usage	Task object or job object
Read-only	While running
Data type	Double

Values

The default value for `Timeout` is large enough so that in practice, tasks and jobs will never time out. You should set the value of `Timeout` to the number of seconds you want to allow for completion of tasks and jobs.

Examples

Set a job's `Timeout` value to 1 minute.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
set(j, 'Timeout', 60)
```

See Also

Functions

`submit`

Properties

`ErrorMessage`, `State`

Type property

Purpose

Type of scheduler object

Description

`Type` indicates the type of scheduler object.

Characteristics

Usage	Scheduler object
Read-only	Always
Data type	String

Values

`Type` is a string indicating the type of scheduler represented by this object.

Purpose

Specify data to associate with object

Description

You configure `UserData` to store data that you want to associate with an object. The object does not use this data directly, but you can access it using the `get` function or dot notation.

`UserData` is stored in the local MATLAB client session, not in the job manager, job data location, or worker. So, one MATLAB client session cannot access the data stored in this property by another MATLAB client session. Even on the same machine, if you close the client session where `UserData` is set for an object, and then access the same object from a later client session via the job manager or job data location, the original `UserData` is not recovered. Likewise, commands such as

```
clear all
clear functions
```

will clear an object in the local session, permanently removing the data in the `UserData` property.

Characteristics

Usage	Scheduler object, job object, or task object
Read-only	Never
Data type	Any type

Values

The default value is an empty vector.

Examples

Suppose you create the job object `job1`.

```
job1 = createJob(jm);
```

You can associate data with `job1` by storing it in `UserData`.

```
coeff.a = 1.0;
coeff.b = -1.25;
job1.UserData = coeff
```

```
get(job1,'UserData')
ans =
    a: 1
    b: -1.2500
```

Purpose

User who created job or job manager object

Description

On a job, the `UserName` property value is a string indicating the login name of the user who created the job.

On a job manager object, the `UserName` property value indicates the user who created the object or who is using the job manager object to access jobs in its queue.

Characteristics

Usage	Job object or job manager object
Read-only	Always for job object
	Never for job manager object, but can be password protected
Data type	String

Examples

Examine a job to see who created it.

```
get(job1, 'UserName')
ans =
jsmith
```

Change the user for a job manager object in your current MATLAB session. Certain security levels display a password prompt.

```
jm = findResource('scheduler','type','jobmanager','name','central-jm');
set(jm, 'UserName', 'MyNewName')
```

See Also

These references apply to using the `UserName` property for job manager objects.

Functions

`changePassword`, `clearLocalPassword`

Properties

`IsUsingSecureCommunication`, `PromptForPassword`, `SecurityLevel`

UseSOAJobSubmission property

Purpose

Allow service-oriented architecture (SOA) submission on HPC Server 2008 cluster

Description

The value you assign to the `UseSOAJobSubmission` property specifies whether to allow SOA job submissions for the scheduler object representing a Microsoft Windows HPC Server 2008 cluster. If you enable SOA submission, MATLAB worker sessions can each evaluate multiple tasks in succession. If you disable SOA submission, a separate MATLAB worker starts for each task.

Ensure that HPC Server 2008 is correctly configured to run SOA jobs on MATLAB Distributed Computing Server. For more details, see the online installation instructions at `http://www.mathworks.com/distconfig`.

Note The MATLAB client from which you submit SOA jobs to the HPC Server 2008 scheduler must remain open for the duration of these jobs. Closing the MATLAB client session while SOA jobs are in the pending, queued, or running state causes the scheduler to cancel these jobs.

Characteristics

Usage	Windows HPC Server scheduler object
Read-only	Never
Data type	Logical

Values

`UseSOAJobSubmission` is `false` by default. SOA job submission works only for HPC Server 2008 clusters, and your scheduler `ClusterVersion` property must be set to `'HPCServer2008'`. If `ClusterVersion` is set to any other value, and you attempt to set `UseSOAJobSubmission` to `true`, an error is generated and the property value remains `false`.

Remarks

If you change the value of ClusterVersion or SchedulerHostname, this resets the values of ClusterSize, JobTemplate, and UseSOAJobSubmission.

Examples

Set the scheduler to allow SOA job submissions.

```
s = findResource('scheduler', 'type', 'hpcserver');
s.UseSOAJobSubmission = true;
```

See Also

Properties

ClusterVersion, JobDescriptionFile, JobTemplate,

Worker property

Purpose

Worker session that performed task

Description

The `Worker` property value is an object representing the worker session that evaluated the task.

Characteristics

Usage	Task object
Read-only	Always
Data type	Worker object

Values

Before a task is evaluated, its `Worker` property value is an empty vector.

Examples

Find out which worker evaluated a particular task.

```
submit(job1)
waitForState(job1,'finished')
t1 = findTask(job1,'ID',1)
t1.Worker.Name
ans =
node55_worker1
```

See Also

Properties

`Tasks`

Purpose

Specify operating system of nodes on which mpiexec scheduler will start labs

Description

`WorkerMachineOsType` specifies the operating system of the nodes that an mpiexec scheduler will start labs on for the running of a parallel job.

Characteristics

Usage	mpiexec scheduler object
Read-only	Never
Data type	String

Values

The only value the property can have is `'pc'` or `'unix'`. The nodes of the labs running an mpiexec job must all be the same platform. The only heterogeneous mixing allowed in the cluster for the same mpiexec job is Intel® Macintosh-based systems with 32-bit Linux-based systems.

See Also

Properties

`Computer`, `HostAddress`, `Hostname`

Glossary

CHECKPOINTBASE
: The name of the parameter in the `mdce_def` file that defines the location of the job manager and worker checkpoint directories.

checkpoint directory
: Location where job manager checkpoint information and worker checkpoint information is stored.

client
: The MATLAB session that defines and submits the job. This is the MATLAB session in which the programmer usually develops and prototypes applications. Also known as the MATLAB client.

client computer
: The computer running the MATLAB client.

cluster
: A collection of computers that are connected via a network and intended for a common purpose.

coarse-grained application
: An application for which run time is significantly greater than the communication time needed to start and stop the program. Coarse-grained distributed applications are also called embarrassingly parallel applications.

codistributed array
: An array partitioned into segments, with each segment residing in the workspace of a different lab.

communicating job
: Job composed of tasks that communicate with each other during evaluation. All tasks must run simultaneously. A special case of communicating job is a MATLAB pool, used for executing `parfor`-loops and `spmd` blocks.

Composite

An object in a MATLAB client session that provides access to data values stored on the labs in a MATLAB pool, such as the values of variables that are assigned inside an `spmd` statement.

computer

A system with one or more processors.

distributed application

The same application that runs independently on several nodes, possibly with different input parameters. There is no communication, shared data, or synchronization points between the nodes. Distributed applications can be either coarse-grained or fine-grained.

distributed computing

Computing with distributed applications, running the application on several nodes simultaneously.

distributed computing demos

Demonstration programs that use Parallel Computing Toolbox software, as opposed to sequential demos.

DNS

Domain Name System. A system that translates Internet domain names into IP addresses.

dynamic licensing

The ability of a MATLAB worker or lab to employ all the functionality you are licensed for in the MATLAB client, while checking out only an engine license. When a job is created in the MATLAB client with Parallel Computing Toolbox software, the products for which the client is licensed will be available for all workers or labs that evaluate tasks for that job. This allows you to run any code on the cluster that you are licensed for on your MATLAB client, without requiring extra licenses for the worker beyond MATLAB Distributed Computing Server software. For a list of products that are not eligible for use with Parallel Computing Toolbox software, see `http://www.mathworks.com/products/ineligible_programs/`.

fine-grained application
An application for which run time is significantly less than the communication time needed to start and stop the program. Compare to coarse-grained applications.

head node
Usually, the node of the cluster designated for running the job manager and license manager. It is often useful to run all the nonworker related processes on a single machine.

heterogeneous cluster
A cluster that is not homogeneous.

homogeneous cluster
A cluster of identical machines, in terms of both hardware and software.

independent job
A job composed of independent tasks, which do not communication with each other during evaluation. Tasks do not need to run simultaneously.

job
The complete large-scale operation to perform in MATLAB, composed of a set of tasks.

job manager
The MathWorks process that queues jobs and assigns tasks to workers. A third-party process that performs this function is called a scheduler. The general term "scheduler" can also refer to a job manager.

job manager checkpoint information
Snapshot of information necessary for the job manager to recover from a system crash or reboot.

job manager database
The database that the job manager uses to store the information about its jobs and tasks.

job manager lookup process
The process that allows clients, workers, and job managers to find each other. It starts automatically when the job manager starts.

lab
When workers start, they work independently by default. They can then connect to each other and work together as peers, and are then referred to as labs.

LOGDIR
The name of the parameter in the `mdce_def` file that defines the directory where logs are stored.

MathWorks job manager
See job manager.

MATLAB client
See client.

MATLAB job scheduler (MJS)
The MathWorks process that queues jobs and assigns tasks to workers. Formerly known as a job manager.

MATLAB pool
A collection of labs that are reserved by the client for execution of `parfor`-loops or `spmd` statements. See also lab.

MATLAB worker
See worker.

mdce
The service that has to run on all machines before they can run a job manager or worker. This is the engine foundation process, making sure that the job manager and worker processes that it controls are always running.

Note that the program and service name is all lowercase letters.

mdce_def file
The file that defines all the defaults for the mdce processes by allowing you to set preferences or definitions in the form of parameter values.

MPI
: Message Passing Interface, the means by which labs communicate with each other while running tasks in the same job.

node
: A computer that is part of a cluster.

parallel application
: The same application that runs on several labs simultaneously, with communication, shared data, or synchronization points between the labs.

private array
: An array which resides in the workspaces of one or more, but perhaps not all labs. There might or might not be a relationship between the values of these arrays among the labs.

random port
: A random unprivileged TCP port, i.e., a random TCP port above 1024.

register a worker
: The action that happens when both worker and job manager are started and the worker contacts job manager.

replicated array
: An array which resides in the workspaces of all labs, and whose size and content are identical on all labs.

scheduler
: The process, either third-party or the MathWorks job manager, that queues jobs and assigns tasks to workers.

spmd (single program multiple data)
: A block of code that executes simultaneously on multiple labs in a MATLAB pool. Each lab can operate on a different data set or different portion of distributed data, and can communicate with other participating labs while performing the parallel computations.

task
: One segment of a job to be evaluated by a worker.

variant array
An array which resides in the workspaces of all labs, but whose content differs on these labs.

worker
The MATLAB session that performs the task computations. Also known as the MATLAB worker or worker process.

worker checkpoint information
Files required by the worker during the execution of tasks.

Index