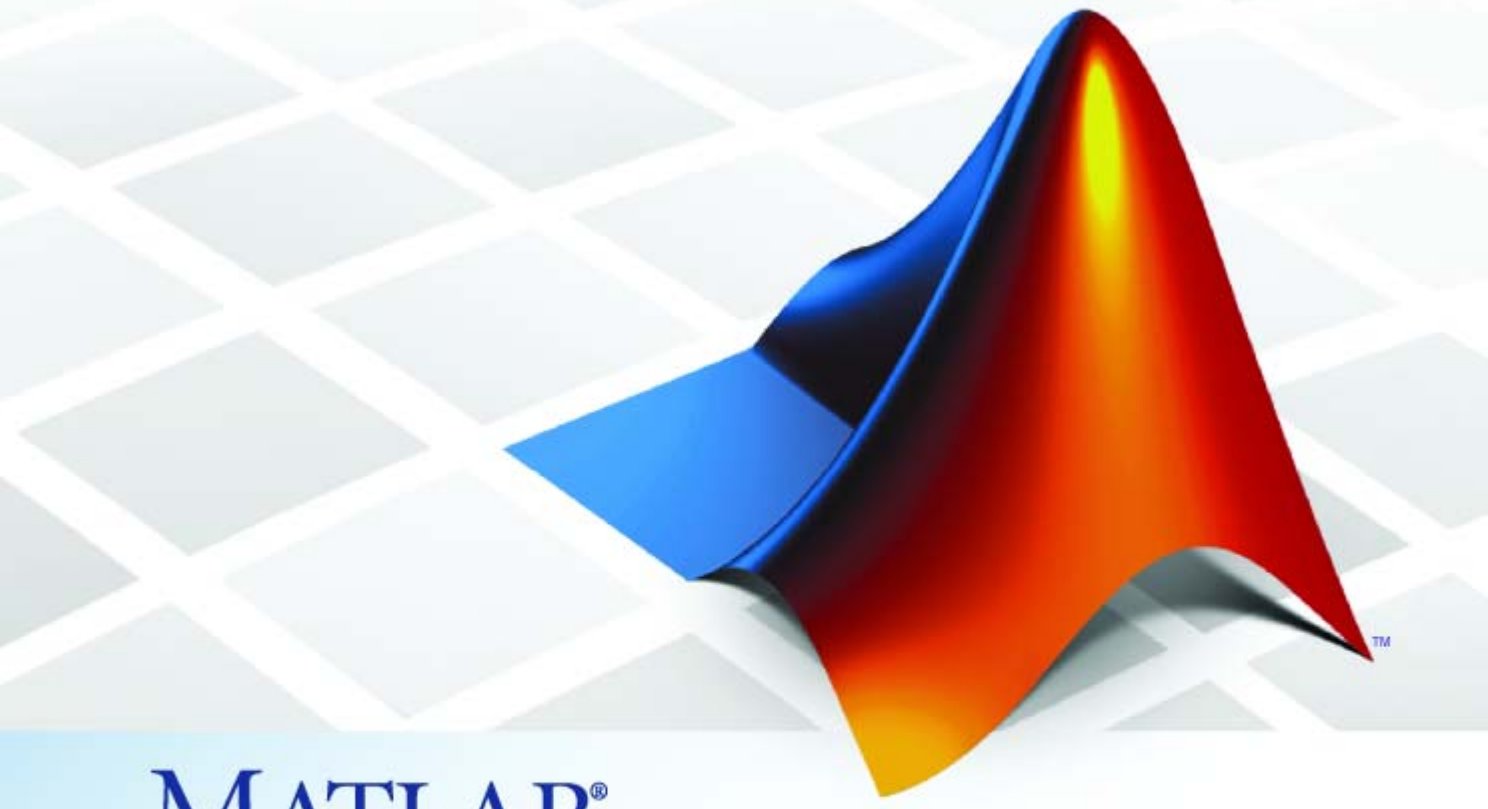
# Parallel Computing Toolbox™ 4

User's Guide



MATLAB®

The MathWorks™

*Accelerating the pace of engineering and science*

### How to Contact The 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*

© COPYRIGHT 2004–2010 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

#### Trademarks

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

#### Patents

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

**Revision History**

| | | |
|---|---|---|
| November 2004 | Online only | New for Version 1.0 (Release 14SP1+) |
| March 2005 | Online only | Revised for Version 1.0.1 (Release 14SP2) |
| September 2005 | Online only | Revised for Version 1.0.2 (Release 14SP3) |
| November 2005 | Online only | Revised for Version 2.0 (Release 14SP3+) |
| March 2006 | Online only | Revised for Version 2.0.1 (Release 2006a) |
| September 2006 | Online only | Revised for Version 3.0 (Release 2006b) |
| March 2007 | Online only | Revised for Version 3.1 (Release 2007a) |
| September 2007 | Online only | Revised for Version 3.2 (Release 2007b) |
| March 2008 | Online only | Revised for Version 3.3 (Release 2008a) |
| October 2008 | Online only | Revised for Version 4.0 (Release 2008b) |
| March 2009 | Online only | Revised for Version 4.1 (Release 2009a) |
| September 2009 | Online only | Revised for Version 4.2 (Release 2009b) |
| March 2010 | Online only | Revised for Version 4.3 (Release 2010a) |

# Contents

## 1 Getting Started


**Product Overview** . . . . . 1-2

**Typical Use Cases** . . . . . 1-3
Parallel for-Loops (parfor) . . . . . 1-3
Batch Jobs . . . . . 1-4
Large Data Sets . . . . . 1-4

**Introduction to Parallel Solutions** . . . . . 1-5
Interactively Running a Loop in Parallel . . . . . 1-5
Running a Batch Job . . . . . 1-7
Running a Batch Parallel Loop . . . . . 1-8
Using Distributed Arrays, spmd, and Composites . . . . . 1-10

**Determining Product Installation and Versions** . . . . . 1-12


## 2 Parallel for-Loops (parfor)


**Getting Started with parfor** . . . . . 2-2
Introduction . . . . . 2-2
When to Use parfor . . . . . 2-3
Setting up MATLAB Resources Using matlabpool . . . . . 2-3
Creating a parfor-Loop . . . . . 2-4
Differences Between for-Loops and parfor-Loops . . . . . 2-5
Reduction Assignments . . . . . 2-6
Displaying Output . . . . . 2-7

**Programming Considerations** . . . . . 2-8
MATLAB Path . . . . . 2-8
Error Handling . . . . . 2-8

Limitations ..... 2-9
Using Objects in parfor Loops ..... 2-11
Performance Considerations ..... 2-12
Compatibility with Earlier Versions of MATLAB Software ..... 2-13

**Advanced Topics** ..... 2-14
About Programming Notes ..... 2-14
Classification of Variables ..... 2-14
Improving Performance ..... 2-28

## 3 Single Program Multiple Data (spmd)

**Using spmd Constructs** ..... 3-2
Introduction ..... 3-2
When to Use spmd ..... 3-2
Setting Up MATLAB Resources Using matlabpool ..... 3-3
Defining an spmd Statement ..... 3-4
Displaying Output ..... 3-6

**Accessing Data with Composites** ..... 3-7
Introduction ..... 3-7
Creating Composites in spmd Statements ..... 3-7
Variable Persistence and Sequences of spmd ..... 3-9
Creating Composites Outside spmd Statements ..... 3-10

**Distributing Arrays** ..... 3-12
Distributed Versus Codistributed Arrays ..... 3-12
Creating Distributed Arrays ..... 3-12
Creating Codistributed Arrays ..... 3-13

**Programming Considerations** ..... 3-15
MATLAB Path ..... 3-15
Error Handling ..... 3-15
Limitations ..... 3-15

## 4 Interactive Parallel Computation with pmode


**Introduction** . . . . . 4-2

**Getting Started with pmode** . . . . . 4-3

**Parallel Command Window** . . . . . 4-10

**Running pmode on a Cluster** . . . . . 4-15

**Plotting in pmode** . . . . . 4-16

**Limitations and Unexpected Results** . . . . . 4-18
Using Graphics in pmode . . . . . 4-18

**Troubleshooting** . . . . . 4-19
Connectivity Testing . . . . . 4-19
Hostname Resolution . . . . . 4-19
Socket Connections . . . . . 4-19


## 5 Math with Codistributed Arrays


**Array Types** . . . . . 5-2
Introduction . . . . . 5-2
Nondistributed Arrays . . . . . 5-2
Codistributed Arrays . . . . . 5-4

**Working with Codistributed Arrays** . . . . . 5-5
How MATLAB Software Distributes Arrays . . . . . 5-5
Creating a Codistributed Array . . . . . 5-7
Local Arrays . . . . . 5-11
Obtaining information About the Array . . . . . 5-12
Changing the Dimension of Distribution . . . . . 5-13
Restoring the Full Array . . . . . 5-14

Indexing into a Codistributed Array ..... 5-15
2-Dimensional Distribution ..... 5-17

**Using a for-Loop Over a Distributed Range (for-drange)** ..... 5-21
Parallelizing a for-Loop ..... 5-21
Codistributed Arrays in a for-drange Loop ..... 5-22

**Using MATLAB Functions on Codistributed Arrays** ... 5-25

## 6 Programming Overview

**Product Introduction** ..... 6-2
Overview ..... 6-2
Toolbox and Server Components ..... 6-3

**Using Parallel Computing Toolbox Software** ..... 6-8
Example: Evaluating a Basic Function ..... 6-8
Example: Programming a Basic Job with a Local Scheduler ..... 6-8
Getting Help ..... 6-10

**Program Development Guidelines** ..... 6-12

**Life Cycle of a Job** ..... 6-14

**Programming with User Configurations** ..... 6-16
Defining Configurations ..... 6-16
Exporting and Importing Configurations ..... 6-23
Validating Configurations ..... 6-24
Applying Configurations in Client Code ..... 6-27

**Programming Tips and Notes** ..... 6-29
Saving or Sending Objects ..... 6-29
Current Working Directory of a MATLAB Worker ..... 6-29
Using clear functions ..... 6-30

Running Tasks That Call Simulink Software ..... 6-30
Using the pause Function ..... 6-30
Transmitting Large Amounts of Data ..... 6-30
Interrupting a Job ..... 6-30
Speeding Up a Job ..... 6-31

**Using the Parallel Profiler** ..... 6-32
Introduction ..... 6-32
Collecting Parallel Profile Data ..... 6-32
Viewing Parallel Profile Data ..... 6-33
Parallel Profiler Demos ..... 6-42

**Benchmarking Performance** ..... 6-44
Demos ..... 6-44
HPC Challenge Benchmarks ..... 6-44

**Troubleshooting and Debugging** ..... 6-45
Object Data Size Limitations ..... 6-45
File Access and Permissions ..... 6-45
No Results or Failed Job ..... 6-47
Connection Problems Between the Client and Job Manager ..... 6-48

## 7 Evaluating Functions in a Cluster

**Evaluating Functions Synchronously** ..... 7-2
Scope of dfeval ..... 7-2
Arguments of dfeval ..... 7-3
Example — Using dfeval ..... 7-4

**Evaluating Functions Asynchronously** ..... 7-8

## 8 Programming Distributed Jobs


**Using a Local Scheduler** . . . . . 8-2
Creating and Running Jobs with a Local Scheduler . . . . . 8-2
Local Scheduler Behavior . . . . . 8-7

**Using a Job Manager** . . . . . 8-8
Creating and Running Jobs with a Job Manager . . . . . 8-8
Sharing Code . . . . . 8-13
Managing Objects in the Job Manager . . . . . 8-16

**Using a Fully Supported Third-Party Scheduler** . . . . . 8-20
Creating and Running Jobs . . . . . 8-20
Sharing Code . . . . . 8-28
Managing Objects . . . . . 8-30

**Using the Generic Scheduler Interface** . . . . . 8-33
Overview . . . . . 8-33
MATLAB Client Submit Function . . . . . 8-34
Example — Writing the Submit Function . . . . . 8-38
MATLAB Worker Decode Function . . . . . 8-39
Example — Writing the Decode Function . . . . . 8-41
Example — Programming and Running a Job in the Client . . . . . 8-42
Supplied Submit and Decode Functions . . . . . 8-47
Managing Jobs . . . . . 8-48
Summary . . . . . 8-51


## 9 Programming Parallel Jobs


**Introduction** . . . . . 9-2

**Using a Supported Scheduler** . . . . . 9-4
Schedulers and Conditions . . . . . 9-4
Coding the Task Function . . . . . 9-4
Coding in the Client . . . . . 9-5

**Using the Generic Scheduler Interface** . . . . . **9-8**
Introduction . . . . . **9-8**
Coding in the Client . . . . . **9-8**

**Further Notes on Parallel Jobs** . . . . . **9-11**
Number of Tasks in a Parallel Job . . . . . **9-11**
Avoiding Deadlock and Other Dependency Errors . . . . . **9-11**

## 10 Object Reference

**Data Objects** . . . . . **10-2**

**Scheduler Objects** . . . . . **10-2**

**Job Objects** . . . . . **10-3**

**Task Objects** . . . . . **10-3**

**Worker Objects** . . . . . **10-3**

## 11 Objects — Alphabetical List

## 12 Function Reference

**Parallel Code Execution** . . . . . **12-2**
Parallel Code on a MATLAB Pool . . . . . **12-2**
Configuration, Input, and Output . . . . . **12-2**
Interactive Functions . . . . . **12-3**

**Distributed and Codistributed Arrays** . . . . . **12-3**
Toolbox Functions . . . . . **12-3**
Overloaded MATLAB Functions . . . . . **12-4**

**Job and Task Programming** . . . . . **12-6**
Job Creation . . . . . **12-6**
Job Management . . . . . **12-7**
Task Execution Information . . . . . **12-8**
Object Control . . . . . **12-9**

**Interlab Communication Within a Parallel Job** . . . . . **12-9**

**Utilities** . . . . . **12-10**

## 13 Functions — Alphabetical List

## 14 Property Reference

**Job Manager Properties** . . . . . **14-2**

**Scheduler Properties** . . . . . **14-3**

**Job Properties** . . . . . **14-5**

**Task Properties** . . . . . **14-6**

**Worker Properties** . . . . . **14-8**

Properties — Alphabetical List
15

Glossary

Index

1

# Getting Started


- "Product Overview" on page 1-2
- "Typical Use Cases" on page 1-3
- "Introduction to Parallel Solutions" on page 1-5
- "Determining Product Installation and Versions" on page 1-12

# Product Overview

Parallel Computing Toolbox™ software allows you to offload work from one MATLAB® session (the client) to other MATLAB sessions, called workers. You can use multiple workers to take advantage of parallel processing. You can use a local worker to keep your MATLAB client session free for interactive work, or with MATLAB® Distributed Computing Server™ you can take advantage of another computer's speed.

Parallel Computing Toolbox software allows you to run as many as eight MATLAB workers on your local machine in addition to your MATLAB client session. MATLAB Distributed Computing Server software allows you to run as many MATLAB workers on a remote cluster of computers as your licensing allows.

# Typical Use Cases

| In this section... |
| --- |
| “Parallel for-Loops (parfor)” on page 1-3 |
| “Batch Jobs” on page 1-4 |
| “Large Data Sets” on page 1-4 |

## 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 for many use 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.

## Batch Jobs

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.

## 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 Running a Loop in Parallel" on page 1-5

"Running a Batch Job" on page 1-7

"Running a Batch Parallel Loop" on page 1-8

"Using Distributed Arrays, spmd, and Composites" on page 1-10

## Interactively Running 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)
```

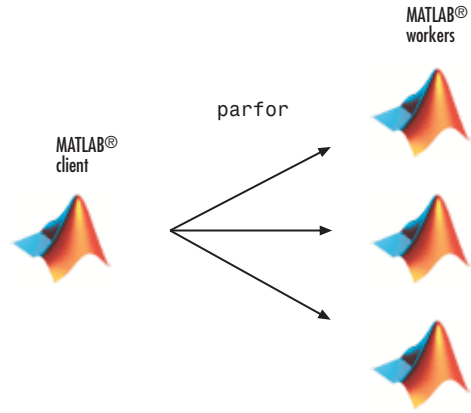
**2** To interactively run code that contains a parallel loop, you first open a MATLAB pool. This reserves a collection of MATLAB worker sessions to run your loop iterations. The MATLAB pool can consist of MATLAB sessions running on your local machine or on a remote cluster:

```
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 parallel configurations, you can control how many workers run your loops, and whether the workers are local or remote. For more information on parallel configurations, see “Programming with User Configurations” on page 6-16.

## Running 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
```

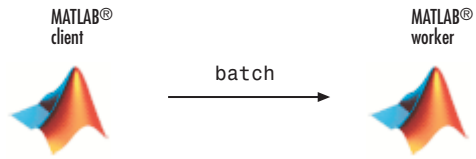
**2** In the MATLAB Editor, enter the text of the `for`-loop:

```
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)
```

## Running 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:

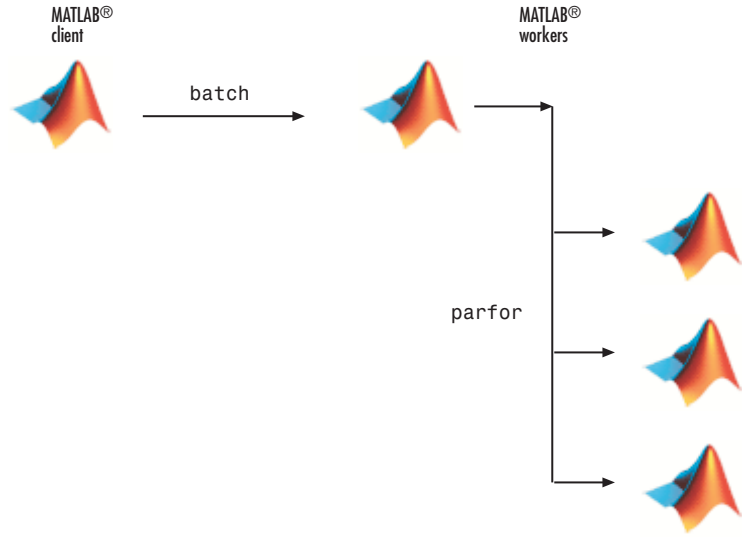
   ```
   parfor i=1:1024
     A(i) = sin(i*2*pi/1024);
   end
   ```

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)
```

# Using Distributed Arrays, spmd, and Composites

## Distributed Arrays

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

First, open the MATLAB pool:

```
matlabpool open  % Use default parallel configuration
```

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

```
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 labs, you can close the MATLAB pool. Data on the labs 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 labs (workers) in the MATLAB pool. The `spmd` block can run on some or all the labs in the pool.

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

This code creates an individual 4-by-4 matrix, `R`, of random numbers on each 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 labs. On the client, these variables are called *Composite* objects. Each element of a composite is a symbol referencing the value (data) on a lab in the pool. Note that because a variable might not be defined on every lab, a Composite might have undefined elements.

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

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

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

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

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

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

A typical use case for `spmd` is to run the same code on a number of labs, 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 labs are accessible from the client as `RES{1}` from lab 1, `RES{2}` from lab 2, etc.

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

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

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

```
matlabpool close
```

Although data persists on the labs 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.

2

# Parallel for-Loops (parfor)


- "Getting Started with parfor" on page 2-2
- "Programming Considerations" on page 2-8
- "Advanced Topics" on page 2-14

# Getting Started with parfor

**In this section...**

"Introduction" on page 2-2
"When to Use parfor" on page 2-3
"Setting up MATLAB Resources Using matlabpool" on page 2-3
"Creating a parfor-Loop" on page 2-4
"Differences Between for-Loops and parfor-Loops" on page 2-5
"Reduction Assignments" on page 2-6
"Displaying Output" on page 2-7

## Introduction

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.

## 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.

## Setting 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 scheduler and cluster by selecting a parallel configuration. For a description of how to manage and use configurations, see “Programming with User Configurations” on page 6-16.

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 parallel configuration. If the local configuration is your default and does not specify the number of workers, this starts one worker per core (maximum of eight) 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.

---

## Creating a parfor-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
for i = 1:8
   A(i) = i;
end
A
```

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

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" on page 2-8

"Error Handling" on page 2-8

"Limitations" on page 2-9

"Using Objects in parfor Loops" on page 2-11

"Performance Considerations" on page 2-12

"Compatibility with Earlier Versions of MATLAB Software" on page 2-13

## MATLAB Path

All workers executing a `parfor`-loop must have the same MATLAB path configuration 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 path on all workers.

## 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 "Naming Variables" 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.

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 parfor-Loops

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

### 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.

### 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 2-28.

## 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 2-29.

## 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`<br>`  loop body`<br>`  .`<br>`  .`<br>`end` | `for i = drange(range)`<br>`  loop body`<br>`  .`<br>`  .`<br>`end` |
| Parallel loop for implicit distribution of work | Not Implemented | `parfor i = range`<br>`  loop body`<br>`  .`<br>`  .`<br>`end` |

# Advanced Topics

| In this section... |
| --- |
| "About Programming Notes" on page 2-14 |
| "Classification of Variables" on page 2-14 |
| "Improving Performance" on page 2-28 |

## 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 M-Lint to help make your `parfor`-loops comply with these guidelines.

**Required (static)**: Description of the guideline or restriction

## Classification of Variables


- "Overview" on page 2-14
- "Loop Variable" on page 2-15
- "Sliced Variables" on page 2-16
- "Broadcast Variables" on page 2-20
- "Reduction Variables" on page 2-20
- "Temporary Variables" on page 2-26

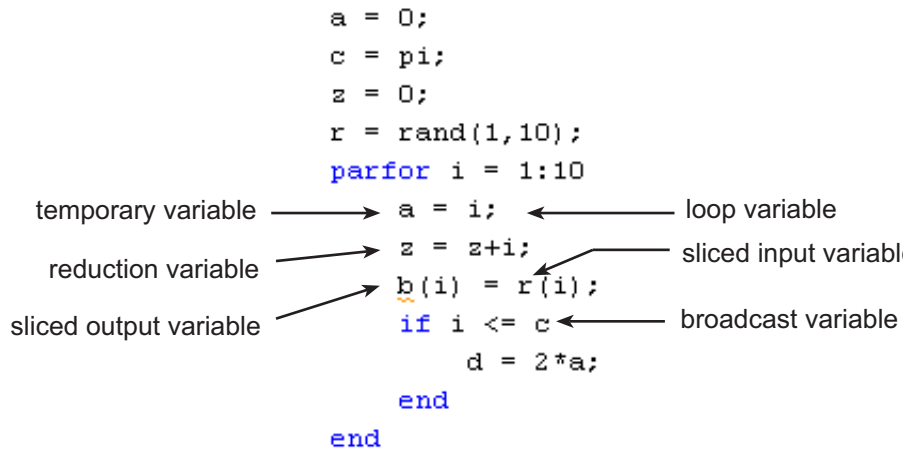

### Overview

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:




## 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) variable; and every other index is a constant, a simple 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 2-22 |
| `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 A > 5*k
      A = A + i;
   else
      A = [A, 4+i];
   end
   % loop body continued
end
```

```
parfor i = 1:n
   if A > 5*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 A > 5*k
      A = [A, 4+i];
   else
      A = [r(i), A];
   % loop body continued
end
```

```
parfor i = 1:n
   if A > 5*k
      A = [A, 4+i];
   else
      A = [A, r(i)];
   % 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 2-20, 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;
parfor i = 1:n
   a = f(a,i);
   % loop body continued
   f = @times;  % Affects f
end
```

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

Note that the operators `&&` and `||` are not listed in the table in "Reduction Variables" on page 2-20. 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. If it is not, different executions of the loop might result in different answers. In other words, although `parfor` gives you the ability to declare that a function is associative, MATLAB does not detect misuse of that ability.

**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 2-20 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`.

## 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.

3

# Single Program Multiple Data (spmd)


- "Using spmd Constructs" on page 3-2
- "Accessing Data with Composites" on page 3-7
- "Distributing Arrays" on page 3-12
- "Programming Considerations" on page 3-15

# Using spmd Constructs

**In this section...**

"Introduction" on page 3-2

"When to Use spmd" on page 3-2

"Setting Up MATLAB Resources Using matlabpool" on page 3-3

"Defining an spmd Statement" on page 3-4

"Displaying Output" on page 3-6

## 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 labs. Variables assigned inside the `spmd` statement on the labs 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 labs. You run one program in the MATLAB client, and those parts of it labeled as `spmd` blocks run on the labs. 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 labs, each lab can have different, unique data for that code. So multiple data sets can be accommodated by multiple labs.

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 labs. Some common cases are:

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

## 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 labs might be running remotely on a cluster, or they might run locally on your MATLAB client machine. You identify a scheduler and cluster by selecting a parallel configuration. For a description of how to manage and use configurations, see "Programming with User Configurations" on page 6-16.

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

```
matlabpool
```

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

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

```
matlabpool 3
```

To use a different configuration, type:

```
matlabpool MyConfigName
```

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

```
matlabpool size
```

This command returns a value indicating the number of labs 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 lab.

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 labs in the MATLAB pool. If you want to limit the execution to only a portion of these labs, specify exactly how many labs to run on:

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

This statement requires that `n` labs run the `spmd` code. `n` must be less than or equal to the number of labs in the open MATLAB pool. If the pool is large enough, but `n` labs are not available, the statement waits until enough labs are available. If `n` is 0, the `spmd` statement uses no labs, 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 labs:

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

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

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

For example, create a random matrix on three labs:

```
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 labs used for an `spmd` statement each have a unique value for `labindex`. This lets you specify code to be run on only certain labs, 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 lab according to `labindex`, and use the same function on each lab to compute a result from the data:

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

The labs 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 labs, 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 Parallel Job" on page 12-9 and "Distributed and Codistributed Arrays" on page 12-3.

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

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

Each lab 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" on page 3-7

"Creating Composites in spmd Statements" on page 3-7

"Variable Persistence and Sequences of spmd" on page 3-9

"Creating Composites Outside spmd Statements" on page 3-10

## Introduction

Composite objects in the MATLAB client session let you directly access data values on the labs. 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 labs.
- Defining variables on labs 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 labs.

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 lab. 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 lab
end
MM{1} % In the client, MM is a Composite with one element per lab
     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 lab. For the labs 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 lab 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 lab 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 lab to client when you explicitly assign a variable in the client workspace using a Composite element:

```
M = MM{1} % Transfer data from lab 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 labs:

```
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 labs to the client, even if the assignment then goes to the same lab. In this case, `NN` must already exist as a Composite:

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

When finished, you can close the pool:

```
matlabpool close
```

## Variable Persistence and Sequences of spmd

The values stored on the labs 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 labs 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 labs

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 labs before an `spmd` statement begins executing on those labs. Assume a MATLAB pool is already open:

```
PP = Composite()
```

By default, this creates a Composite with an element for each lab in the MATLAB pool. You can also create Composites on only a subset of the labs 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 lab:

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

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

# Distributing Arrays

| In this section... |
| --- |
| "Distributed Versus Codistributed Arrays" on page 3-12 |
| "Creating Distributed Arrays" on page 3-12 |
| "Creating Codistributed Arrays" on page 3-13 |

## Distributed Versus Codistributed Arrays

You can create a distributed array in the MATLAB client, and its data is stored on the labs 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 labs. You cannot control the details of distribution when creating a distributed array.

You can create a codistributed array by executing on the labs 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 labs from which you execute code to create or manipulate them. Distributed arrays are partitioned among labs 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 labs of an open MATLAB pool.

- Use any of the overloaded distributed object methods to directly construct a distributed array on the labs. 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:

```
mablabpool open local 2
W = ones(6,6);
W = distributed(W); % Distribute to the labs
spmd
    T = W*2; % Calculation performed on labs, in parallel.
             % T and W are both codistributed arrays here.
end
T  % View results in client.
   % 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 labs running that job.
- Use any of the overloaded codistributed object methods to directly construct a codistributed array on the labs. This technique does not require that the array already exists in the labs. 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 lab 1, and 12 parts on lab 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 Considerations

| In this section... |
| --- |
| "MATLAB Path" on page 3-15 |
| "Error Handling" on page 3-15 |
| "Limitations" on page 3-15 |

## MATLAB Path

All labs executing an `spmd` statement must have the same MATLAB path configuration 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 labs, if possible. For more information, see the `matlabpool` reference page. When the labs are running on a different platform than the client, use the function `pctRunOnAll` to properly set the MATLAB path on all labs.

## Error Handling

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

Errors and warnings produced on labs are annotated with the lab 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 labs. 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.

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 labs 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.

4

# Interactive Parallel Computation with pmode

This chapter describes interactive pmode in the following sections:


- "Introduction" on page 4-2
- "Getting Started with pmode" on page 4-3
- "Parallel Command Window" on page 4-10
- "Running pmode on a Cluster" on page 4-15
- "Plotting in pmode" on page 4-16
- "Limitations and Unexpected Results" on page 4-18
- "Troubleshooting" on page 4-19

# Introduction

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

The way the labs remain synchronized is that each lab becomes idle when it completes a command or statement, waiting until all the labs working on this job have completed the same statement. Only when all the labs 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 lab running the job, where you can enter commands, see results, access each lab'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 labs is lost. Starting another pmode session always begins from a clean state.

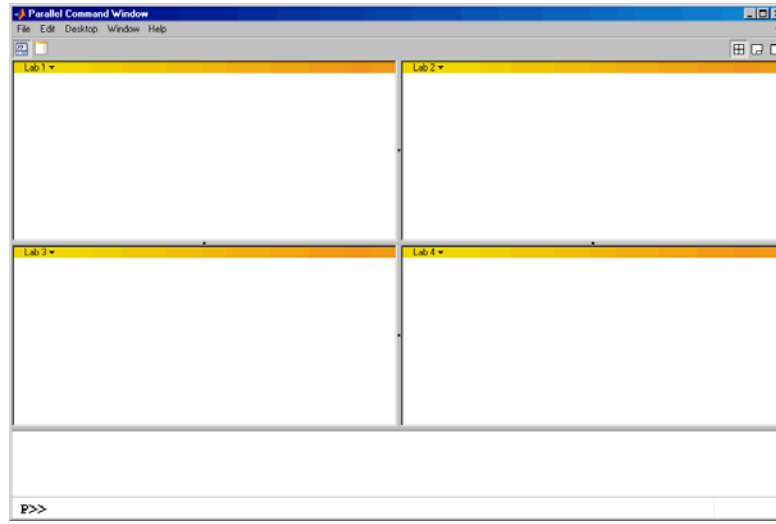
# Getting Started with pmode

This example uses a local scheduler and runs the labs 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 labs, creates a parallel job to run on those labs, and opens the Parallel Command Window.



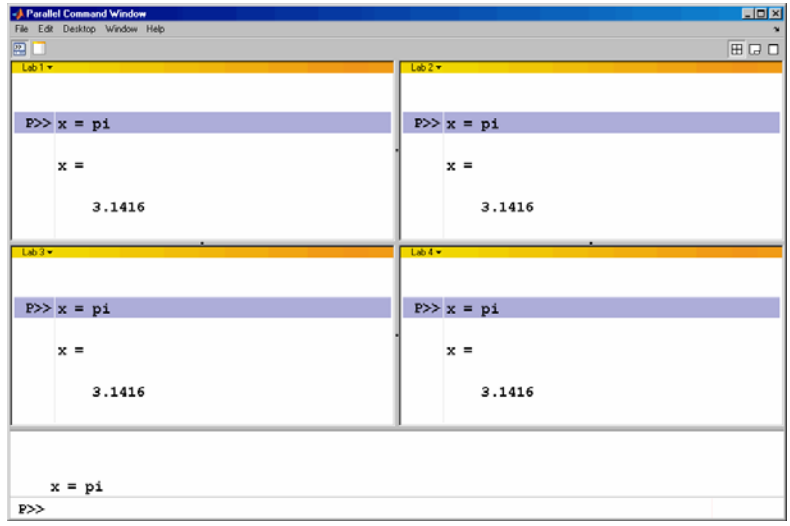

You can control where the command history appears. For this exercise, the position is set by clicking **Window > History Position > Above Prompt**, but you can set it according to your own preference.

**2** To illustrate that commands at the pmode prompt are executed on all labs, 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 labs.

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




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

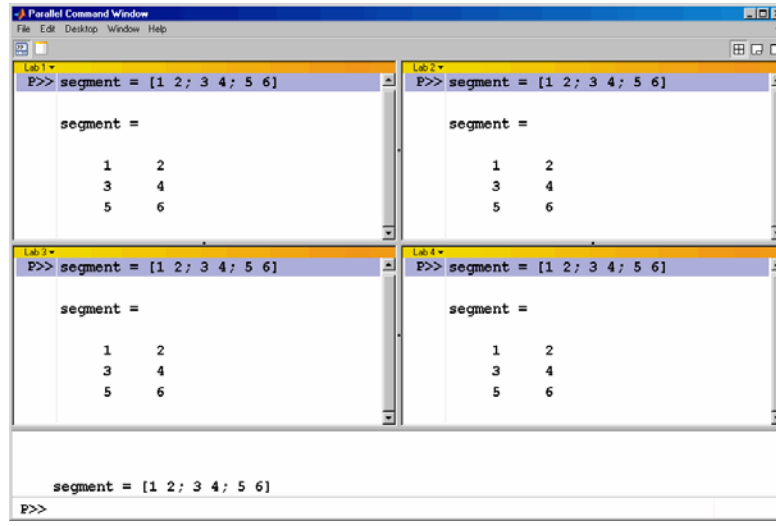
```
P>> x = labindex
```

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

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

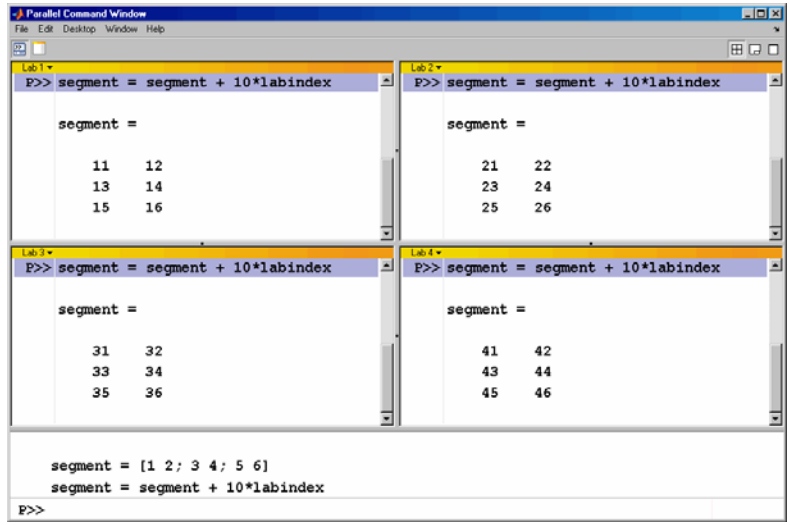
**6** Create a replicated array on all the labs.

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



**7** Assign a unique value to the array on each lab, dependent on the lab number. With a different value on each lab, this is a variant array.

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




**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 labs.

```
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 labs. On each lab, `segment` provided the data for the local portion of the `whole` array.

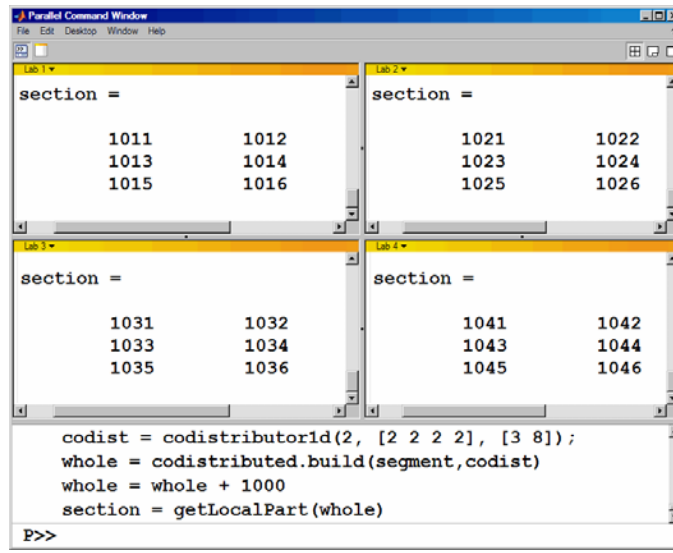
**9** Now, when you operate on the codistributed array `whole`, each lab handles the calculations on only its portion, or segment, of the array, not the whole array.

```
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 lab.

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

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




**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 labs. See the `gather` reference page for the syntax to gather the array into the workspace of only one lab.

**12** Because the labs 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
```

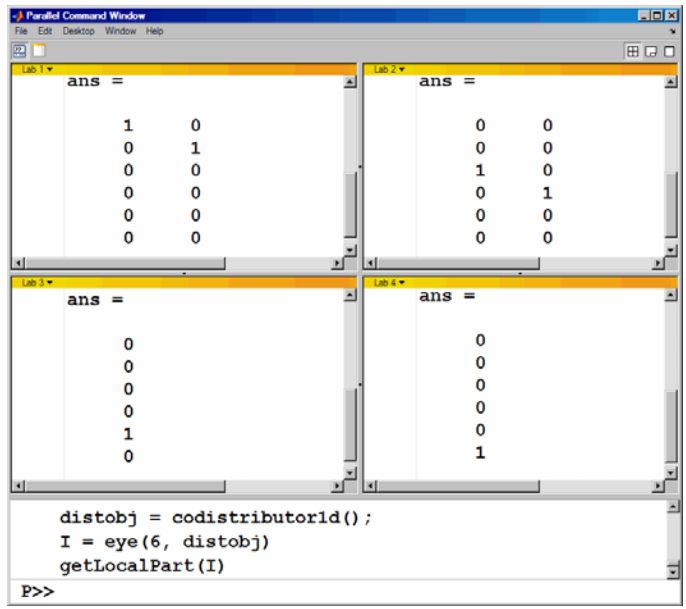
To see the array, type its name.

```
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.

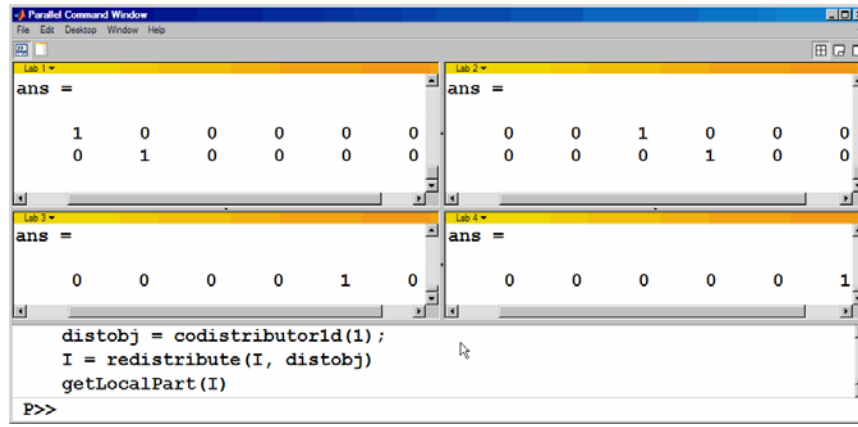
```
P>> distobj = codistributor1d();
P>> I = eye(6, distobj)
P>> getLocalPart(I)
```

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



**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)
```




**15** Exit pmode and return to the regular MATLAB desktop.

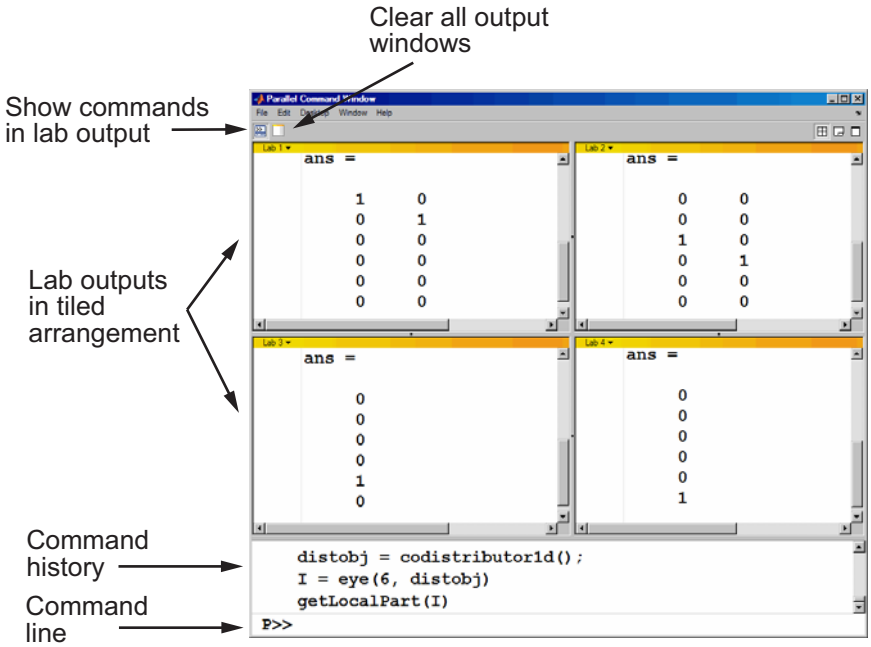
```
P>> pmode exit
```

# Parallel Command Window

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

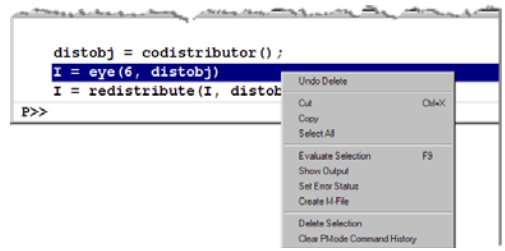
```
pmode start local 4
```

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

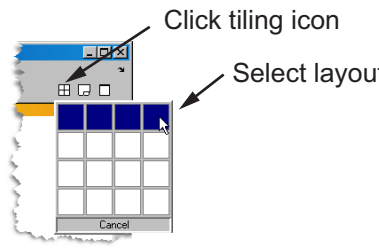


The Parallel Command Window offers much of the same functionality as the MATLAB desktop, including command line, output, and command history.

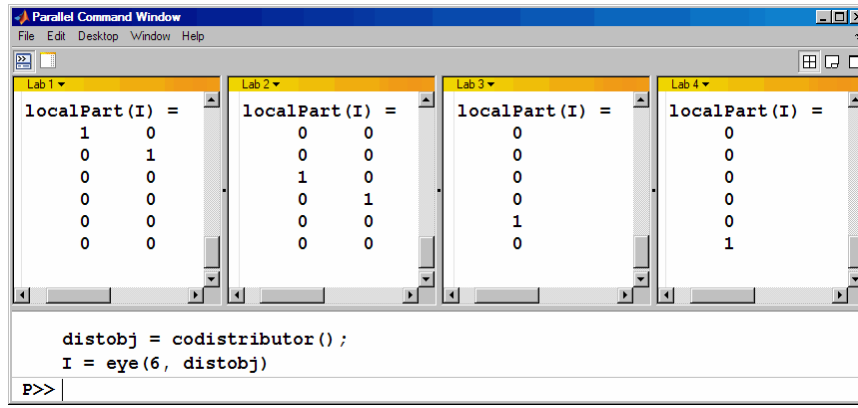
When you select one or more lines in the command history and right-click, you see the following context menu.




You have several options for how to arrange the tiles showing your lab 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.



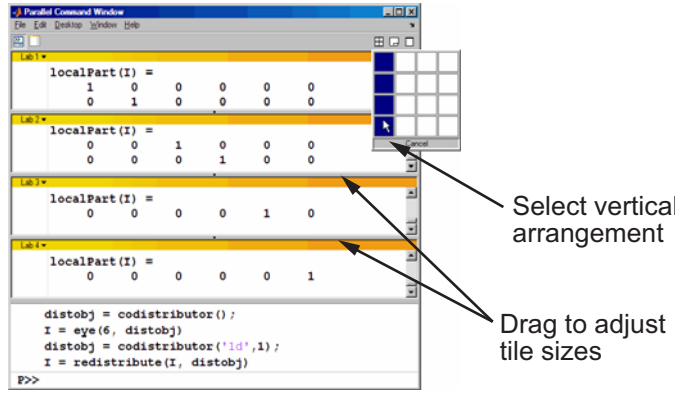
This arrangement results in the following figure, which might be more convenient for viewing data distributed by columns.



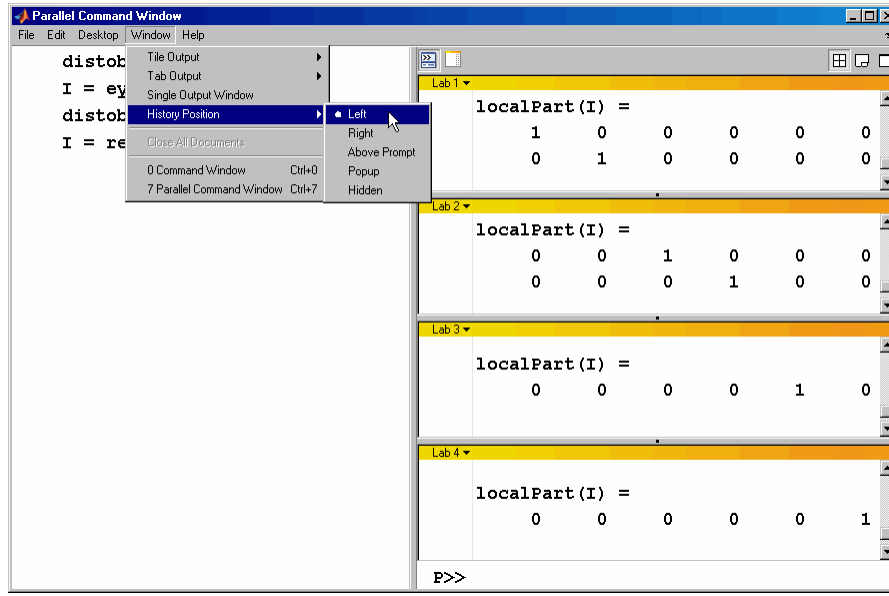

Alternatively, if the data is distributed by rows, you might want to stack the lab tiles vertically. For the following figure, the data is reformatted with the command

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

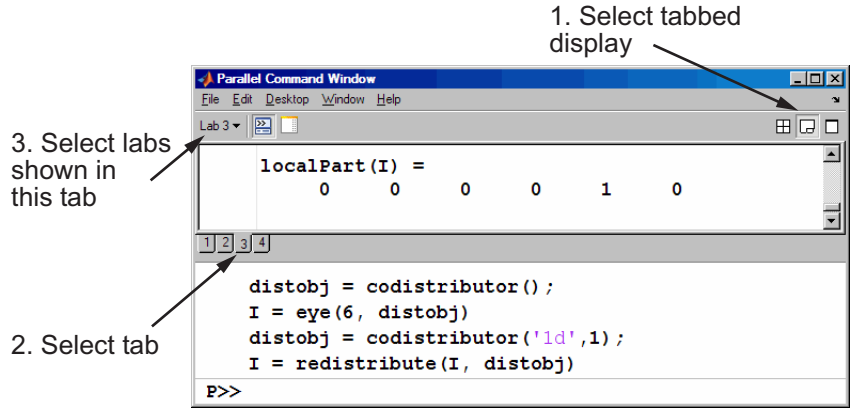
When you rearrange the tiles, you see the following.



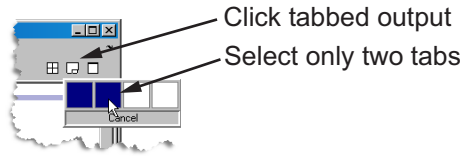
You can control the relative positions of the command window and the lab output. The following figure shows how to set the output to display beside the input, rather than above it.



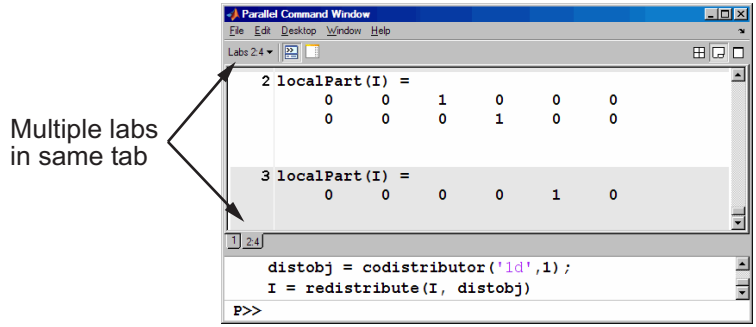

You can choose to view the lab outputs by tabs.



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




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



# Running pmode on a Cluster

When you run pmode on a cluster of labs, 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 a configuration. For more details about creating and using configurations, see "Programming with User Configurations" on page 6-16.

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

```
pmode start <config-name> <num-labs>
```

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

If you omit `<config-name>`, pmode uses the default configuration (see the `defaultParallelConfig` reference page).

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

# Plotting in pmode

Because the labs 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 lab.
2. Transfer the whole array from any lab 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, each lab 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 lab 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 lab 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.

# Limitations and Unexpected Results

## Using Graphics in pmode

### Displaying a GUI

The labs that run the tasks of a parallel job are MATLAB sessions without displays. As a result, these labs 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 lab
```

Then use the graphical tool on the MATLAB client.

### Using Simulink Software

Because the labs 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 labs 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 labs via pmode; also, if the labs had accessed the model previously, they must close and open the model again to see the latest saved changes.

# Troubleshooting

| In this section... |
| --- |
| "Connectivity Testing" on page 4-19 |
| "Hostname Resolution" on page 4-19 |
| "Socket Connections" on page 4-19 |

## 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 lab 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 lab 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 lab and client machines.
- Use `pctconfig` to change the client's `pmodeport` property. This determines the port that the labs will use to contact the client in the next pmode session.

5

# Math with Codistributed Arrays

This chapter describes the distribution or partition of data across several labs, and the functionality provided for operations on that data in `spmd` statements, parallel jobs, and pmode. The sections are as follows.


- "Array Types" on page 5-2
- "Working with Codistributed Arrays" on page 5-5
- "Using a for-Loop Over a Distributed Range (for-drange)" on page 5-21
- "Using MATLAB Functions on Codistributed Arrays" on page 5-25

# Array Types

| In this section... |
| --- |
| "Introduction" on page 5-2 |
| "Nondistributed Arrays" on page 5-2 |
| "Codistributed Arrays" on page 5-4 |

## 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 lab and assigns a common variable to them. Any operation performed on that variable affects all individual arrays assigned to it. If you display from lab 1 the value assigned to this variable, all labs 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 lab:

- "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 labs, and its size and content are identical on all labs. When you create the array, MATLAB assigns it to the same variable on all labs. If you display in spmd the value assigned to this variable, all labs 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 labs, but its content differs on one or more labs. When you create the array, MATLAB assigns a different value to the same variable on all labs. If you display the value assigned to this variable, all labs 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 lab.

```
spmd
    B = magic(3);      %replicated on all labs
    B = B + labindex;  %now a variant array, different on each lab
end
```

### Private Arrays

A *private array* is defined on one or more, but not all labs. You could create this array by using the lab index 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 lab. *Codistributed arrays*, on the other hand, are partitioned into segments, with each segment residing in the workspace of a different lab. Each lab has its own array segment to work with. Reducing the size of the array that each lab 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 over four labs. The resulting array D is also 3-by-10 in size, but only a segment of the full array resides on each lab.

```
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" on page 5-5
"Creating a Codistributed Array" on page 5-7
"Local Arrays" on page 5-11
"Obtaining information About the Array" on page 5-12
"Changing the Dimension of Distribution" on page 5-13
"Restoring the Full Array" on page 5-14
"Indexing into a Codistributed Array" on page 5-15
"2-Dimensional Distribution" on page 5-17

## How MATLAB Software Distributes Arrays

When you distribute an array to a number of labs, MATLAB software partitions the array into segments and assigns one segment of the array to each lab. You can partition a two-dimensional array horizontally, assigning columns of the original array to the different labs, 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 labs, you can partition it either by columns, giving each lab an 80-by-250 segment, or by rows, with each lab getting a 20-by-1000 segment. If the array dimension does not divide evenly over the number of labs, 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 lab creates an identical array in its own workspace and assigns it to variable `A`, where `A` is local to that lab. The second command distributes `A`, creating a single 80-by-1000 array `D` that spans all four labs. lab 1 stores columns 1 through 250, lab 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 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 labs and thus takes more time.

## How MATLAB Displays a Codistributed Array

For each lab, 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 labs, with two columns on each lab, 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 Lab

In distributing an array of `N` rows, if `N` is evenly divisible by the number of labs, MATLAB stores the same number of rows (`N/numlabs`) on each lab. When this number is not evenly divisible by the number of labs, 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:

- "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" on page 5-8 — Start with a large array that is replicated on all labs, and partition it so that the pieces are distributed across the labs. 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 lab, 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.

## 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 labs using the `codistributed` function. Each lab 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 lab.

As a simple example, the following line of code creates a 4-by-8 replicated matrix on each lab 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 lab, while only a segment of array `D` exists on each lab.

```
spmd, size(A), size(D), end
```

Examining the variables in the client workspace, an array that is codistributed among the labs 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 lab first, and then combine them into a single array that is distributed across the labs.

This example creates a 4-by-250 variant array A on each of four labs and then uses `codistributor` to distribute these segments across four labs, 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 lab:

```
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 labs 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.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 lab is a piece of a larger array. Each lab 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 lab. Suppose you distribute an array across four labs:

```
spmd(4)
    A = [1:80; 81:160; 161:240];
    D = codistributed(A);
    size(D)
       L = getLocalPart(D);
    size(L)
end
```

returns on each lab:

```
3    80
3    20
```

Each lab 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 lab, because the 80 columns of `D` are distributed over four labs.

### 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 lab:

```
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 labs.

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 labs:

```
spmd
    D = rand(8, 16, codistributor());
    size(getLocalPart(D))
```

```
end
```

returns on each lab:

```
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 lab:

```
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 labs and combines them into a replicated array on all labs, or into a single array on one lab.

Distribute a 4-by-10 array to four labs 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 lab 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 labs, the starting index on lab 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 labs, and you want to locate its element number 225,000. That is, you want to know what lab contains this element, and in what position in the local part of the vector on that lab. 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 lab %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 lab 1.
```

If you run this code on a pool of five workers you get this result:

```
Lab 2:
  Element is in position 25000 on lab 2.
```

Notice if you use a pool of a different size, the element ends up in a different location on a different lab, 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 labs, with a 4-by-4 block as the local part on each lab. In this case, the lab grid is a 2-by-2 arrangement of the labs, 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 labs according to this scheme:

| LAB 1 | | | | LAB 2 | | | |
|---|---|---|---|---|---|---|---|
| 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 |
| LAB 3 | | | | LAB 4 | | | |

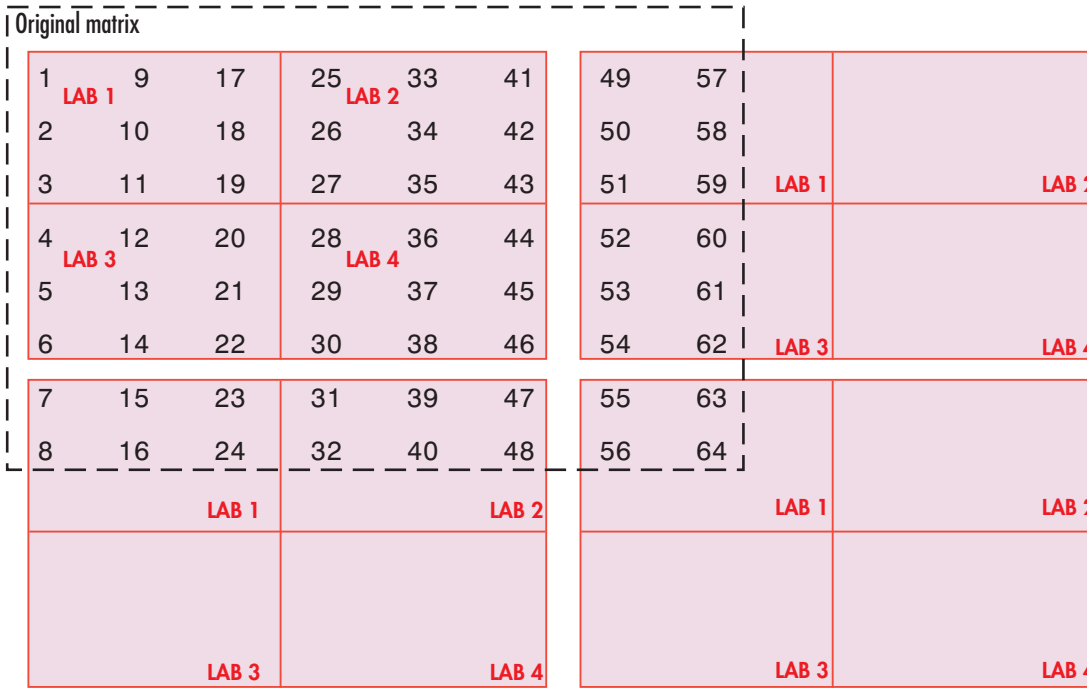
If the lab grid does not perfectly overlay the dimensions of the codistributed array, you can still use `'2dbc'` distribution, which is block cyclic. In this case, you can imagine the lab grid being repeatedly overlaid in both dimensions until all the original matrix elements are included.

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 labs. The code looks like this:

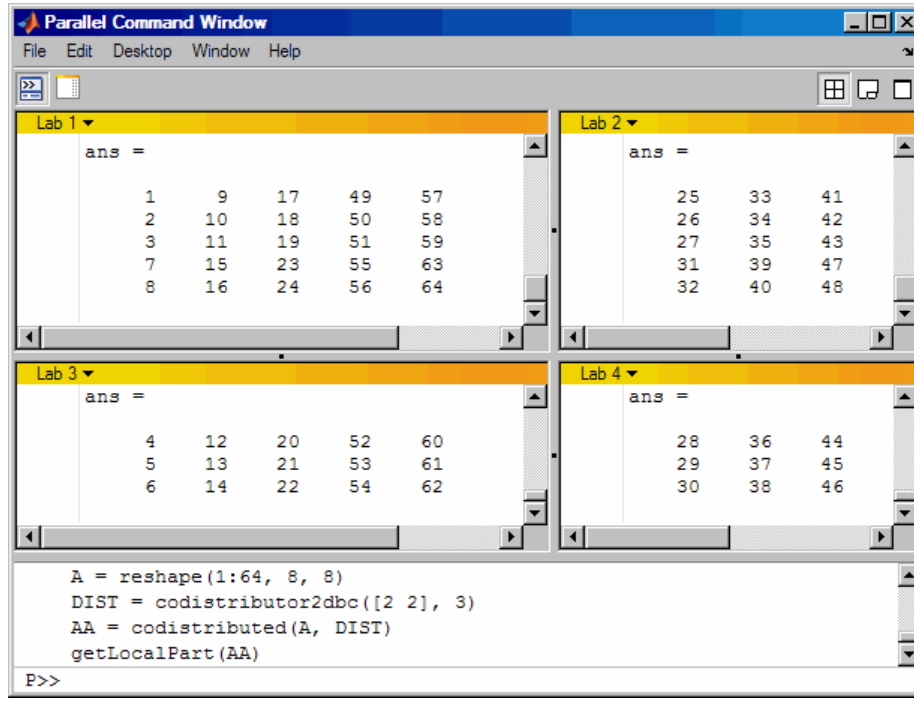
```
P>> DIST = codistributor2dbc([2 2], 3)
P>> AA = codistributed(A, DIST)
```

The first “row” of the lab grid is distributed to lab 1 and lab 2, but that contains only six of the eight columns of the original matrix. Therefore, the next two columns are distributed to lab 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 labs:




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...**

"Parallelizing a for-Loop" on page 5-21

"Codistributed Arrays in a for-drange Loop" on page 5-22

**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 labs. If `results` was simply a replicated array, as it would have been when running the original code in parallel, each lab 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) labs.

When using the `load` function, you need to be careful that the data files are accessible to all labs 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 lab 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 labs, this example uses `gather` to collect the data onto lab 1.

A lab 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 lab performs its portion of the loop, so that the labs are all working simultaneously. Because of this, no communication is allowed between the labs while executing a for-drange loop. In particular, a lab has access only to its partition of a codistributed array. Any calculations in such a loop that

require a lab to access portions of a codistributed array from another lab 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 labs.

```
D = eye(8, 8, codistributor())
E = zeros(8, 8, codistributor())
```

By default, these arrays are distributed by columns; that is, each of the four labs contains two columns of each array. If you use these arrays in a `for-drange` loop, any calculations must be self-contained within each lab. In other words, you can only perform calculations that are limited within each lab to the two columns of the arrays that the labs 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 lab has access to the entire column of `D` and the entire column of `E` necessary to perform the calculation, as each lab 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 labs, so an error occurs, indicating that communication between the labs 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 labs:

```
spmd
  PP = codistributed.zeros(6,8,12);
  RR = rand(6,8,12,codistributor())
  % Default distribution:
  %   by third dimension, evenly across 4 labs.

  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.

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 | `cast`, `cell2mat`, `cell2struct`, `celldisp`, `cellfun`, `char`, `double`, `fieldnames`, `int16`, `int32`, `int64`, `int8`, `logical`, `num2cell`, `rmfield`, `single`, `struct2cell`, `swapbytes`, `typecast`, `uint16`, `uint32`, `uint64`, `uint8` |
| 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`, `isequalwithequalnans`, `isfinite`, `isinf`, `isnan`, `length`, `ndims`, `numel`, `size`, `tril`, `triu` |

| Type of Function | Function Names |
|---|---|
| Matrix functions | `chol`, `eig`, `lu`, `norm`, `normest`, `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` |

6

# 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.


- "Product Introduction" on page 6-2
- "Using Parallel Computing Toolbox Software" on page 6-8
- "Program Development Guidelines" on page 6-12
- "Life Cycle of a Job" on page 6-14
- "Programming with User Configurations" on page 6-16
- "Programming Tips and Notes" on page 6-29
- "Using the Parallel Profiler" on page 6-32
- "Benchmarking Performance" on page 6-44
- "Troubleshooting and Debugging" on page 6-45

# Product Introduction

| In this section... |
| --- |
| "Overview" on page 6-2 |
| "Toolbox and Server Components" on page 6-3 |

## Overview

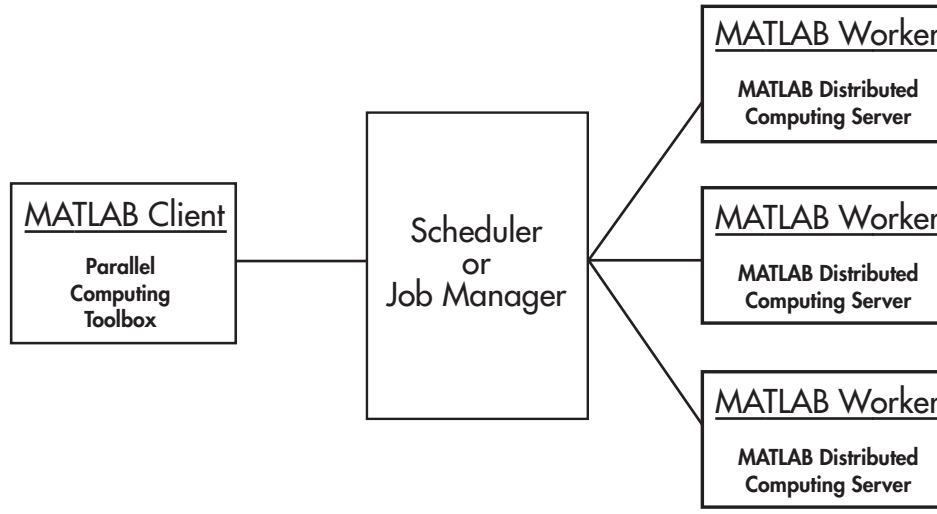
Parallel Computing Toolbox and MATLAB Distributed Computing Server software let you solve computationally and data-intensive problems using MATLAB and Simulink on multicore and multiprocessor computers. Parallel processing constructs such as parallel for-loops and code blocks, distributed arrays, parallel 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. MATLAB Distributed Computing Server software is the product that performs the execution of your job by evaluating each of its tasks and returning the result to your client session.

The *job manager* is the part of the engine that coordinates the execution of jobs and the evaluation of their tasks. The job manager distributes the tasks for evaluation to the server's individual MATLAB sessions called *workers*. Use of the MathWorks™ job manager is optional; the distribution of tasks to workers can also be performed by a third-party scheduler, such as Microsoft® Windows HPC Server (including CCS) or Platform LSF® schedulers.

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




**Basic Parallel Computing Configuration**

## Toolbox and Server Components


- "Job Managers, 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


### Job Managers, Workers, and Clients

The job manager can be run on any machine on the network. The job manager runs jobs in the order in which they are submitted, unless any jobs in its queue are promoted, demoted, canceled, or destroyed.

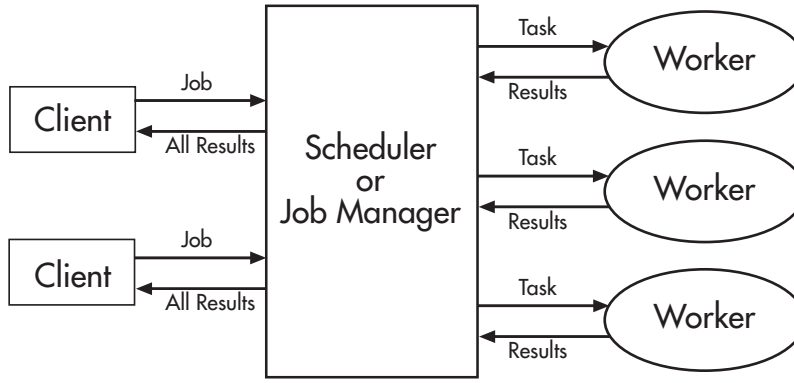
Each worker is given a task from the running job by the job manager, executes the task, returns the result to the job manager, and then is given another task. When all tasks for a running job have been assigned to workers, the job manager starts running the next job with the next available worker.

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. The workers evaluate tasks one at a time, returning the results to the job manager. The job manager then returns the results of all the tasks in the job to the client session.

---

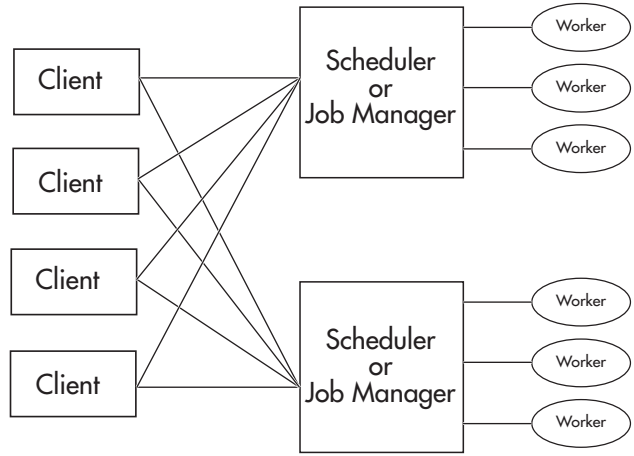
**Note** For testing your application locally or other purposes, you can configure a single computer as client, worker, and job manager. You can also have more than one worker session or more than one job manager session on a machine.

---




**Interactions of Parallel Computing Sessions**

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




**Configuration with Multiple Clients and Job Managers**

## Local Scheduler

A feature of Parallel Computing Toolbox software is the ability to run a local scheduler and up to eight 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 application before running it on your cluster.

## Third-Party Schedulers

As an alternative to using the MathWorks job manager, 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, mpiexec, or a generic scheduler.

### Choosing Between a Third-Party Scheduler and Job Manager.

You should consider the following when deciding to use a scheduler or the MathWorks job manager for distributing your tasks:

- Does your cluster already have a scheduler?

If you already have a scheduler, you may be required to use it as a means of controlling access to the cluster. Your existing scheduler might be just as easy to use as a job manager, 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 MathWorks job manager 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 MathWorks job manager can handle all file and data sharing necessary for your parallel computing applications. This might be helpful in configurations where shared access is limited.

- Are you interested in batch mode or managed interactive processing?

  When you use a job manager, worker processes usually remain running at all times, dedicated to their job manager. 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 may 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 job manager.

- 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 job manager 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, job managers, 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 MathWorks job manager, every machine that hosts a worker or job manager session must also run the mdce service.

The mdce service controls the worker and job manager sessions and recovers them when their host machines crash. If a worker or job manager machine crashes, when the mdce service starts up again (usually configured to start at machine boot time), it automatically restarts the job manager 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 job manager by calling methods and configuring properties of a *job manager 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 job manager or in the scheduler's data 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 job manager or in the scheduler's data location, and you access them through *task objects*.

# Using Parallel Computing Toolbox Software

**In this section...**

"Example: Evaluating a Basic Function" on page 6-8

"Example: Programming a Basic Job with a Local Scheduler" on page 6-8

"Getting Help" on page 6-10

## Example: Evaluating a Basic Function

The `dfeval` function allows you to evaluate a function in a cluster of workers without having to individually define jobs and tasks yourself. When you can divide your job into similar tasks, using `dfeval` might be an appropriate way to run your job. The following code uses a local scheduler on your client computer for `dfeval`.

```
results = dfeval(@sum, {[1 1] [2 2] [3 3]}, 'Configuration', 'local')
results =
    [2]
    [4]
    [6]
```

This example runs the job as three tasks in three separate MATLAB worker sessions, reporting the results back to the session from which you ran `dfeval`.

For more information about `dfeval` and in what circumstances you can use it, see Chapter 7, "Evaluating Functions in a Cluster".

## Example: Programming a Basic Job with a Local Scheduler

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 scheduler. Use `findResource` to indicate that you are using the local scheduler and create the object `sched`, which represents the scheduler. (For more information, see "Find a Job Manager" on page 8-8 or "Creating and Running Jobs" on page 8-20.)

   ```
   sched = findResource('scheduler', 'type', 'local')
   ```

2. Create a job. Create job `j` on the scheduler. (For more information, see "Create a Job" on page 8-10.)

   ```
   j = createJob(sched)
   ```

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 8-11.)

   ```
   createTask(j, @sum, 1, {[1 1]})
   createTask(j, @sum, 1, {[2 2]})
   createTask(j, @sum, 1, {[3 3]})
   ```

4. Submit the job to the scheduler 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 eight at one time. (For more information, see "Submit a Job to the Job Queue" on page 8-12.)

   ```
   submit(j);
   ```

5. Wait for the job to complete, then get the results from all the tasks of the job. (For more information, see "Retrieve the Job's Results" on page 8-13.)

   ```
   waitForState(j)
   results = getAllOutputArguments(j)
   results =
       [2]
       [4]
       [6]
   ```

6 Destroy the job. When you have the results, you can permanently remove the job from the scheduler's data location.

```
destroy(j)
```

## Getting Help


- "Command-Line Help" on page 6-10
- "Help Browser" on page 6-11


### Command-Line Help

You can get command-line help on the toolbox object functions by using the syntax

```
help distcomp.objectType/functionName
```

For example, to get command-line help on the `createTask` function, type

```
help distcomp.job/createTask
```

The available choices for *`objectType`* are listed in the Chapter 10, "Object Reference".

**Listing Available Functions.** To find the functions available for each type of object, type

```
methods(obj)
```

where `obj` is an object of one of the available types.

For example, to see the functions available for job manager objects, type

```
jm = findResource('scheduler','type','jobmanager');
methods(jm)
```

To see the functions available for job objects, type

```
job1 = createJob(jm)
methods(job1)
```

To see the functions available for task objects, type

```
task1 = createTask(job1,1,@rand,{3})
methods(task1)
```

## Help Browser

You can open the Help browser with the `doc` command. To open the browser on a specific reference page for a function or property, type

```
doc distcomp/RefName
```

where *`RefName`* is the name of the function or property whose reference page you want to read.

For example, to open the Help browser on the reference page for the `createJob` function, type

```
doc distcomp/createJob
```

To open the Help browser on the reference page for the `UserData` property, type

```
doc distcomp/UserData
```

# 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 a distributed or parallel job.** If your application involves large data sets on which you need simultaneous calculations performed, you might benefit from a parallel job with distributed arrays. If your application involves looped or repetitive calculations that can be performed independently of each other, a distributed job might be appropriate.

3. **Modify your code for division.** Decide how you want your code divided. For a distributed job, determine how best to divide it into tasks; for example, each iteration of a for-loop might define one task. For a parallel job, determine how best to take advantage of parallel processing; for example, a large array can be distributed across all your labs.

4. **Use pmode to develop parallel functionality.** Use pmode with the local scheduler to develop your functions on several workers (labs) 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 distributed or parallel job with a local scheduler.** Create a parallel or distributed 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 a distributed job, that its computations are properly divided into tasks.

6 **Run the distributed job on only one cluster node.** Run your distributed job with one task to verify that remote distribution is working between your client and the cluster, and to verify file and path dependencies.

7 **Run the distributed or parallel job on multiple cluster nodes.** Scale up your job to include as many tasks as you need for a distributed job, or as many workers (labs) as you need for a parallel job.

---

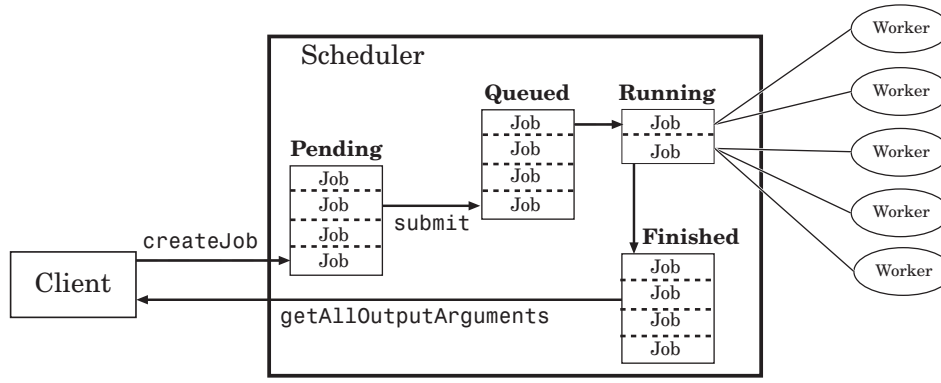
**Note** The client session of MATLAB must be running the Java™ Virtual Machine (JVM™) to use Parallel Computing Toolbox software. Do not start MATLAB with the `-nojvm` flag.

---

# 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 illustrated the stages in the life cycle of a job. In the job manager, 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 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 |

| Job Stage | Description |
|---|---|
| | finished. You can change the order of the jobs in the queue with the `promote` and `demote` functions. |
| Running | When a job reaches the top of the queue, the scheduler distributes the job's tasks to worker sessions for evaluation. The job's state is `running`. If more workers are available than necessary 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, a 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 job manager, 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, it remains in the job manager or `DataLocation` directory, even if you clear all the objects from the client session. The job manager or scheduler keeps all the jobs it has executed, until you restart the job manager in a clean state. Therefore, you can retrieve information from a job at a later time or in another client session, so long as the job manager has not been restarted with the `-clean` option.

To permanently remove completed jobs from the job manager or scheduler's data location, use the `destroy` function.

# Programming with User Configurations

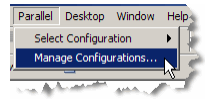
| In this section... |
| --- |
| "Defining Configurations" on page 6-16 |
| "Exporting and Importing Configurations" on page 6-23 |
| "Validating Configurations" on page 6-24 |
| "Applying Configurations in Client Code" on page 6-27 |

## Defining Configurations

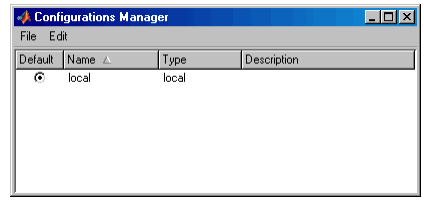
Configurations allow you to define certain parameters and properties, then have your settings applied when creating objects in the MATLAB client. The functions that support the use of configurations are

- `batch` (also supports default configuration)
- `createJob` (also supports default configuration)
- `createMatlabPoolJob` (also supports default configuration)
- `createParallelJob` (also supports default configuration)
- `createTask`
- `dfeval`
- `dfevalasync`
- `findResource`
- `matlabpool` (also supports default configuration)
- `pmode` (also supports default configuration)
- `set`

You create and modify configurations through the Configurations Manager. You access the Configurations Manager using the **Parallel** pull-down menu on the MATLAB desktop. Select **Parallel > Manage Configurations** to open the Configurations Manger.




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

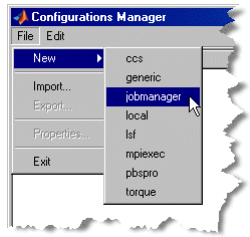



The following example provides instructions on how to create and modify configurations using the Configurations Manager and its menus and dialog boxes.

## Example — Creating and Modifying User Configurations

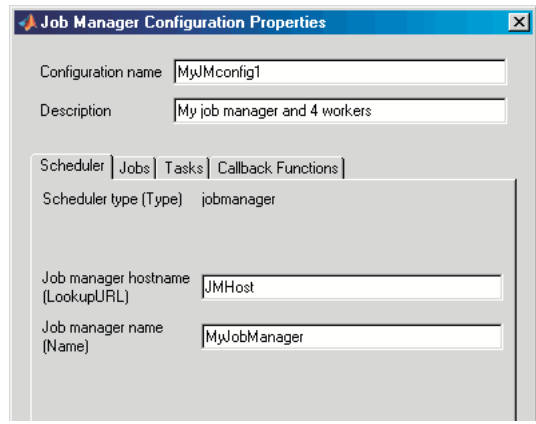
Suppose you want to create a configuration to set several properties for some jobs being run by a job manager.

1 In the Configurations Manager, select **New > jobmanager**. This specifies that you want a new configuration whose type of scheduler is a job manager.




This opens a new Job Manager Configuration Properties dialog box.

2 Enter a configuration name `MyJMconfig1` and a description as shown in the following figure. In the **Scheduler** tab, enter the host name for the machine on which the job manager is running and the name of the job manager. If you are entering information for an actual job manager already running on your network, enter the appropriate text. If you are unsure about job manager names and locations on your network, ask your system administrator for help.

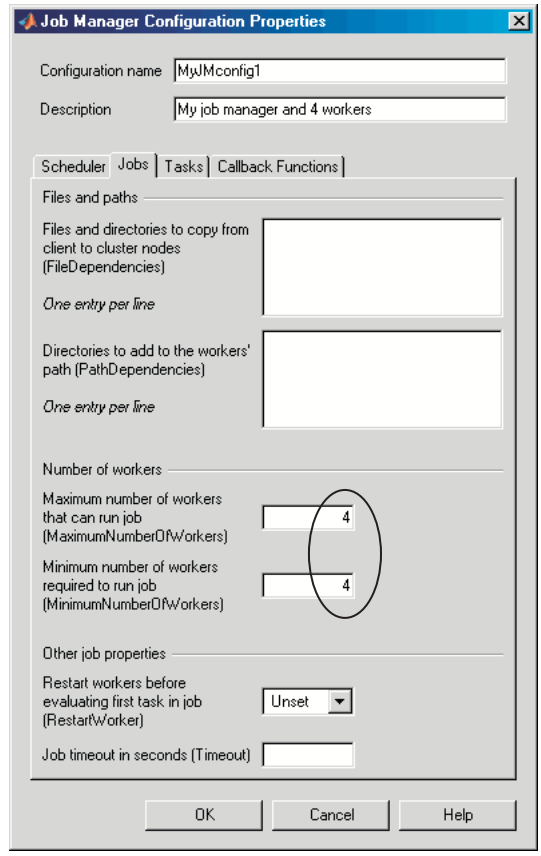



---

**Note** Fields that indicate "Unset" or that you leave empty, have no effect on their property values. For those properties, the configuration does not alter the values that you had set programmatically before applying the configuration.

---

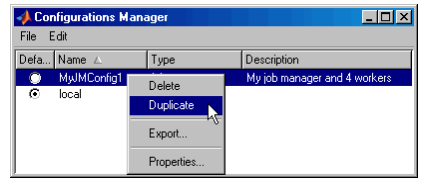
3 In the **Jobs** tab, enter 4 and 4 for the maximum and minimum number of workers. This specifies that for jobs using this configuration, they require at least four workers and use no more than four workers. Therefore, the job runs on exactly four workers, even if it has to wait until four workers are available before starting.




4 Click **OK** to save the configuration and close the dialog box. Your new configuration now appears in the Configurations Manager listing.

5 To create a similar configuration with just a few differences, you can duplicate an existing configuration and modify only the parts you need to change:

   a In the Configurations Manager, right-click the configuration `MyJMconfig1` in the list and select **Duplicate**.



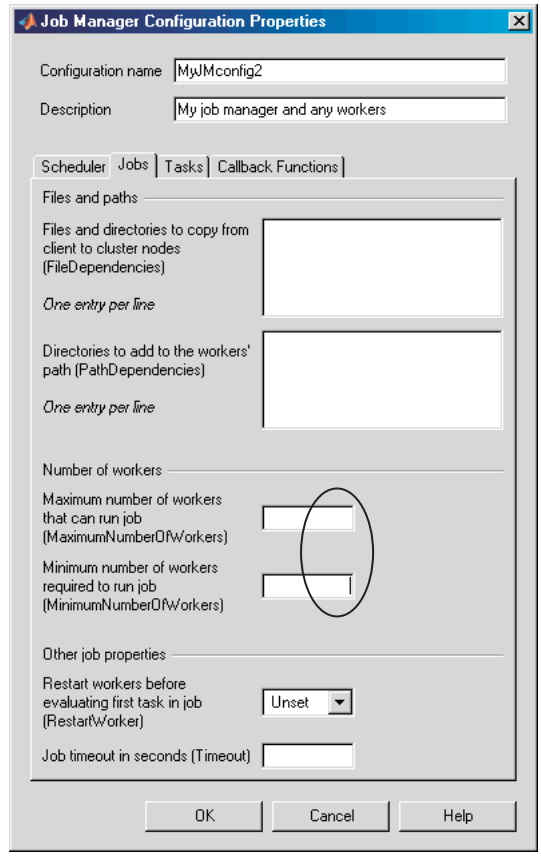

   The duplicate configuration is created with a default name using the original name along with the extension `.copy1`.

   b Double-click the new configuration to open its properties dialog.

   c Change the name of the new configuration to `MyJMconfig2`.

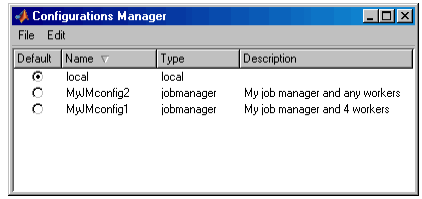
   d Edit the description field to change its text to `My job manager and any workers`.

6 Select the **Jobs** tab. Remove the 4 from each of the fields for minimum and maximum workers.




7 Click **OK** to save the configuration and close the properties dialog.

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




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

## Exporting and Importing Configurations

Parallel configurations are stored as part of your MATLAB preferences, so they are generally available on an individual user basis. To make a parallel configuration available to someone else, you can export it to a separate `.mat` file. In this way, a repository of configurations can be created so that all users of a computing cluster can share common configurations.

To export a parallel configuration:

1. In the Configurations Manager, select (highlight) the configuration you want to export.
2. Click **File > Export**. (Alternatively, you can right-click the configuration in the listing and select **Export**.)
3. In the Export Configuration dialog box, specify a location and name for the file. The default file name is the same as the name of the configuration it contains, with a `.mat` extension appended; these do not need to be the same, so you can alter the names if you want to.

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

1. In the Configuration Manager, click **File > Import**.
2. In the Import Configuration dialog box, browse to find the `.mat` file for the configuration you want to import. Select the file and click **Import**.

   The imported configuration appears in your Configurations Manager list. Note that the list contains the configuration name, which is not necessarily the file name. If you already have a configuration with the same name as the one you are importing, the imported configuration gets an extension added to its name so you can distinguish it.

You can also import configurations programmatically with the `importParallelConfig` function. For details and examples, see the `importParallelConfig` reference page.

### Exporting Configurations for MATLAB Compiler

You can use an exported configuration with MATLAB® Compiler™ to identify cluster setup information for running compiled applications on a cluster. For example, the `setmcruserdata` function can use the exported configuration file name to set the value for the key `ParallelConfigurationFile`. For more information, see "Improving Data Access Using the MCR User Data Interface" in the MATLAB Compiler documentation.

---

**Note** MATLAB Compiler does not support configurations that use the local scheduler or local workers.

---

## Validating Configurations

The Configurations Manager includes a tool for validating configurations.

To validate a configuration, follow these steps:

1. Open the Configurations Manager by selecting on the desktop **Parallel > Manage Configurations**.

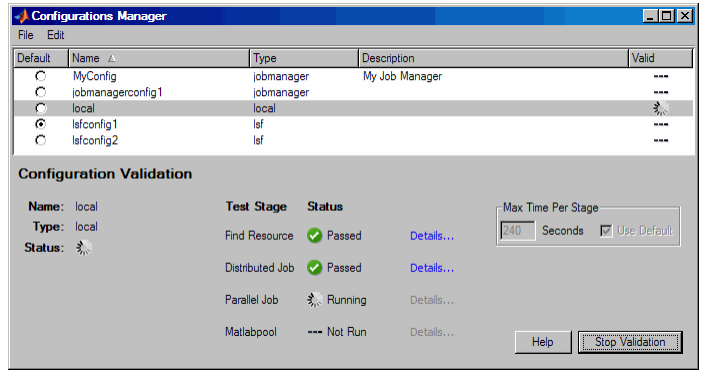
2 In the Configurations Manager, click the name of the configuration you want to test in the the list of those available. Note that you can highlight a configuration this way without changing the selected default configuration. So a configuration selected for validation does not need to be your default configuration.

3 Click **Start Validation**.

The Configuration Validation tool attempts four operations to validate the chosen configuration:

- Uses `findResource` to locate the scheduler
- Runs a distributed job using the configuration
- Runs a parallel job using the configuration
- Runs a MATLAB pool job using the configuration

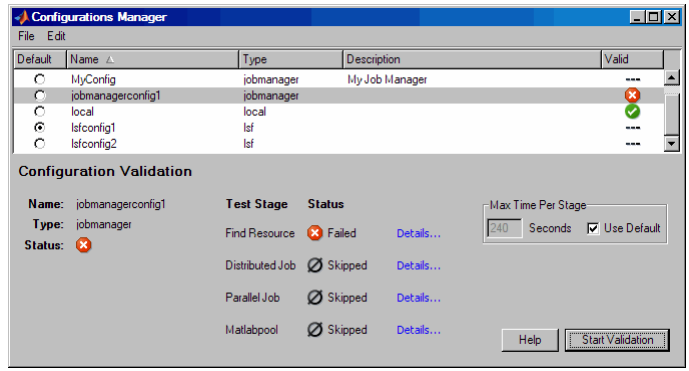
While the tests are running, the Configurations Manager displays their progress as shown here.




You can adjust the timeout allowed for each stage of the testing. If your cluster does not have enough workers available to perform the validation, the test times out and returns a failure.

**Note** You cannot run a configuration validation if you have a MATLAB pool open.

The configuration listing displays the overall validation result for each configuration. The following figure shows overall validation results for one configuration that passed and one that failed. The selected configuration is the one that failed.




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

For each stage of the validation testing, you can click **Details** to get more information about that stage. This information includes any error messages, debug logs, and other data that might be useful in diagnosing problems or helping to determine proper configuration or network settings.

The Configuration Validation tool keeps the test results available until the current MATLAB session closes.

## Applying Configurations in Client Code

In the MATLAB client where you create and define your parallel computing objects, you can use configurations when creating the objects, or you can apply configurations to objects that already exist.

### Selecting a Default Configuration

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

- In the MATLAB desktop, click **Parallel > Select Configuration**, and from there, all your configurations are available. The current default configuration appears with a dot next to it. You can select any configuration on the list as the default.
- In the Configurations Manager, the **Default** column indicates with a radio button which configuration is currently the default configuration. You can click any other button in this column to change the default configuration.
- You can get or set the default configuration programmatically by using the `defaultParallelConfig` function. The following sets of commands achieve the same thing:

```
defaultParallelConfig('MyJMconfig1')
matlabpool open

matlabpool open MyJMconfig1
```

### Finding Schedulers

When executing the `findResource` function, you can use configurations to identify a particular scheduler and apply property values. For example,

```
jm = findResource('scheduler', 'Configuration', 'our_jobmanager')
```

This command finds the scheduler defined by the settings of the configuration named `our_jobmanager` and sets property values on the scheduler object based on settings in the configuration. The advantage of configurations is that you can alter your scheduler choices without changing your MATLAB application code, merely by changing the configuration settings

For a third-party scheduler such as Platform LSF, the command might look like

```
lsfsched = findResource('scheduler', 'Configuration', 'my_lsf_config');
```

## Creating Jobs

Because the properties of scheduler, job, and task objects can be defined in a configuration, you do not have to define them in your application. Therefore, the code itself can accommodate any type of scheduler. For example,

```
job1 = createJob(sched, 'Configuration', 'MyConfig');
```

The configuration defined as `MyConfig` must define any and all properties necessary and appropriate for your scheduler and configuration, and the configuration must not include any parameters inconsistent with your setup. All changes necessary to use a different scheduler can now be made in the configuration, without any modification needed in the application.

## Setting Job and Task Properties

You can set the properties of a job or task with configurations when you create the objects, or you can apply a configuration after you create the object. The following code creates and configures two jobs with the same property values.

```
job1 = createJob(jm, 'Configuration', 'our_jobmanager_config')
job2 = createJob(jm)
set(job2, 'Configuration', 'our_jobmanager_config')
```

Notice that the `Configuration` property of a job indicates the configuration that was applied to the job.

```
get(job1, 'Configuration')
    our_jobmanager_config
```

When you apply a configuration to an object, all the properties defined in that configuration get applied to the object, and the object's `Configuration` property is set to reflect the name of the configuration that you applied. If you later directly change any of the object's individual properties, the object's `Configuration` property is cleared.

# Programming Tips and Notes

**In this section...**

"Saving or Sending Objects" on page 6-29
"Current Working Directory of a MATLAB Worker" on page 6-29
"Using clear functions" on page 6-30
"Running Tasks That Call Simulink Software" on page 6-30
"Using the pause Function" on page 6-30
"Transmitting Large Amounts of Data" on page 6-30
"Interrupting a Job" on page 6-30
"Speeding Up a Job" on page 6-31

## Saving or Sending Objects

Do not use the `save` or `load` function on Parallel Computing Toolbox objects. Some of the information that these objects require is stored in the MATLAB session persistent memory and would not be saved to a file.

Similarly, you cannot send a parallel computing object between parallel computing processes by means of an object's properties. For example, you cannot pass a job manager, job, task, or worker object to MATLAB workers as part of a job's `JobData` property.

## 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
```

## Using clear functions

Executing

```
clear functions
```

clears all Parallel Computing Toolbox objects from the current MATLAB session. They still remain in the job manager. For information on recreating these objects in the client session, see "Recovering Objects" on page 8-17.

## 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.

## 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`, `destroy`, `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 not negligible 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.

# Using the Parallel Profiler

**In this section...**

"Introduction" on page 6-32

"Collecting Parallel Profile Data" on page 6-32

"Viewing Parallel Profile Data" on page 6-33

"Parallel Profiler Demos" on page 6-42

## Introduction

The parallel profiler provides an extension of the `profile` command and the profile viewer specifically for parallel jobs, to enable you to see how much time each lab spends evaluating each function and how much time communicating or waiting for communications with the other 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 parallel 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 parallel 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 parallel 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 lab and the communications between the labs. Such information includes:

- Execution time of each function on each lab
- Execution time of each line of code in each function

- Amount of data transferred between each lab
- Amount of time each lab 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
```

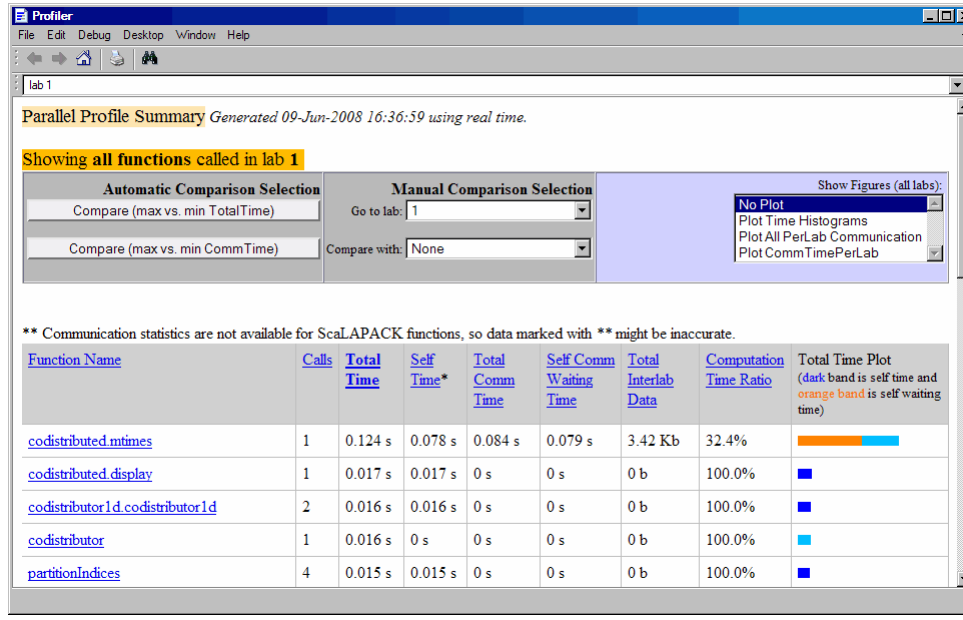
The remainder of this section is an example that illustrates some of the features of the parallel profile viewer. This example executes in a pmode session running on four local labs. Initiate pmode by typing in the MATLAB Command Window:

```
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 lab 1.

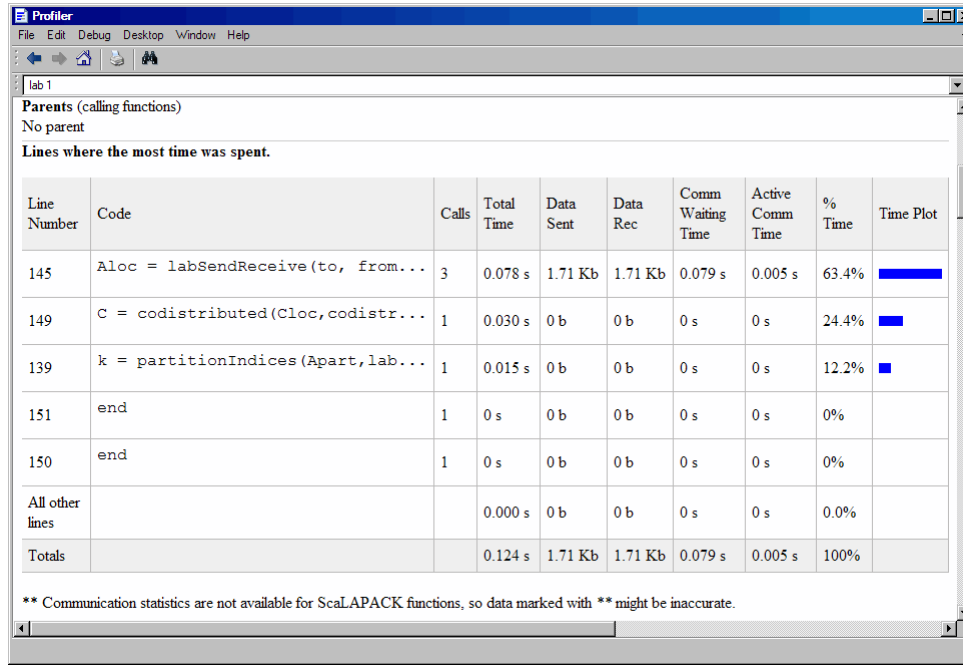



The function summary report displays the data for each function executed on a lab in sortable columns with the following headers:

| Column Header | Description |
|---|---|
| Calls | How many times the function was called on this lab |
| Total Time | The total amount of time this lab spent executing this function |
| Self Time | The time this lab spent inside this function, not within children or subfunctions |
| Total Comm Time | The total time this lab spent transferring data with other labs, including waiting time to receive data |
| Self Comm Waiting Time | The time this lab spent during this function waiting to receive data from other labs |

| Column Header | Description |
|---|---|
| Total Interlab Data | The amount of data transferred to and from this lab 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 lab |

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:

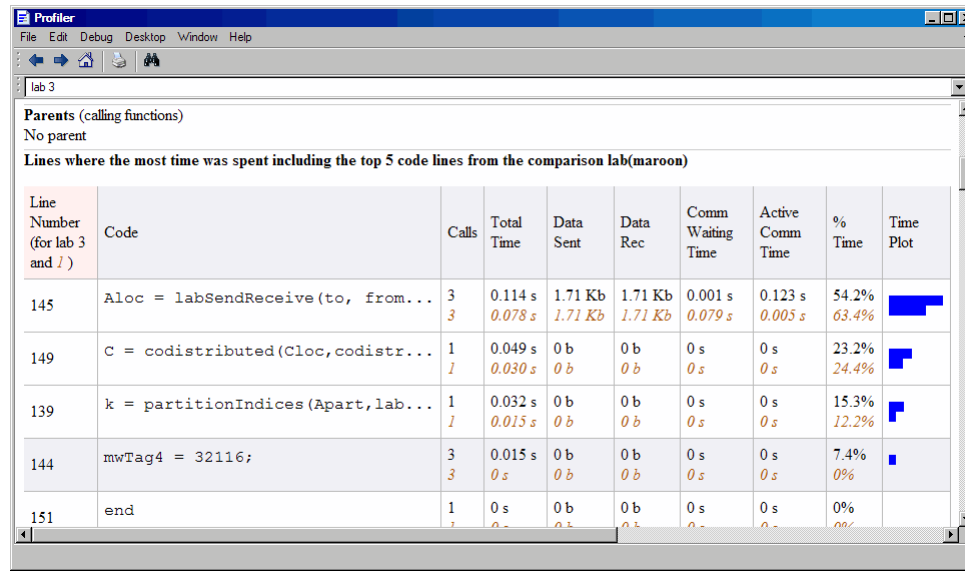



| Line Number | Code | Calls | Total Time | Data Sent | Data Rec | Comm Waiting Time | Active Comm Time | % Time | Time Plot |
|---|---|---|---|---|---|---|---|---|---|
| 145 | `Aloc = labSendReceive(to, from...` | 3 | 0.078 s | 1.71 Kb | 1.71 Kb | 0.079 s | 0.005 s | 63.4% | |
| 149 | `C = codistributed(Cloc,codistr...` | 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% | |

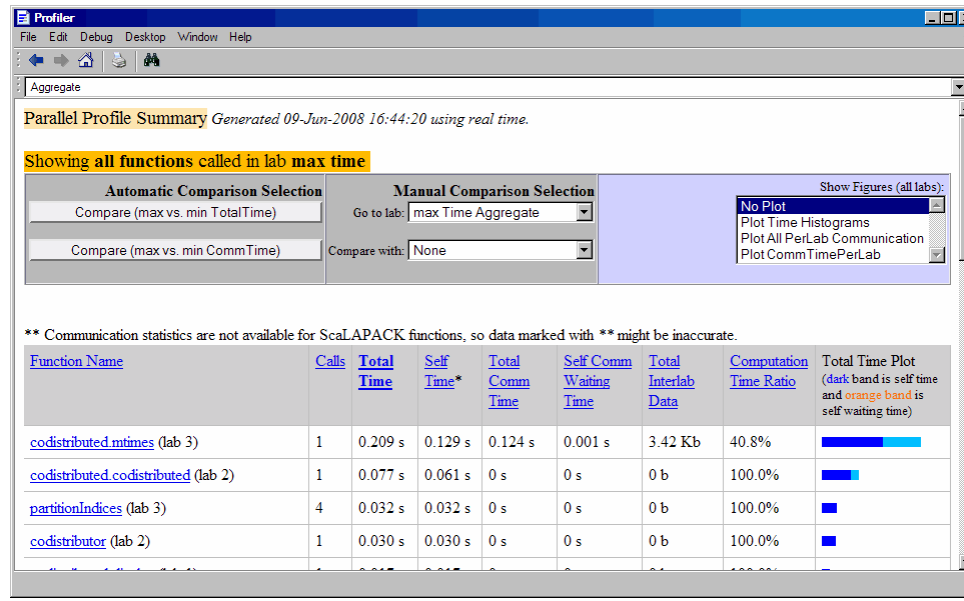
The code that is displayed in the report is taken from the client. If the code has changed on the client since the parallel job ran on the labs, or if the labs are running a different version of the functions, the display might not accurately reflect what actually executed.

You can display information for each lab, or use the comparison controls to display information for several labs simultaneously. Two buttons provide **Automatic Comparison Selection**, allowing you to compare the data from the labs that took the most versus the least amount of time to execute the code, or data from the labs that spent the most versus the least amount of time in performing interlab communication. **Manual Comparison Selection** allows you to compare data from specific labs or labs that meet certain criteria.

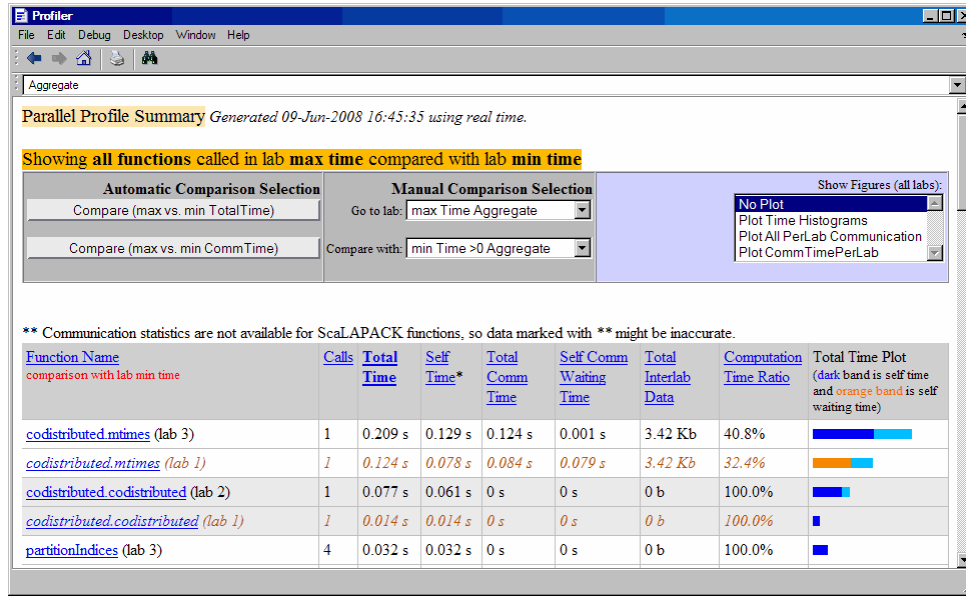
The following listing from the summary report shows the result of using the **Automatic Comparison Selection** of **Compare (max vs. min TotalTime)**. The comparison shows data from lab 3 compared to lab 1 because these are the labs that spend the most versus least amount of time executing the code.



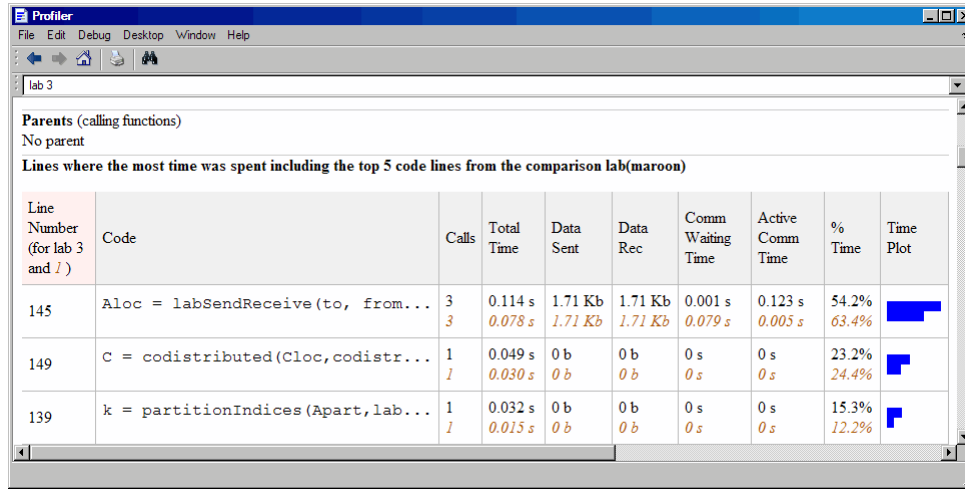
The following figure shows a summary of all the functions executed during the profile collection time. The **Manual Comparison Selection** of **max Time Aggregate** means that data is considered from all the labs for all functions to determine which lab spent the maximum time on each function. Next to each function's name is the lab that took the longest time to execute that function. The other columns list the data from that lab.



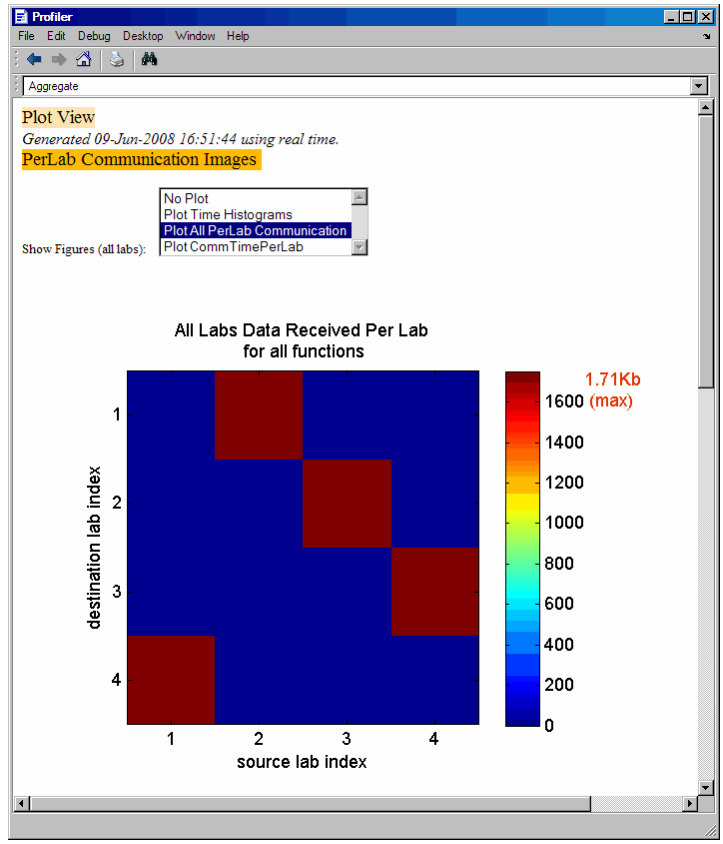
The next figure shows a summary report for the labs that spend the most versus least time for each function. A **Manual Comparison Selection** of **max Time Aggregate** against **min Time >0 Aggregate** generated this summary. Both aggregate settings indicate that the profiler should consider data from all labs for all functions, for both maximum and minimum. This report lists the data for `codistributed.mtimes` from labs 3 and 1, because they spent the maximum and minimum times on this function. Similarly, other functions are listed.



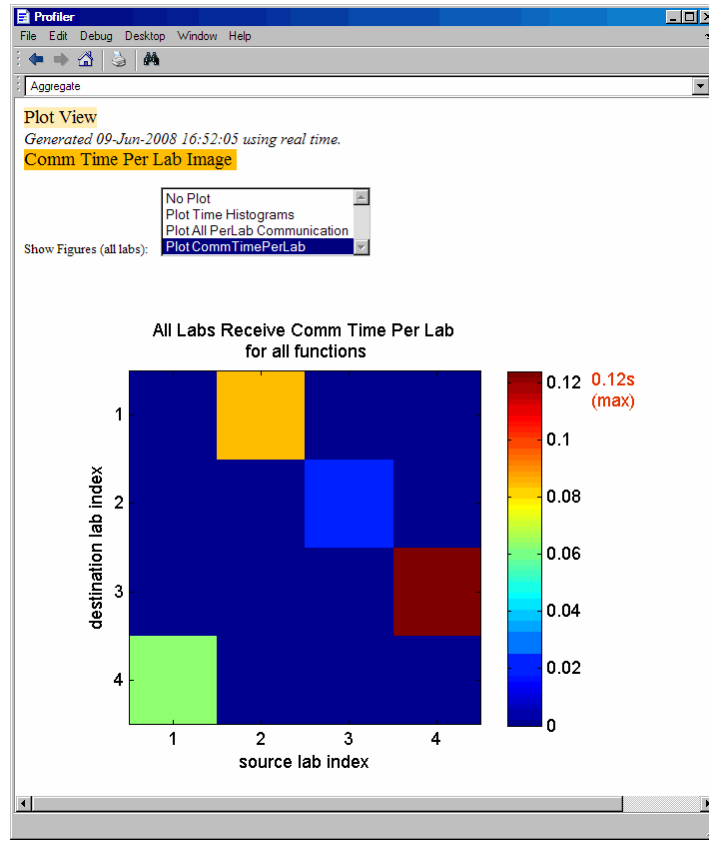
Click on a function name in the summary listing of a comparison to get a detailed comparison. The detailed comparison for `codistributed.mtimes` looks like this, displaying line-by-line data from both labs:



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 lab receives from each other lab for all functions.



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




Plots like those in the previous two figures can help you determine the best way to balance work among your labs, 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" on page 6-44

"HPC Challenge Benchmarks" on page 6-44

## 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" on page 6-45 |
| "File Access and Permissions" on page 6-45 |
| "No Results or Failed Job" on page 6-47 |
| "Connection Problems Between the Client and Job Manager" on page 6-48 |

## 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 a job manager as a scheduler, 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 "Setting 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 `PathDependencies` property. Another solution is to transfer the function file to the worker by adding `function_name.m` to the `FileDependencies` 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 directory?
- Can the worker find the file or directory?
- 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 e-mails 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 data location. The data location may 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., `getAllOutputArguments(job)` returns an empty cell array), it is probable that the job failed and some of its tasks have their `ErrorMessage` and `ErrorIdentifier` 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.

```
sched = findResource('scheduler', 'type', 'lsf')
failedjob = findJob(sched, 'State', 'failed');
message = getDebugLog(sched, failedjob(1))
```

## Connection Problems Between the Client and Job Manager

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 job manager 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 Job Manager

If you cannot locate your job manager with

```
findResource('scheduler','type','jobmanager')
```

the most likely reasons for this failure are

- The client cannot contact the job manager host via multicast. Try to fully specify where to look for the job manager by using the `LookupURL` property in your call to `findResource`:

  ```
  findResource('scheduler','type','jobmanager', ...
                              'LookupURL','JobMgrHostName')
  ```

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

### Job Manager Cannot See the Client

If `findResource` displays a warning message that the job manager cannot open a TCP connection to the client computer, the most likely reasons for this are

- Firewalls do not allow traffic from the job manager to the client.
- The job manager cannot resolve the short hostname of the client computer. Use `pctconfig` to change the hostname that the job manager will use for contacting the client.

7

# Evaluating 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:


- "Evaluating Functions Synchronously" on page 7-2
- "Evaluating Functions Asynchronously" on page 7-8

# Evaluating Functions Synchronously

**In this section...**

"Scope of dfeval" on page 7-2

"Arguments of dfeval" on page 7-3

"Example — Using dfeval" on page 7-4

## 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 "Programming with User Configurations" on page 6-16, 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 — Using 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]
```

# Evaluating 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.

8

# Programming Distributed Jobs

A distributed job is one whose tasks do not directly communicate with 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 distributed jobs:


- "Using a Local Scheduler" on page 8-2
- "Using a Job Manager" on page 8-8
- "Using a Fully Supported Third-Party Scheduler" on page 8-20
- "Using the Generic Scheduler Interface" on page 8-33

# Using a Local Scheduler

**In this section...**

"Creating and Running Jobs with a Local Scheduler" on page 8-2

"Local Scheduler Behavior" on page 8-7

## Creating and Running Jobs with a Local Scheduler

For jobs that require more control than the functionality offered by `dfeval`, you have to program all the steps for creating and running the job. Using the local scheduler lets you create and test your jobs without using the resources of your 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 from a local scheduler 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 scheduler:

- "Create a Scheduler Object" on page 8-3
- "Create a Job" on page 8-3
- "Create Tasks" on page 8-5
- "Submit a Job to the Scheduler" on page 8-5
- "Retrieve the Job's Results" on page 8-6

Note that the objects that the client session uses to interact with the scheduler are only references to data that is actually contained in the scheduler's data location, 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 data

location. You can find existing jobs using the `findJob` function or the `Jobs` property of the scheduler object.

## Create a Scheduler Object

You use the `findResource` function to create an object in your local MATLAB session representing the local scheduler.

```
sched = findResource('scheduler','type','local');
```

## Create a Job

You create a job with the `createJob` function. This statement creates a job in the scheduler's data 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(sched)

Job ID 1 Information
====================

                  UserName : eng864
                     State : pending
                SubmitTime :
                 StartTime :
           Running Duration :

- Data Dependencies

           FileDependencies : {}
           PathDependencies : {}

- Associated Task(s)

            Number Pending  : 0
            Number Running  : 0
            Number Finished : 0
           TaskID of errors :
```

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

```
get(job1)
       Configuration: ''
                Name: 'Job1'
                  ID: 1
            UserName: 'eng864'
                 Tag: ''
               State: 'pending'
          CreateTime: 'Mon Jan 08 15:40:18 EST 2007'
          SubmitTime: ''
           StartTime: ''
          FinishTime: ''
               Tasks: [0x1 double]
    FileDependencies: {0x1 cell}
    PathDependencies: {0x1 cell}
             JobData: []
              Parent: [1x1 distcomp.localscheduler]
            UserData: []
```

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.

```
sched

Local Scheduler Information
===========================

                      Type : local
              ClusterOsType : pc
               DataLocation : C:\WINNT\Profiles\eng864\App...
       HasSharedFilesystem : true

- Assigned Jobs

           Number Pending  : 1
           Number Queued   : 0
           Number Running  : 0
           Number Finished : 0
```

```
- Local Specific Properties

        ClusterMatlabRoot : D:\apps\matlab
```

## 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')
ans =
    Tasks: 5 by 1
    =============

 Task ID       State      End Time     Function Name   Error
 ------------------------------------------------------------
       1     pending                          @rand
       2     pending                          @rand
       3     pending                          @rand
       4     pending                          @rand
       5     pending                          @rand
```

## Submit a Job to the Scheduler

To run your job and have its tasks evaluated, you submit the job to the scheduler with the `submit` function.

```
submit(job1)
```

The local scheduler starts up to eight workers and distributes the tasks of `job1` to its workers for evaluation.

### 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. After waiting for the job to complete, use the function `getAllOutputArguments` to retrieve the results from all the tasks in the job.

```
waitForState(job1)
results = getAllOutputArguments(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 scheduler, job, and task objects:

```
sched
job1
get(job1,'Tasks')
```

## Local Scheduler Behavior

The local scheduler runs in the MATLAB client session, so you do not have to start any separate scheduler process for the local scheduler. When you submit a job for evaluation to the local scheduler, the scheduler starts a MATLAB worker for each task in the job, but only up to as many workers as the scheduler is configured to allow. 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 configuration, up to a maximum of eight. If not configured, the default is to run only as many workers as computational cores on the machine.

The local scheduler has no interaction with any other scheduler, 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 eight 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.

# Using a Job Manager

**In this section...**

"Creating and Running Jobs with a Job Manager" on page 8-8

"Sharing Code" on page 8-13

"Managing Objects in the Job Manager" on page 8-16

## Creating and Running Jobs with a Job Manager

For jobs that are more complex or require more control than the functionality offered by `dfeval`, you have to program all the steps for creating and running of the job.

This section details the steps of a typical programming session with Parallel Computing Toolbox software using a MathWorks job manager:

- "Find a Job Manager" on page 8-8
- "Create a Job" on page 8-10
- "Create Tasks" on page 8-11
- "Submit a Job to the Job Queue" on page 8-12
- "Retrieve the Job's Results" on page 8-13

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

### Find a Job Manager

You use the `findResource` function to identify available job managers and to create an object representing a job manager in your local MATLAB session.

To find a specific job manager, use parameter-value pairs for matching. In this example, `MyJobManager` is the name of the job manager, while `MyJMhost` is the hostname of the machine running the job manager lookup service.

```
jm = findResource('scheduler','type','jobmanager', ...
                    'Name','MyJobManager','LookupURL','MyJMhost');
get(jm)
          Configuration: ''
                   Name: 'MyJobManager'
               Hostname: 'bonanza'
            HostAddress: {'123.123.123.123'}
                   Type: 'jobmanager'
          ClusterOsType: 'pc'
                   Jobs: [0x1 double]
                  State: 'running'
               UserData: []
            ClusterSize: 2
    NumberOfBusyWorkers: 0
            BusyWorkers: [0x1 double]
    NumberOfIdleWorkers: 2
            IdleWorkers: [2x1 distcomp.worker]
```

If your network supports multicast, you can omit property values to search on, and `findResource` returns all available job managers.

```
all_managers = findResource('scheduler','type','jobmanager')
```

You can then examine the properties of each job manager to identify which one you want to use.

```
for i = 1:length(all_managers)
  get(all_managers(i))
end
```

When you have identified the job manager you want to use, you can isolate it and create a single object.

```
jm = all_managers(3)
```

### Create a Job

You create a job with the `createJob` function. Although you execute this command in the client session, the job is actually created on the job manager.

```
job1 = createJob(jm)
```

This statement creates a job on the job manager `jm`, and creates the job object `job1` in the client session. Use `get` to see the properties of this job object.

```
get(job1)
             Configuration: ''
                      Name: 'job_3'
                        ID: 3
                  UserName: 'eng864'
                       Tag: ''
                     State: 'pending'
             RestartWorker: 0
                   Timeout: Inf
    MaximumNumberOfWorkers: 2.1475e+009
    MinimumNumberOfWorkers: 1
                CreateTime: 'Thu Oct 21 19:38:08 EDT 2004'
                SubmitTime: ''
                 StartTime: ''
                FinishTime: ''
                     Tasks: [0x1 double]
          FileDependencies: {0x1 cell}
          PathDependencies: {0x1 cell}
                   JobData: []
                    Parent: [1x1 distcomp.jobmanager]
                  UserData: []
                 QueuedFcn: []
                RunningFcn: []
              FinishedFcn: []
```

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

```
get(jm)
```

```
          Configuration: ''
                   Name: 'MyJobManager'
               Hostname: 'bonanza'
            HostAddress: {'123.123.123.123'}
                   Type: 'jobmanager'
          ClusterOsType: 'pc'
                   Jobs: [1x1 distcomp.job]
                  State: 'running'
               UserData: []
            ClusterSize: 2
    NumberOfBusyWorkers: 0
            BusyWorkers: [0x1 double]
    NumberOfIdleWorkers: 2
            IdleWorkers: [2x1 distcomp.worker]
```

You can transfer files to the worker by using the `FileDependencies` property of the job object. For details, see the `FileDependencies` reference page and "Sharing Code" on page 8-13.

## Create Tasks

After you have created your job, you can create tasks for the job using the `createTask` function. Tasks define the functions to be evaluated by the workers during the running of the job. Often, the tasks of a job are all identical. In this example, each task will generate a 3-by-3 matrix of random numbers.

```
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
createTask(job1, @rand, 1, {3,3});
```

The `Tasks` property of `job1` is now a 5-by-1 matrix of task objects.

```
get(job1,'Tasks')
ans =
    distcomp.task: 5-by-1
```

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 `FileDependencies` and `PathDependencies` 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 `FileDependencies` or `PathDependencies`. If the same worker evaluates subsequent tasks for this job, `jobStartup` does not run between tasks.
3. Run the `taskStartup` function. You can specify this function in `FileDependencies` or `PathDependencies`. 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 `getAllOutputArguments` to retrieve the results from all the tasks in the job.

```
results = getAllOutputArguments(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
```

## Sharing 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:

- "Directly Accessing Files" on page 8-14

- "Passing Data Between Sessions" on page 8-14
- "Passing MATLAB Code for Startup and Finish" on page 8-15


## Directly Accessing Files

If the workers all have access to the same drives on the network, they can access needed 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 path so that it looks for files in the right places. You can define the path

- By using the job's `PathDependencies` 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 `FileDependencies` or `PathDependencies` 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 job manager 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 "Defining the Script Defaults" in the MATLAB Distributed Computing Server System Administrator's Guide.

## Passing Data Between Sessions

A number of properties on task and job objects are designed for passing code or data from client to job manager 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. All these properties are described in detail in their own reference pages:

- `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.
- `FileDependencies` — This property of the job object lists all the directories and files that get zipped and sent to the workers. At the worker, the data is unzipped, and the entries defined in the property are added to the path of the MATLAB worker session.
- `PathDependencies` — This property of the job object provides pathnames that are added to the MATLAB workers' 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 `FileDependencies`. 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 6-45.

## Passing 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 directory on the worker's MATLAB 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 directory

```
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 `FileDependencies` property, or include the path names to their locations in the `PathDependencies` property.

The worker gives precedence to the versions provided in the `FileDependencies` property, then to those pointed to in the `PathDependencies` 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` directory of the worker's MATLAB installation.

## Managing Objects in the Job Manager

Because all the data of jobs and tasks resides in the job manager, 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 8-17
- "Recovering Objects" on page 8-17
- "Resetting Callback Properties" on page 8-18
- "Permanently Removing Objects" on page 8-18

## 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 remain in place. Job objects and task objects reside on the job manager. Local objects in the client session can refer to job managers, jobs, tasks, and workers. When the client session ends, only these local reference objects are lost, not the actual objects in the engine.

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 job manager. The job manager maintains its job and task objects. You can retrieve the job results later in another client session.

## Recovering 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 job manager and worker objects in the client session by using the `findResource` function. These client objects refer to sessions running in the engine.

```
jm = findResource('scheduler','type','jobmanager', ...
             'Name','Job_Mgr_123','LookupURL','JobMgrHost')
```

If your network supports multicast, you can find all available job managers by omitting any specific property information.

```
jm_set = findResource('scheduler','type','jobmanager')
```

The array `jm_set` contains all the job managers accessible from the client session. You can index through this array to determine which job manager is of interest to you.

```
jm = jm_set(2)
```

When you have access to the job manager by the object `jm`, you can create objects that reference all those objects contained in that job manager. All the jobs contained in the job manager are accessible in its `Jobs` property, which is an array of job objects.

```
all_jobs = get(jm,'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 job manager for particular job identified by any of its properties, such as its `State`.

```
finished_jobs = findJob(jm,'State','finished')
```

This command returns an array of job objects that reference all finished jobs on the job manager `jm`.

## Resetting Callback Properties

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.

## Permanently Removing Objects

Jobs in the job manager continue to exist even after they are finished, and after the job manager is stopped and restarted. The ways to permanently remove jobs from the job manager are explained in the following sections:


- "Destroying Selected Objects" on page 8-18
- "Starting a Job Manager from a Clean State" on page 8-19


**Destroying Selected Objects.** From the command line in the MATLAB client session, you can call the `destroy` function for any job or task object. If you destroy a job, you destroy all tasks contained in that job.

For example, find and destroy all finished jobs in your job manager that belong to the user `joep`.

```
jm = findResource('scheduler','type','jobmanager', ...
           'Name','MyJobManager','LookupURL','JobMgrHost')
finished_jobs = findJob(jm,'State','finished','UserName','joep')
```

```
destroy(finished_jobs)
clear finished_jobs
```

The `destroy` function permanently removes these jobs from the job manager. The `clear` function removes the object references from the local MATLAB workspace.

**Starting a Job Manager from a Clean State.** When a job manager starts, by default it starts so that it resumes its former session with all jobs intact. Alternatively, a job manager 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 job manager of the specified name on a particular host.

As a network administration feature, the `-clean` flag of the job manager startup script is described in "Starting in a Clean State" in the MATLAB Distributed Computing Server System Administrator's Guide.

# Using a Fully Supported Third-Party Scheduler

**In this section...**

"Creating and Running Jobs" on page 8-20

"Sharing Code" on page 8-28

"Managing Objects" on page 8-30

## Creating and Running Jobs

If your network already uses Platform LSF (Load Sharing Facility), Microsoft Windows HPC Server (including CCS), PBS Pro, or a TORQUE scheduler, you can use Parallel Computing Toolbox software to create jobs to be distributed by your existing scheduler. This section provides instructions for using your scheduler.

This section details the steps of a typical programming session with Parallel Computing Toolbox software for jobs distributed to workers by a fully supported third-party scheduler.

This section assumes you have an 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`.

The following sections illustrate how to program Parallel Computing Toolbox software to use these schedulers:

- "Find an LSF, PBS Pro, or TORQUE Scheduler" on page 8-21
- "Find a Windows HPC Server Scheduler" on page 8-22
- "Create a Job" on page 8-24
- "Create Tasks" on page 8-25
- "Submit a Job to the Job Queue" on page 8-26
- "Retrieve the Job's Results" on page 8-27

## Find an LSF, PBS Pro, or TORQUE Scheduler

You use the `findResource` function to identify the type of scheduler and to create an object representing the scheduler in your local MATLAB client session.

You specify the scheduler type for `findResource` to search for with one of the following:

```
sched = findResource('scheduler','type','lsf')
sched = findResource('scheduler','type','pbspro')
sched = findResource('scheduler','type','torque')
```

You set properties on the scheduler object to specify

- Where the job data is stored
- That the workers should access job data directly in a shared file system
- The MATLAB root for the workers to use

```
set(sched, 'DataLocation', '\\share\scratch\jobdata')
set(sched, 'HasSharedFilesystem', true)
set(sched, 'ClusterMatlabRoot', '\\apps\matlab\')
```

Alternatively, you can use a parallel configuration to find the scheduler and set the object properties with a single `findResource` statement.

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`.

---

**Note** In a shared file system, all nodes require access to the directory specified in the scheduler object's `DataLocation` directory. See the `DataLocation` reference page for information on setting this property for a mixed-platform environment.

---

You can look at all the property settings on the scheduler object. If no jobs are in the `DataLocation` directory, the `Jobs` property is a 0-by-1 array.

```
get(sched)
                      Configuration: ''
                               Type: 'lsf'
                       DataLocation: '\\share\scratch\jobdata'
                 HasSharedFilesystem: 1
                               Jobs: [0x1 double]
                  ClusterMatlabRoot: '\\apps\matlab\'
                      ClusterOsType: 'unix'
                           UserData: []
                        ClusterSize: Inf
                        ClusterName: 'CENTER_MATRIX_CLUSTER'
                         MasterName: 'masterhost.clusternet.ourdomain.com'
                   SubmitArguments: ''
   ParallelSubmissionWrapperScript: [1x92 char]
```

## Find a Windows HPC Server Scheduler

You use the `findResource` function to identify the Windows HPC Server scheduler and to create an object representing the scheduler in your local MATLAB client session.

You specify `'hpcserver'` as the scheduler type for `findResource` to search for.

```
sched = findResource('scheduler','type','hpcserver')
```

You set properties on the scheduler object to specify

- Where the job data is stored
- The MATLAB root for the workers to use
- The name of the scheduler host
- Cluster version, and whether to use SOA job submission (available only on Microsoft Windows HPC Server 2008).

```
set(sched, 'DataLocation', '\\share\scratch\jobdata');
set(sched, 'ClusterMatlabRoot', '\\apps\matlab\');
set(sched, 'SchedulerHostname', 'server04');
```

```
set(sched, 'ClusterVersion', 'HPCServer2008');
set(sched, 'UseSOAJobSubmission', false);
```

Alternatively, you can use a parallel configuration to find the scheduler and set the object properties with a single `findResource` statement.

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`.

---

**Note** Because Windows HPC Server requires a shared file system, all nodes require access to the directory specified in the scheduler object's `DataLocation` directory.

---

You can look at all the property settings on the scheduler object. If no jobs are in the `DataLocation` directory, the `Jobs` property is a 0-by-1 array.

```
get(sched)
          Configuration: ''
                   Type: 'hpcserver'
           DataLocation: '\\share\scratch\jobdata'
    HasSharedFilesystem: 1
                   Jobs: [0x1 double]
      ClusterMatlabRoot: '\\apps\matlab\'
          ClusterOsType: 'pc'
               UserData: []
            ClusterSize: Inf
      SchedulerHostname: 'server04'
    UseSOAJobSubmission: 0
            JobTemplate: ''
     JobDescriptionFile: ''
         ClusterVersion: 'HPCServer2008'
```

## 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 directory specified by the scheduler object's `DataLocation` property.

```
j = createJob(sched)
```

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

```
get(j)
       Configuration: ''
                Name: 'Job1'
                  ID: 1
            UserName: 'eng1'
                 Tag: ''
               State: 'pending'
          CreateTime: 'Fri Jul 29 16:15:47 EDT 2005'
          SubmitTime: ''
           StartTime: ''
          FinishTime: ''
               Tasks: [0x1 double]
    FileDependencies: {0x1 cell}
    PathDependencies: {0x1 cell}
             JobData: []
              Parent: [1x1 distcomp.lsfscheduler]
            UserData: []
```

This output varies only slightly between jobs that use LSF and Windows HPC Server schedulers, but is quite different from a job that uses a job manager. For example, jobs on LSF or Windows HPC Server schedulers have 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 scheduler's `Jobs` property is now a 1-by-1 array of `distcomp.simplejob` objects, indicating the existence of your job.

```
get(sched, 'Jobs')
```

```
    Jobs: [1x1 distcomp.simplejob]
```

You can transfer files to the worker by using the `FileDependencies` property of the job object. Workers can access shared files by using the `PathDependencies` property of the job object. For details, see the `FileDependencies` and `PathDependencies` reference pages and "Sharing Code" on page 8-28.

---

**Note** 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 computer at a time.

---

## 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 all identical except for different arguments or data. In this example, each task will generate 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')
ans =
    distcomp.simpletask: 5-by-1
```

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 scheduler's job queue.

```
submit(j)
```

The scheduler distributes the tasks of job `j` to MATLAB workers for evaluation. For each task, the scheduler starts a MATLAB worker session on a worker node; this MATLAB worker session runs for only as long as it takes to evaluate the one task. If the same node evaluates another task in the same job, it does so with a different MATLAB worker session.

Each worker performs the following steps for task evaluation:

1. Receive `FileDependencies` and `PathDependencies` from the job. Place files and modify the path accordingly.
2. Run the `jobStartup` function. You can specify this function in `FileDependencies` or `PathDependencies`.
3. Run the `taskStartup` function. You can specify this function in `FileDependencies` or `PathDependencies`.

   If you have enabled `UseSOAJobSubmission` with HPC Server 2008, the scheduler can use a worker to evaluate multiple tasks in sequence. In this case, the worker runs `taskStartup` before evaluating each task, without rerunning `jobStartup` or receiving dependencies again.
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.

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

```
waitForState(j)
```

The default state to wait for is `finished`. This function causes MATLAB to pause until the `State` property of `j` is `'finished'`.

---

**Note** 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.

---

## 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 `getAllOutputArguments` to retrieve the results from all the tasks in the job.

```
results = getAllOutputArguments(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
```

## Sharing Code

Because different machines evaluate the tasks of a job, each machine must have access to all the files needed to evaluate its tasks. The following sections explain the basic mechanisms for sharing data:


- "Directly Accessing Files" on page 8-28
- "Passing Data Between Sessions" on page 8-29
- "Passing MATLAB Code for Startup and Finish" on page 8-29


### Directly Accessing Files

If all the workers have access to the same drives on the network, they can access needed 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 path so that it looks for files in the correct places. You can define the path by

- Using the job's `PathDependencies` property. This is the preferred method for setting the path, because it is specific to the job.
- 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 `FileDependencies` or `PathDependencies` property. Otherwise, the version of each of these files that is used is the one highest on the worker's path.

## Passing Data Between Sessions

A number of properties on task and job objects are for passing code or data from client to scheduler or 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. All these properties are described in detail in their own reference pages:

- `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 depending on file caching, the data might be passed to a worker node only once per job, saving time if that node is evaluating more than one task for the job.
- `FileDependencies` — This property of the job object lists all the directories and files that get zipped and sent to the workers. At the worker, the data is unzipped, and the entries defined in the property are added to the path of the MATLAB worker session.
- `PathDependencies` — This property of the job object provides pathnames that are added to the MATLAB workers' path, reducing the need for data transfers in a shared file system.

## Passing 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 directory on the worker's MATLAB path, such as `toolbox/distcomp/user`.

Three additional files can initialize and clean 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 directory

```
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 `FileDependencies` property, or include the pathnames to their locations in the `PathDependencies` property.

The worker gives precedence to the versions provided in the `FileDependencies` property, then to those pointed to in the `PathDependencies` 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` directory of the worker's MATLAB installation.

## Managing Objects

Objects that the client session uses to interact with the scheduler are only references to data that is actually contained in the directory specified by the `DataLocation` property. After jobs and tasks are created, you can shut down your client session, restart it, and your job will still be stored in that remote location. You can find existing jobs using the `Jobs` property of the recreated scheduler object.

The following sections describe how to access these objects and how to permanently remove them:

- "What Happens When the Client Session Ends?" on page 8-31
- "Recovering Objects" on page 8-31

• "Destroying Jobs" on page 8-32

## 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, job and task data remains in the directory identified by `DataLocation`. When the client session ends, only its local reference objects are lost, not the data of the scheduler.

Therefore, if you have submitted your job to the scheduler job queue for execution, you can quit your client session of MATLAB, and the job will be executed by the scheduler. The scheduler maintains its job and task data. You can retrieve the job results later in another client session.

## Recovering Objects

A client session of Parallel Computing Toolbox software can access any of the objects in the `DataLocation`, whether the current client session or another client session created these objects.

You create scheduler objects in the client session by using the `findResource` function.

```
sched = findResource('scheduler', 'type', 'LSF');
set(sched, 'DataLocation', '/share/scratch/jobdata');
```

When you have access to the scheduler by the object `sched`, you can create objects that reference all the data contained in the specified location for that scheduler. All the job and task data contained in the scheduler data location are accessible in the scheduler object's `Jobs` property, which is an array of job objects.

```
all_jobs = get(sched, '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 scheduler object for a particular job identified by any of its properties, such as its `State`.

```
finished_jobs = findJob(sched, 'State', 'finished')
```

This command returns an array of job objects that reference all finished jobs on the scheduler `sched`, whose data is found in the specified `DataLocation`.

## Destroying Jobs

Jobs in the scheduler continue to exist even after they are finished. From the command line in the MATLAB client session, you can call the `destroy` function for any job object. If you destroy a job, you destroy all tasks contained in that job. The job and task data is deleted from the `DataLocation` directory.

For example, find and destroy all finished jobs in your scheduler whose data is stored in a specific directory.

```
sched = findResource('scheduler', 'name', 'LSF');
set(sched, 'DataLocation', '/share/scratch/jobdata');
finished_jobs = findJob(sched, 'State', 'finished');
destroy(finished_jobs);
clear finished_jobs
```

The `destroy` function in this example permanently removes from the scheduler data those finished jobs whose data is in `/apps/data/project_88`. The `clear` function removes the object references from the local MATLAB client workspace.

# Using the Generic Scheduler Interface

**In this section...**

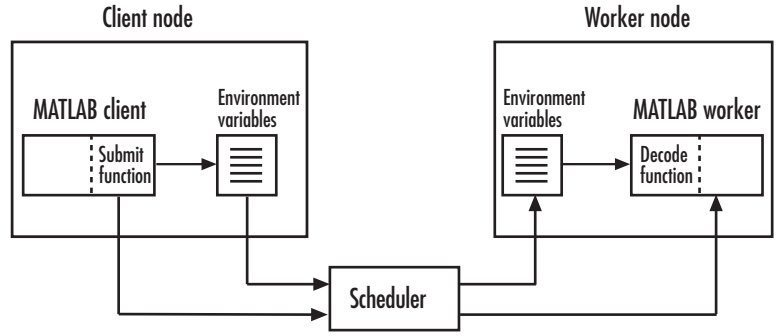
"Overview" on page 8-33
"MATLAB Client Submit Function" on page 8-34
"Example — Writing the Submit Function" on page 8-38
"MATLAB Worker Decode Function" on page 8-39
"Example — Writing the Decode Function" on page 8-41
"Example — Programming and Running a Job in the Client" on page 8-42
"Supplied Submit and Decode Functions" on page 8-47
"Managing Jobs" on page 8-48
"Summary" on page 8-51

## 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 a MathWorks job manager 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 scheduler, the function identified by the scheduler object's `SubmitFcn` property executes in the MATLAB client session. You set the scheduler's `SubmitFcn` 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(sched, 'SubmitFcn', @mysubmitfunc)
```

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

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

Inside the function of this example, the three argument objects are known as `scheduler`, `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 `SubmitFcn` property.

```
time_limit = 300
testlocation = 'Plant30'
set(sched, 'SubmitFcn', {@mysubmitfunc, time_limit, testlocation})
```

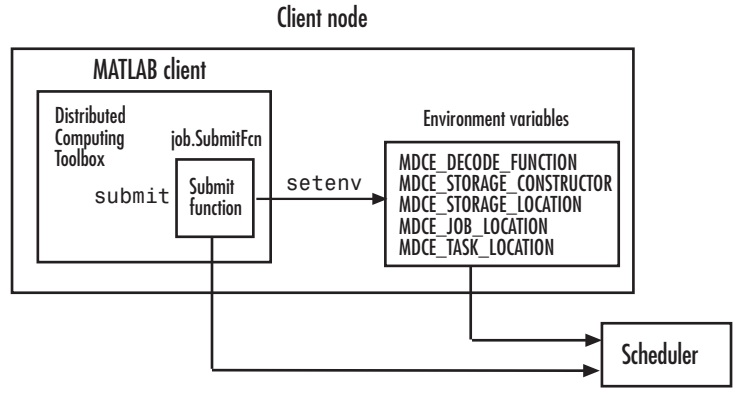
In this example, the submit function requires five arguments: the three defaults, along with the numeric value of `time_limit` and the string value of `testlocation`. The function's declaration line might look like this:

```
function mysubmitfunc(scheduler, 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




## Identifying 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 8-39.

## Passing Job and Task Data

The third input argument (after scheduler 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 scheduler DataLocation 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.

## Defining 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 |
|---|---|---|
| Condor® | `condor_submit` | Not by default. Command can pass all or specific 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. `scheduler`, `job`, and `props` (so named only for this example) refer to the first three arguments to the submit function.

| Argument Object | Property |
|---|---|
| `scheduler` | `MatlabCommandToRun` |
| `scheduler` | `ClusterMatlabRoot` |
| `job` | `MinimumNumberOfWorkers` |

| Argument Object | Property |
|---|---|
| job | MaximumNumberOfWorkers |
| props | NumberOfTasks |

## Example — Writing 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 scheduler object, the job object, and the props object.

```
function mysubmitfunc(scheduler, 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 8-34.

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.

**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.

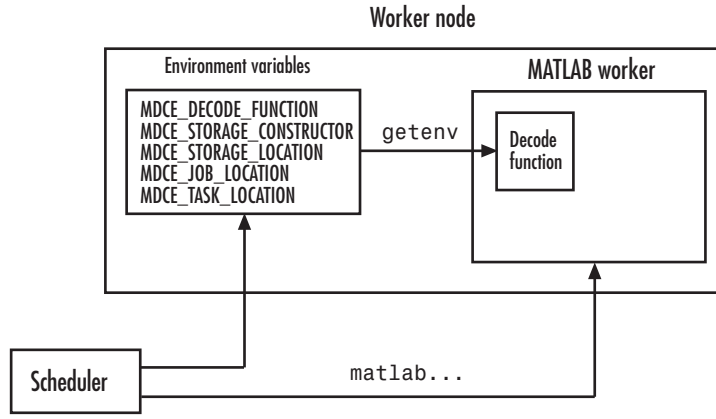
---

## 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




## Identifying 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 8-34. 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 `PathDependencies` or `FileDependencies` 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 directory 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 directory 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.

### Reading 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 — Writing 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 — Programming and Running a Job in the Client

### 1. Create a Scheduler Object

You use the `findResource` function to create an object representing the scheduler in your local MATLAB client session.

You can specify `'generic'` as the `name` for `findResource` to search for. (Any scheduler name starting with the string `'generic'` creates a generic scheduler object.)

```
sched = findResource('scheduler', 'type', 'generic')
```

Generic schedulers must use a shared file system for workers to access job and task data. Set the `DataLocation` 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(sched, 'DataLocation', '\\share\scratch\jobdata')
set(sched, 'HasSharedFilesystem', true)
```

---

**Note** All nodes require access to the directory specified in the scheduler object's `DataLocation` directory. See the `DataLocation` reference page for information on setting this property for a mixed-platform environment.

---

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, 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(sched, 'ClusterMatlabRoot', '\\apps\matlab\')
```

You can look at all the property settings on the scheduler object. If no jobs are in the `DataLocation` directory, 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(sched)
          Configuration: ''
                   Type: 'generic'
           DataLocation: '\\share\scratch\jobdata'
    HasSharedFilesystem: 1
                   Jobs: [0x1 double]
      ClusterMatlabRoot: '\\apps\matlab\'
          ClusterOsType: 'pc'
               UserData: []
            ClusterSize: Inf
     MatlabCommandToRun: 'worker'
              SubmitFcn: []
       ParallelSubmitFcn: []
```

You must set the `SubmitFcn` property to specify the submit function for this scheduler.

```
set(sched, 'SubmitFcn', @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 a job manager or any other type of 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 directory specified by the scheduler object's `DataLocation` property.

```
j = createJob(sched)
```

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

```
get(j)
       Configuration: ''
                Name: 'Job1'
                  ID: 1
            UserName: 'neo'
                 Tag: ''
               State: 'pending'
          CreateTime: 'Fri Jan 20 16:15:47 EDT 2006'
          SubmitTime: ''
           StartTime: ''
          FinishTime: ''
               Tasks: [0x1 double]
    FileDependencies: {0x1 cell}
    PathDependencies: {0x1 cell}
             JobData: []
              Parent: [1x1 distcomp.genericscheduler]
            UserData: []
```

---

**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 a job manager. 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 scheduler's `Jobs` property is now a 1-by-1 array of `distcomp.simplejob` objects, indicating the existence of your job.

```
get(sched)
          Configuration: ''
                   Type: 'generic'
           DataLocation: '\\share\scratch\jobdata'
    HasSharedFilesystem: 1
                   Jobs: [1x1 distcomp.simplejob]
      ClusterMatlabRoot: '\\apps\matlab\'
          ClusterOsType: 'pc'
               UserData: []
            ClusterSize: Inf
     MatlabCommandToRun: 'worker'
              SubmitFcn: @mysubmitfunc
       ParallelSubmitFcn: []
```

## 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')
ans =
    distcomp.simpletask: 5-by-1
```

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 `waitForState` function.

```
waitForState(j)
```

The default state to wait for is `finished` or `failed`. 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 `getAllOutputArguments` to retrieve the results from all the tasks in the job.

```
results = getAllOutputArguments(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 directory

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

In this directory are subdirectories for each of several types of scheduler, containing wrappers, submit functions, and decode functions for distributed and parallel jobs. For example, the directory *matlabroot*`/toolbox/distcomp/examples/integration/pbs` contains the following files for use with a PBS scheduler:

| Filename | Description |
|---|---|
| `pbsSubmitFcn.m` | Submit function for a distributed job |
| `pbsDecodeFunc.m` | Decode function for a distributed job |
| `pbsParallelSubmitFcn.m` | Submit function for a parallel job |
| `pbsParallelDecode.m` | Decode function for a parallel job |
| `pbsWrapper.sh` | Script that is submitted to PBS to start workers that evaluate the tasks of a distributed job |
| `pbsParallelWrapper.sh` | Script that is submitted to PBS to start labs that evaluate the tasks of a parallel job |

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 directories for PBS schedulers (`pbs`), Platform LSF schedulers (`lsf`), generic UNIX-based scripts (`ssh`), Sun Grid Engine (`sge`), and mpiexec on Microsoft Windows operating systems (`winmpiexec`). In addition, the `pbs` and `lsf` directories have subdirectories called `nonshared`, which contain scripts for use when there is a nonshared file system between the client and cluster computers. Each of these subdirectories contains a file called `README`, which provides instruction on how to use its scripts.

As more files or solutions 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 page also provides contact information in case you have any questions.

## Managing Jobs

While you can use the `get`, `cancel`, and `destroy` 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 destroy 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.

### Saving 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)
```

## Defining 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`
- `myDestroyJob.m`
- `myCancelTask.m`
- `myDestroyTask.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 destroying a job, and what to do for canceling and destroying tasks.

## Destroying or Canceling a Running Job

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

- `CancelJobFcn`
- `DestroyJobFcn`
- `CancelTaskFcn`
- `DestroyTaskFcn`

You can set the properties in the Configurations Manager for your scheduler, or on the command line:

```
schdlr = findResource(scheduler, 'type', 'generic');
% set required properties
set(schdlr, 'CancelJobFcn', @myCancelJob)
set(schdlr, 'DestroyJobFcn', @myDestroyJob)
set(schdlr, 'CancelTaskFcn', @myCancelTask)
set(schdlr, 'DestroyTaskFcn', @myDestroyTask)
```

Continue with job creation and submission as usual.

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

```
end
submit(j1)
```

While it is running or queued, you can cancel or destroy 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 `myDestroyJob`, which sends the appropriate commands to the scheduler:

```
destroy(j1)
```

### Getting 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`, `waitForState`, 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.

## 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 `SubmitFcn` 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.

9

# Programming Parallel Jobs

Parallel 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 parallel jobs:


- "Introduction" on page 9-2
- "Using a Supported Scheduler" on page 9-4
- "Using the Generic Scheduler Interface" on page 9-8
- "Further Notes on Parallel Jobs" on page 9-11

# Introduction

A parallel 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 referred to as *labs*.

In principle, creating and running parallel jobs is similar to programming distributed jobs:

1. Find a scheduler.
2. Create a parallel job.
3. Create a task.
4. 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 8-12.
5. Retrieve the results.

The differences between distributed jobs and parallel jobs are summarized in the following table.

| Distributed Job | Parallel Job |
|---|---|
| MATLAB sessions, called *workers*, perform the tasks but do not communicate with each other. | MATLAB sessions, 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 labs running the parallel job. |
| Tasks need not run simultaneously. Tasks are distributed to workers as the workers become available, so a worker can perform several of the tasks in a job. | Tasks run simultaneously, so you can run the job only on as many labs as are available at run time. The start of the job might be delayed until the required number of labs is available. |

A parallel job has only one task that runs simultaneously on every lab. The function that the task runs can take advantage of a lab's awareness of how many labs are running the job, which lab this is among those running the job, and the features that allow labs to communicate with each other.

# Using a Supported Scheduler

**In this section...**

"Schedulers and Conditions" on page 9-4

"Coding the Task Function" on page 9-4

"Coding in the Client" on page 9-5

## Schedulers and Conditions

You can run a parallel job using any type of scheduler. This section illustrates how to program parallel jobs for supported schedulers (job manager, local scheduler, Microsoft Windows HPC Server (including CCS), Platform LSF, PBS Pro, TORQUE, or mpiexec).

To use this supported interface for parallel 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** If all these conditions are not met, you must use the generic scheduler interface with any third-party scheduler running a parallel job, including `pmode`, `matlabpool`, `spmd`, and `parfor`. See "Using the Generic Scheduler Interface" on page 9-8.

---

## Coding the Task Function

In this section a simple example illustrates the basic principles of programming a parallel job with a third-party scheduler. In this example, the lab whose `labindex` value is `1` creates a magic square comprised of a number of rows and columns that is equal to the number of labs running the job (`numlabs`). In this case, four labs run a parallel job with a 4-by-4 magic square. The first lab broadcasts the matrix with `labBroadcast` to all the other labs , 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 labs
    A = labBroadcast(1,magic(numlabs))
else
    % Receive broadcast on other labs
    A = labBroadcast(1)
end

% Calculate sum of column identified by labindex for this lab
column_sum = sum(A(:,labindex))

% Calculate total sum by combining column sum from all labs
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 lab by the job's `FileDependencies` property.

While this example has one lab create the magic square and broadcast it to the other labs, there are alternative methods of getting data to the labs. Each lab could create the matrix for itself. Alternatively, each lab 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 `FileDependencies` property. The solution to choose depends on your network configuration and the nature of the data.

## Coding in the Client

As with distributed jobs, you find a scheduler and create a scheduler object in your MATLAB client by using the `findResource` function. There are slight differences in the arguments for `findResource`, depending on the scheduler you use, but using configurations to define as many properties as possible minimizes coding differences between the scheduler types.

You can create and configure the scheduler object with this code:

```
sched = findResource('scheduler', 'configuration', myconfig)
```

where `myconfig` is the name of a user-defined configuration for the type of scheduler you are using. Any required differences for various scheduling options are controlled in the configuration. You can have one or more separate configurations for each type of scheduler. For complete details, see "Programming with User Configurations" on page 6-16. Create or modify configurations according to the instructions of your system administrator.

When your scheduler object is defined, you create the job object with the `createParallelJob` function.

```
pjob = createParallelJob(sched);
```

The function file `colsum.m` (created in "Coding the Task Function" on page 9-4) is on the MATLAB client path, but it has to be made available to the labs. One way to do this is with the job's `FileDependencies` property, which can be set in the configuration you used, or by:

```
set(pjob, 'FileDependencies', {'colsum.m'})
```

Here you might also set other properties on the job, for example, setting the number of workers to use. Again, configurations 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 labs, you can established this in the configuration, or by the following client code:

```
set(pjob, 'MaximumNumberOfWorkers', 4)
set(pjob, 'MinimumNumberOfWorkers', 4)
```

You create the job's one task with the usual `createTask` function. In this example, the task returns only one argument from each lab, and there are no input arguments to the `colsum` function.

```
t = createTask(pjob, @colsum, 1, {})
```

Use `submit` to run the job.

```
submit(pjob)
```

Make the MATLAB client wait for the job to finish before collecting the results. The results consist of one value from each lab. The `gplus` function in the task shares data between the labs, so that each lab has the same result.

```
waitForState(pjob)
results = getAllOutputArguments(pjob)
results =
    [136]
    [136]
    [136]
    [136]
```

# Using the Generic Scheduler Interface

| In this section... |
| --- |
| "Introduction" on page 9-8 |
| "Coding in the Client" on page 9-8 |

## Introduction

This section discusses programming parallel 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 parallel 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 "Using the Generic Scheduler Interface" on page 8-33 in the chapter on Programming Distributed Jobs.

## Coding in the Client

### Configuring the Scheduler Object

Coding a parallel job for a generic scheduler involves the same procedure as coding a distributed job.

1. Create an object representing your scheduler with `findResource`.
2. Set the appropriate properties on the scheduler object if they are not defined in the configuration. Because the scheduler itself is often common to many users and applications, it is probably best to use a configuration for programming these properties. See "Programming with User Configurations" on page 6-16.

   Among the properties required for a parallel job is `ParallelSubmitFcn`. The toolbox comes with several submit functions for various schedulers and platforms; see the following section, "Supplied Submit and Decode Functions" on page 9-9.

3 Use `createParallelJob` to create a parallel job object for your scheduler.

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 directory

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

In this directory are subdirectories for each of several types of scheduler, containing wrappers, submit functions, and decode functions for distributed and parallel jobs. For example, the directory *matlabroot*`/toolbox/distcomp/examples/integration/pbs` contains the following files for use with a PBS scheduler:

| Filename | Description |
|---|---|
| `pbsSubmitFcn.m` | Submit function for a distributed job |
| `pbsDecodeFunc.m` | Decode function for a distributed job |
| `pbsParallelSubmitFcn.m` | Submit function for a parallel job |
| `pbsParallelDecode.m` | Decode function for a parallel job |
| `pbsWrapper.sh` | Script that is submitted to PBS to start workers that evaluate the tasks of a distributed job |
| `pbsParallelWrapper.sh` | Script that is submitted to PBS to start labs that evaluate the tasks of a parallel job |

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 directories for PBS schedulers (`pbs`), Platform LSF schedulers (`lsf`), generic UNIX-based scripts (`ssh`), Sun Grid Engine (`sge`), and mpiexec on Microsoft Windows operating systems (`winmpiexec`). In addition, the `pbs` and `lsf` directories have subdirectories called `nonshared`, which contain scripts for use when there is a nonshared file

system between the client and cluster computers. Each of these subdirectories contains a file called `README`, which provides instruction on how to use its scripts.

As more files or solutions 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 page also provides contact information in case you have any questions.

# Further Notes on Parallel Jobs

**In this section...**

"Number of Tasks in a Parallel Job" on page 9-11

"Avoiding Deadlock and Other Dependency Errors" on page 9-11

## Number of Tasks in a Parallel Job

Although you create only one task for a parallel job, the system copies this task for each worker that runs the job. For example, if a parallel 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 lab whose `labindex` is `1`, and so on, so that the `ID` property for the task object and `labindex` for the lab that ran that task have the same value. Therefore, the sequence of results returned by the `getAllOutputArguments` function corresponds to the value of `labindex` and to the order of tasks in the job's `Tasks` property.

## Avoiding Deadlock and Other Dependency Errors

Because code running in one lab for a parallel job can block execution until some corresponding code executes on another lab, the potential for deadlock exists in parallel jobs. This is most likely to occur when transferring data between labs 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 lab.

```
if labindex == 1
    assembled = gather(D);
end
```

The reason this fails is because the `gather` function requires communication between all the labs across which the array is distributed. When the `if` statement limits execution to a single lab, the other labs 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 lab: `assembled = gather(D, 1)`.

In another example, suppose you want to transfer data from every lab to the next lab on the right (defined as the next higher `labindex`). First you define for each lab what the labs 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 lab until the corresponding receiving lab executes its `labReceive` function. In this case, all the labs 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 labs 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.

10

# Object Reference


| | |
|---|---|
| Data Objects (p. 10-2) | Representing data on multiple labs |
| Scheduler Objects (p. 10-2) | Representing job manager, local scheduler, or third-party scheduler |
| Job Objects (p. 10-3) | Representing different types of jobs |
| Task Objects (p. 10-3) | Representing different types of tasks |
| Worker Objects (p. 10-3) | Representing MATLAB worker sessions |

## Data Objects

| | |
|---|---|
| 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 |

## Scheduler Objects

| | |
|---|---|
| ccsscheduler | Access Microsoft Windows HPC Server scheduler |
| genericscheduler | Access generic scheduler |
| jobmanager | Control job queue and execution |
| localscheduler | Access local scheduler on client machine |
| lsfscheduler | Access Platform LSF scheduler |
| mpiexec | Directly access mpiexec for job distribution |
| pbsproscheduler | Access PBS Pro scheduler |
| torquescheduler | Access TORQUE scheduler |

# Job Objects

| | |
|---|---|
| job | Define job behavior and properties when using job manager |
| matlabpooljob | Define MATLAB pool job behavior and properties when using job manager |
| paralleljob | Define parallel job behavior and properties when using job manager |
| simplejob | Define job behavior and properties when using local or third-party scheduler |
| simplematlabpooljob | Define MATLAB pool job behavior and properties when using local or third-party scheduler |
| simpleparalleljob | Define parallel job behavior and properties when using local or third-party scheduler |

# Task Objects

| | |
|---|---|
| simpletask | Define task behavior and properties when using local or third-party scheduler |
| task | Define task behavior and properties when using job manager |

# Worker Objects

| | |
|---|---|
| worker | Access information about MATLAB worker session |

11

# 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 job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `findJob` | Find job objects stored in scheduler |
| `getDebugLog` | Read output messages from job run by supported third-party or local 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 |
| `ClusterVersion` | Version of HPC Server scheduler |

| | |
|---|---|
| `Configuration` | Specify configuration to apply to object or toolbox function |
| `DataLocation` | Specify directory 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 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 |

# 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 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 job object in scheduler and client |
| `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 directory 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

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 finish or change state |
| `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 |
| `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 |

## 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

| | |
|---|---|
| `createJob` | Create job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `demote` | Demote job in job manager queue |
| `findJob` | Find job objects stored in scheduler |
| `pause` | Pause job manager queue |
| `promote` | Promote job in job manager 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 |
| `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 |
| `State` | Current state of task, job, job manager, or worker |
| `Type` | Type of scheduler object |
| `UserData` | Specify data to associate with 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 job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `findJob` | Find job objects stored in scheduler |
| `getDebugLog` | Read output messages from job run by supported third-party or local 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 directory 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 job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `findJob` | Find job objects stored in scheduler |
| `getDebugLog` | Read output messages from job run by supported third-party or local scheduler |
| `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 directory 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 |
| `ParallelSubmission-WrapperScript` | 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 finish or change state |
| `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 |
| `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 |

## See Also

`paralleljob`, `simplematlabpooljob`, `simpleparalleljob`

# mpiexec

## 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 job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `findJob` | Find job objects stored in scheduler |
| `getDebugLog` | Read output messages from job run by supported third-party or local 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 directory 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`

# paralleljob

## 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 finish or change state |
| `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 |
| `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 |

## See Also

`job`, `simplejob`, `simpleparalleljob`

## 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 job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `findJob` | Find job objects stored in scheduler |
| `getDebugLog` | Read output messages from job run by supported third-party or local scheduler |
| `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 directory 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 |
| `ParallelSubmission-WrapperScript` | 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

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 finish or change state |
| `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 |

## 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 finish or change state |
| `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 |

## 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 finish or change state |
| `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 |

## 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 job object in scheduler and client |
| `createMatlabPoolJob` | Create MATLAB pool job |
| `createParallelJob` | Create parallel job object |
| `findJob` | Find job objects stored in scheduler |
| `getDebugLog` | Read output messages from job run by supported third-party or local scheduler |
| `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 directory 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 |
| `ParallelSubmission-WrapperScript` | 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 (p. 12-2) | Constructs for automatically running code in parallel |
| Distributed and Codistributed Arrays (p. 12-3) | Data partitioned across multiple MATLAB sessions |
| Job and Task Programming (p. 12-6) | Parallel computation through individual tasks |
| Interlab Communication Within a Parallel Job (p. 12-9) | Communications between labs during job execution |
| Utilities (p. 12-10) | Utilities for using Parallel Computing Toolbox |

# Parallel Code Execution


| | |
|---|---|
| Parallel Code on a MATLAB Pool (p. 12-2) | Parallel computations on a pool of MATLAB sessions |
| Configuration, Input, and Output (p. 12-2) | Data access and setup control |
| Interactive Functions (p. 12-3) | Parallel code development and debugging |


## Parallel Code on a MATLAB Pool

| | |
|---|---|
| `batch` | Run MATLAB script as batch job |
| `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 |

## Configuration, Input, and Output

| | |
|---|---|
| `defaultParallelConfig` | Default parallel computing configuration |
| `diary` | Display or save Command Window text of batch job |
| `exist` | Check whether Composite is defined on labs |
| `importParallelConfig` | Import parallel configuration .mat file |

| | |
|---|---|
| `load` | Load workspace variables from batch job |
| `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 (p. 12-3) | Array creation and manipulation |
| Overloaded MATLAB Functions (p. 12-4) | Predefined arrays with overloaded functions |

### 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 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 |

# Job and Task Programming


| | |
|---|---|
| Job Creation (p. 12-6) | Job and task definition |
| Job Management (p. 12-7) | Job and task execution |
| Task Execution Information (p. 12-8) | Information on the processes evaluating a task |
| Object Control (p. 12-9) | Parallel Computing Toolbox objects |


## Job Creation

| | |
|---|---|
| `createJob` | Create job object in scheduler and client |
| `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` | File for user-defined options to run when task finishes |
| `taskStartup` | File for user-defined options to run when task starts |

## Job Management

| | |
|---|---|
| `cancel` | Cancel job or task |
| `demote` | Demote job in job manager 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 by supported third-party or local scheduler |
| getJobSchedulerData | Get specific user data for job on generic scheduler |
| pause | Pause job manager queue |
| promote | Promote job in job manager queue |
| resume | Resume processing queue in job manager |
| setJobSchedulerData | Set specific user data for job on generic scheduler |
| submit | Queue job in scheduler |
| wait | Wait for job to finish or change state |
| waitForState | Wait for object to change state |

## Task Execution Information

| | |
|---|---|
| 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 Parallel 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 |

## Utilities

| | |
|---|---|
| `help` | Help for toolbox functions in Command Window |
| `pctRunDeployedCleanup` | Clean up after deployed parallel applications |

13

# Functions — Alphabetical List

# batch

## Purpose

Run MATLAB script as batch job

## Syntax

```
j = batch('aScript')
j = batch(..., 'p1', v1, 'p2', v2, ...)
```

## Arguments

| | |
|---|---|
| `j` | The MATLAB pool job object. |
| `'aScript'` | The script of MATLAB code to be evaluated by the MATLAB pool job. |
| *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 according to the scheduler defined in the default parallel configuration. The function returns `j`, a handle to the job object that runs the script. The script file `aScript.m` is added to the `FileDependencies` and copied to the worker.

`j = batch(..., 'p1', v1, 'p2', v2, ...)` allows additional parameter-value pairs that modify the behavior of the job. 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.
- `'Configuration'` — A single string that is the name of a parallel configuration to use to find the correct cluster. By default it is the string returned from `defaultParallelConfig`. If you want the configuration's settings applied to the job properties, you must explicitly specify the configuration, even if using the default. To apply properties from the default parallel configuration, specify it with:

```
batch(...,'Configuration', defaultParallelConfig)
```

- `'PathDependencies'` — A string or cell array of strings that defines paths to be added to the workers' MATLAB path before the script is executed.
- `'FileDependencies'` — A string or cell array of strings. Each string in the list identifies either a file or a directory, which is transferred to the worker.
- `'CurrentDirectory'` — 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, unless any `FileDependencies` are defined. 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 positive scalar integer that defines the number of labs to make into a MATLAB pool for the job to run on in addition to the worker running the batch script. The script uses the pool for execution of statements such as `parfor` and `spmd`. A value of `N` for the property `Matlabpool` is effectively the same as adding a call to `matlabpool N` into the script. Because the MATLAB pool requires `N` workers in addition to the worker running the batch script, there must be at least `N+1` workers available on the cluster. The default value is 0, which causes the script to run on only the single worker without a MATLAB pool.

## Remarks

As a matter of good programming practice, when you no longer need it, you should destroy 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', 'matlabpool', 0);
```

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. This job requires a total of nine workers:

```
j = batch('script1', 'matlabpool', 8);
```

Run a batch MATLAB pool job on a local worker, which employs two other local workers:

```
j = batch('script1', 'configuration', 'local', ...
    'matlabpool', 2);
```

Clean up a batch job's data after you are finished with it:

```
destroy(j)
```

## See Also

`diary`, `load`, `wait`

## 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

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.

## Remarks

If `obj` references an object in the job manager, it is cleared from the workspace, but it remains in the job manager. You can restore `obj` to the workspace with the `findResource`, `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.

```
job1 = createJob(jm);
job2 = createJob(jm);
job1copy = job1;
clear job1 job2;
j1 = jm.Jobs(1);
j2 = jm.Jobs(2);
isequal (job1copy, j1)
ans =
     1
isequal (job1copy, j2)
ans =
     0
```

## See Also

`createJob`, `createTask`, `findJob`, `findResource`, `findTask`

# codistributed

## 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)`.

## Remarks

`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

# codistributed.build

## 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

# codistributed.cell

## 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` MATLAB function reference page

`distributed.cell`

# codistributed.colon

## 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` MATLAB function reference page

`codistributor1d`, `codistributor2dbc`, `for`

# codistributed.eye

## Purpose

Create codistributed identity matrix

## Syntax

```
C = codistirubed.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 = codistirubed.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` MATLAB function reference page

`codistributed.ones`, `codistributed.speye`, `codistributed.zeros`, `distributed.eye`

# codistributed.false

## 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` MATLAB function reference page

`codistributed.true`, `distributed.false`

# codistributed.inf

## 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` MATLAB function reference page

`codistributed.nan`, `distributed.inf`

# codistributed.nan

## 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` MATLAB function reference page

`codistributed.inf`, `distributed.nan`

# codistributed.ones

## 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` MATLAB function reference page

`codistributed.eye`, `codistributed.zeros`, `distributed.ones`

# codistributed.rand

## 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.

## Remarks

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` MATLAB function reference page

`codistributed.randn`, `codistributed.sprand`, `codistributed.sprandn`, `distributed.rand`

# codistributed.randn

## 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.

## Remarks

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` MATLAB function reference page

`codistributed.rand`, `codistributed.sprand`, `codistributed.sprandn`, `distributed.randn`

# codistributed.spalloc

## 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` MATLAB function reference page

`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` MATLAB function reference page

`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
```

## Remarks

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` MATLAB function reference page

`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
```

## Remarks

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` MATLAB function reference page

`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` MATLAB function reference page

`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` MATLAB function reference page

`codistributed.eye`, `codistributed.ones`, `distributed.zeros`

# codistributor

## 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 = nrol/2 = sqrt(numlabs/2)`.
- `nrow <= ncol`.
- If `numlabs` is a prime, `nrow = 1`, `ncol = nmlabs`.
- `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`

# Composite

## 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 job object in scheduler and client

## Syntax

```
obj = createJob()
obj = createJob(scheduler)
obj = createJob(..., 'p1', v1, 'p2', v2, ...)
obj = createJob(..., 'configuration', 'ConfigurationName',
   ...)
```

## Arguments

| | |
|---|---|
| `obj` | The 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

`obj = createJob()` creates a job using the scheduler identified by the default parallel configuration and sets the property values of the job as specified in the default configuration.

`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 `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.

`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. For details about defining and applying configurations, see "Programming with User Configurations" on page 6-16.

## Examples

Construct a job object using the default configuration.

```
obj = createJob();
```

Add tasks to the job.

```
for i = 1:10
    createTask(obj, @rand, 1, {10});
end
```

Run the job.

```
submit(obj);
```

Wait for the job to finish running, and retrieve the job results.

```
waitForState(obj);
out = getAllOutputArguments(obj);
```

Display the random matrix returned from the third task.

```
disp(out{3});
```

Destroy the job.

```
destroy(obj);
```

## See Also

`createParallelJob`, `createTask`, `findJob`, `findResource`, `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 will 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 "Programming with User Configurations" on page 6-16.

## 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 "Programming with User Configurations" on page 6-16.

## 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(...,'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 parallel job has only one task, this form of vectorized task creation is not appropriate for parallel 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(...,'**configuration**', 'ConfigurationName',...) creates a task job object with the property values specified in the configuration ConfigurationName. For details about defining and applying configurations, see "Programming with User Configurations" on page 6-16.

## Examples

Create a job object.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
```

Add a task object which generates a 10-by-10 random matrix.

```
obj = 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.

```
waitForState(j);
taskoutput = get(obj, '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.

```
jm = findResource('scheduler','type','jobmanager', ...
           'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
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

Demote job in job manager queue

## Syntax

```
demote(jm, job)
```

## Arguments

| | |
|---|---|
| `jm` | The job manager object that contains the job. |
| `job` | Job object demoted in the job queue. |

## Description

`demote(jm, job)` demotes the job object `job` that is queued in the job manager `jm`.

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.

## Remarks

After a call to `demote` or `promote`, there is no change in the order of job objects contained in the `Jobs` property of the job manager 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(jm)`.

## Examples

Create and submit multiple jobs to the job manager identified by the default parallel configuration:

```
jm = findResource();
j1 = createJob('name','Job A');
j2 = createJob('name','Job B');
j3 = createJob('name','Job C');
submit(j1);submit(j2);submit(j3);
```

Demote one of the jobs by one position in the queue:

```
demote(jm, j2)
```

Examine the new queue sequence:

```
[pjobs, qjobs, rjobs, fjobs] = findJob(jm);
get(qjobs, 'Name')
```

```
'Job A'
'Job C'
'Job B'
```

## See Also

createJob, findJob, promote, submit

## Purpose

Remove job or task object from parent and memory

## Syntax

```
destroy(obj)
```

## Arguments

| | |
|---|---|
| `obj` | Job or task object deleted from 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.

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.

## Remarks

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.

```
jm = findResource('scheduler','type','jobmanager', ...
                  'name','MyJobManager','LookupURL','JobMgrHost');
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`.

## See Also

`createJob`, `createTask`

# 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 “Programming with User Configurations” on page 6-16. 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 "Evaluating 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`

## 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 "Programming with

User Configurations" on page 6-16. Configurations enable you to use `dfevalasync` with any type of scheduler.

For further discussion on the usage of `dfevalasync`, see "Evaluating 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` MATLAB function reference page

`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.

## 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` MATLAB function reference page

`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` MATLAB function reference page

`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` MATLAB function reference page

`codistributed.false`, `distributed.true`

## 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` MATLAB function reference page

`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` MATLAB function reference page

`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` MATLAB function reference page

`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'`.

## Remarks

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` MATLAB function reference page

`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'`.

## Remarks

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` MATLAB function reference page

`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` MATLAB function reference page

`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` MATLAB function reference page

`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.

## Remarks

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` MATLAB function reference page

`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.

## Remarks

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` MATLAB function reference page

`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` MATLAB function reference page

`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` MATLAB function reference page

`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` MATLAB function reference page

`Composite`, `distributed`, `dsave`, `matlabpool`

# dsave

## 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` MATLAB function reference page

`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`

# findJob

## Purpose

Find job objects stored in scheduler

## Syntax

```
out = findJob(sched)
[pending queued running finished] = 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`. |
| `finished` | Array of jobs whose `State` is `finished` 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 finished] = findJob(sched)` returns arrays of all job objects stored in the scheduler `sched`, by state. Within `pending`, `running`, and `finished`, 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.

`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`, `submit`

# findResource

## Purpose

Find available parallel computing resources

## Syntax

```
out = findResource()
out = findResource('scheduler', ... 'configuration',
   'ConfigurationName')
out = findResource('scheduler', 'type', 'SchedType')
out = findResource('worker')
out = findResource('scheduler', 'type', 'jobmanager',
   'LookupURL', 'host:port')
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. |
| *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 "Programming with User Configurations" on page 6-16.

---

**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`*`')` and `out = findResource('worker')` return an array, `out`, containing objects representing all available parallel computing schedulers of the given type, or workers. *`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')` and `out = findResource('worker', 'LookupURL', 'host:port')` use 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 4 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(... ,'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'`.

## Remarks

Note that it is permissible 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 and host.

```
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')
```

## See Also

`findJob`, `findTask`

# findTask

## Purpose

Task objects belonging to job object

## Syntax

```
tasks = findTask(obj)
[pending running finished] = 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`. |
| `finished` | Array of tasks in job `obj` whose `State` is `finished`. |
| *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 finished] = findTask(obj)` returns arrays of all task objects stored in the job object `obj`, sorted by state. Within each state (`pending`, `running`, and `finished`), 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`.

## Remarks

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.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
obj = createJob(jm);
```

Add a task to the job object.

```
createTask(obj, @rand, 1, {10})
```

Create the task object `t`, which refers to the task we just added to `obj`.

```
t = findTask(obj)
```

## See Also

`createJob`, `createTask`, `findJob`

# for

## 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` MATLAB function reference page

`numlabs`, `parfor`

# gather

## Purpose

Transfer distributed array data 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.

## Remarks

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, so that it can be saved from there.

```
n = 10;
spmd
    C = codistributed(magic(n));
    out = gather(C, 1);
    if labindex == 1
        save data.mat out;
    end
end
```

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
```

## See Also

`codistributed`, `pmode`

# gcat

## Purpose

Global concatenation

## Syntax

```
Xs = gcat(X)
Xs = gcat(X, dim)
```

## Description

`Xs = gcat(X)` concatenates the variant arrays X from each lab in the second dimension. The result is replicated on all labs.

`Xs = gcat(X, dim)` concatenates the variant arrays X from each lab in the dim-th dimension.

## Examples

With four labs,

```
Xs = gcat(labindex)
```

returns `Xs = [1 2 3 4]` on all four labs.

## See Also

`cat` MATLAB function reference page

`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.

## Remarks

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.

## Remarks

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`

## 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

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.

## Remarks

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`

# getCurrentJobmanager

## 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.

## Remarks

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.

## Remarks

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`

# getCurrentWorker

## 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.

## Remarks

If the function is executed in a MATLAB session that is not a worker or if you are using a third-party scheduler instead of a job manager, you get an empty result.

## Examples

Create a job with one task, and have the task return the name of the worker that evaluates it.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t = createTask(j, @() get(getCurrentWorker,'Name'), 1, {});
submit(j)
waitForState(j)
get(t,'OutputArgument')
ans =
    'c5_worker_43'
```

The function of the task `t` is an anonymous function that first executes `getCurrentWorker` to get an object representing the worker that is evaluating the task. Then the task function uses `get` to examine the `Name` property value of that object. The result is placed in the `OutputArgument` property of the task.

## See Also

`getCurrentJob`, `getCurrentJobmanager`, `getCurrentTask`, `getFileDependencyDir`

## Purpose

Read output messages from job run by supported third-party or local scheduler

## Syntax

```
str = getDebugLog(sched, job_or_task)
```

## Arguments

| | |
|---|---|
| `str` | Variable to which messages are returned as a string expression. |
| `sched` | Scheduler 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(sched, 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 by the scheduler identified by `sched`. You cannot use this function to retrieve messages from a task if the scheduler is mpiexec.

## Examples

Construct a scheduler object so you can create a parallel job. Assume that you have already defined a configuration called `mpiexec` to define the properties of the scheduler object.

```
mpiexecObj = findResource('scheduler', 'Configuration', 'mpiexec');
```

Create and submit a parallel job.

```
job = createParallelJob(mpiexecObj);
createTask(job, @labindex, 1, {});
submit(job);
```

Look at the debug log.

```
getDebugLog(mpiexecObj, job);
```

**See Also** findResource, createJob, createParallelJob, createTask

## 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

### Functions

`getCurrentJob`, `getCurrentJobmanager`, `getCurrentTask`, `getCurrentWorker`

### Properties

`FileDependencies`

# getJobSchedulerData

## 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 "Managing Jobs" on page 8-48.

## 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`

# 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`

# gop

## 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` 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)
```

## Description

`s = gplus(x)` returns the addition of the `x` from each lab. The result is replicated on all labs.

## Examples

With four labs,

```
s = gplus(labindex)
```

returns `s = 1 + 2 + 3 + 4 = 10` on all four labs.

## See Also

`gop`, `labindex`

## 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

# inspect

## 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`.

## Remarks

You can also open the Property Inspector via the Workspace browser by double-clicking an object.

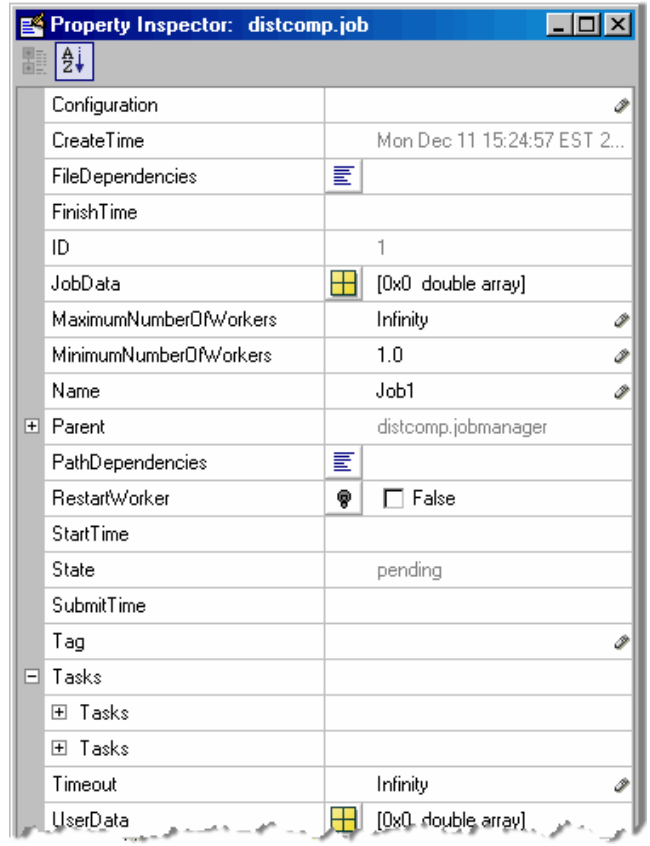
The Property Inspector does not automatically update its display. To refresh the Property Inspector, open it again.

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)
```




## See Also

`get`, `set`

# isaUnderlying

## 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` MATLAB function reference page

## 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 "Array Types" 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 “Array Types” 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 "Array Types" 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`

# isreplicated

## 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 “Array Types” on page 5-2. `isreplicated` also returns true for a Composite `X` if all its elements are identical.

## Remarks

`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 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.

The file resides in the worker's MATLAB installation at

*`matlabroot`*`/toolbox/distcomp/user/jobStartup.m`

You add MATLAB code to the file to define job initialization actions to be performed on the worker when it first evaluates a task for this job.

Alternatively, you can create a file called `jobStartup.m` and include it as part of the job's `FileDependencies` property. The version of the file in `FileDependencies` takes precedence over the version in the worker's MATLAB installation.

For further detail, see the text in the installed `jobStartup.m` file.

## See Also

### Functions

`poolStartup`, `taskFinish`, `taskStartup`

### Properties

`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`

## 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

## 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`

# labProbe

## 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`

## 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.

## Remarks

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`

## 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`.

## Remarks

This function might 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`

# labSendReceive

## 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`

# load

## 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. If the job is not finished, or if the job encountered an error while running, `load` will throw an error.

`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`

# matlabpool

## Purpose

Open or close pool of MATLAB sessions for parallel computation

## Syntax

```
matlabpool
matlabpool open
matlabpool open poolsize
matlabpool open configname
matlabpool open configname poolsize
matlabpool poolsize
matlabpool configname
matlabpool configname poolsize
matlabpool close
matlabpool close force
matlabpool close force configname
matlabpool size
matlabpool('open', ...)
matlabpool('close', ...)
matlabpool( open ,...,  FileDependencies , filecell)
matlabpool( addfiledependencies , filecell)
matlabpool updatefiledependencies
```

## 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 labs.

`matlabpool` or `matlabpool open` starts a worker pool using the default parallel configuration, with the pool size specified by that configuration. (For information about setting up and selecting parallel configurations, see "Programming with User Configurations" on page 6-16.) You can also specify the pool size using `matlabpool open poolsize`, but most schedulers have a maximum number of processes that they can start (8 for a local scheduler). If the configuration specifies a job manager as the scheduler, `matlabpool` reserves its workers from among those already running and available under that job manager. If the configuration specifies a third-party scheduler, `matlabpool` instructs the scheduler to start the workers.

`matlabpool open configname` or `matlabpool open configname poolsize` starts a worker pool using the Parallel Computing Toolbox

user configuration identified by `configname` rather than the default configuration to locate a scheduler. If the pool size is specified, it overrides the maximum and minimum number of workers specified in the configuration, 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 configname`, and `matlabpool configname poolsize` operate as `matlabpool open ...`, and are provided for convenience.

`matlabpool` **`close`** stops the worker pool, destroys the parallel job, and makes all parallel language features revert to using the MATLAB client for computing their results.

`matlabpool` **`close force`** destroys all parallel jobs created by `matlabpool` for the current user under the scheduler specified by the default configuration, including any jobs currently running.

`matlabpool` **`close force`** `configname` destroys all parallel jobs being run under the scheduler specified in the configuration `configname`.

`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. The default is `'`**`open`**`'`. For example, the following are equivalent:

```
matlabpool open MyConfig 4
matlabpool('open', 'MyConfig', 4)
```

`matlabpool(` **`open`** `,...,` **`FileDependencies`** `, filecell)` starts a worker pool and allows you to specify file dependencies so that you can pass necessary files to the workers in the pool. The cell array `filecell` is appended to the `FileDependencies` specified in the configuration used for startup. The `'FileDependencies'` 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( addfiledependencies , filecell)` allows you to add extra file dependencies to an already running pool. `filecell` is a cell array of strings, identical in form to those you use when adding file dependencies to a job or when you open a MATLAB pool. Each string can specify either absolute or relative files, directories, or a file on the MATLAB path. The command transfers the files to each worker, placing the files in the file dependencies directory, exactly the same as if you set them at the time the pool was opened.

`matlabpool updatefiledependencies` checks all the file dependencies of the current pool to see if they have changed, and replicates any changes to each of the labs in the pool. In this way, you can send code changes out to remote labs. This checks dependencies that you added with the `matlabpool addfiledependencies` command as well as those you specified when the pool was started (by a configuration or command-line argument).

## Remarks

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 directory 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 is an issue especially in a mixed-platform environment where the client is not the same platform

as the workers, where directories local to or mapped from the client are not available in the same way to the workers, or where directories 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 configuration to define the number of labs:

```
matlabpool
```

Start a pool of 16 labs using a configuration called `myConf`:

```
matlabpool open myConf 16
```

Start a pool of 2 labs using the local configuration:

```
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 configuration, and pass two code files to the workers:

```
matlabpool('open', 'FileDependencies', {'mod1.m', 'mod2.m'})
```

## See Also

`defaultParallelConfig`, `pctRunOnAll`, `parfor`

## 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 job manager, job, and task objects, and examine what methods are available for each.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
methods(jm)
Methods for class distcomp.jobmanager:
createJob          demote        pause         resume
createParallelJob  findJob       promote

j1 = createJob(jm);
methods(j1)
Methods for class distcomp.job:
cancel      destroy   getAllOutputArguments  waitForState
createTask  findTask  submit

t1 = createTask(j1, @rand, 1, {3});
methods(t1)
Methods for class distcomp.task:
cancel   destroy  waitForState
```

**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.)

## Remarks

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`<br>`-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 records information about built-in functions such as `eig` or `labReceive`. |

| Option | Description |
|---|---|
| `-messagedetail default`<br>`-messagedetail simplified` | This option specifies the detail at which communication information is stored.<br>`-messagedetail default` collects information on a per-lab instance.<br>`-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.<br>For information about the structure of returned data, see `mpiprofile info` below. |
| `-history`<br>`-nohistory`<br>`-historysize <size>` | `mpiprofile` supports these options in the same way as the standard `profile`.<br>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 "Using the Parallel Profiler" on page 6-32.

## 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` MATLAB function reference page

`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.

## Remarks

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

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];`<br>`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 6-45.

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.

## Remarks

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`

# pload

## 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` MATLAB function reference pages

`labindex`, `numlabs`, `pmode`, `psave`

# pmode

## Purpose

Interactive Parallel Command Window

## Syntax

```
pmode start
pmode start numlabs
pmode start conf numlabs
pmode quit
pmode exit
pmode client2lab clientvar labs labvar
pmode lab2client labvar lab clientvar
pmode cleanup conf
```

## 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 labs running the job. The labs 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 labs.

`pmode start` starts pmode, using the default configuration to define the scheduler and number of labs. (The initial default configuration is `local`; you can change it by using the function `defaultParallelConfig`.) You can also specify the number of labs using `pmode start numlabs`, but note that the local scheduler allows for only up to eight labs.

`pmode start conf numlabs` starts pmode using the Parallel Computing Toolbox configuration `conf` to locate the scheduler, submits a parallel job with the number of labs identified by `numlabs`, and connects the Parallel Command Window with the labs. If the number of labs is specified, it overrides the minimum and maximum number of workers specified in the configuration.

`pmode quit` or `pmode exit` stops the parallel job, destroys it, and closes the Parallel Command Window. You can enter this command at the MATLAB prompt or the `pmode` prompt.

`pmode client2lab clientvar labs labvar` copies the variable `clientvar` from the MATLAB client to the variable `labvar` on the labs

identified by `labs`. If `labvar` is omitted, the copy is named `clientvar`. `labs` can be either a single lab index or a vector of lab indices. You can enter this command at the MATLAB prompt or the `pmode` prompt.

`pmode` **`lab2client`** `labvar lab clientvar` copies the variable `labvar` from the lab identified by `lab`, to the variable `clientvar` on the MATLAB client. If `clientvar` is omitted, the copy is named `labvar`. 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 lab is transferred. To transfer an entire codistributed array, first use the `gather` function to assemble the whole array into the labs' workspaces.

`pmode` **`cleanup`** `conf` destroys all parallel jobs created by `pmode` for the current user running under the scheduler specified in the configuration `conf`, including jobs that are currently running. The configuration is optional; the default configuration 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 conf 4
pmode('start', 'conf', 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 configuration to identify the scheduler and number of labs.

```
pmode start
```

Start `pmode` using the `local` configuration with four local labs.

```
pmode start local 4
```

Start `pmode` using the configuration `myconfig` and eight labs on the cluster.

```
pmode start myconfig 8
```

Execute a command on all labs.

```
P>> x = 2*labindex;
```

Copy the variable `x` from lab 7 to the MATLAB client.

```
pmode lab2client x 7
```

Copy the variable `y` from the MATLAB client to labs 1 to 8.

```
pmode client2lab y 1:8
```

Display the current working directory of each lab.

```
P>> pwd
```

## See Also

`createParallelJob`, `defaultParallelConfig`, `findResource`

## 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.

The file resides in the worker's MATLAB installation at

```
matlabroot/toolbox/distcomp/user/poolStartup.m
```

You add MATLAB code to the file to define initialization actions to be performed on the worker every time it forms part of a new MATLAB pool.

Alternatively, you can create a file called `poolStartup.m` and include it as part of the job's `FileDependencies` property, or include its containing folder in the job's `PathDependencies` property. The precedence for these options is:

1. `FileDependencies`
2. `PathDependencies`
3. Worker machine's installation of *`matlabroot`*`/toolbox/distcomp/user/poolStartup.m`

`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, see the text in the installed `poolStartup.m` file.

## See Also

### Functions

jobStartup, taskFinish, taskStartup

### Properties

FileDependencies, PathDependencies

## Purpose

Promote job in job manager queue

## Syntax

```
promote(jm, job)
```

## Arguments

| | |
|---|---|
| `jm` | The job manager object that contains the job. |
| `job` | Job object promoted in the queue. |

## Description

`promote(jm, job)` promotes the job object `job`, that is queued in the job manager `jm`.

If `job` is not the first job in the queue, `promote` exchanges the position of `job` and the previous job.

## Remarks

After a call to `promote` or `demote`, there is no change in the order of job objects contained in the `Jobs` property of the job manager 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(jm)`.

## Examples

Create and submit multiple jobs to the scheduler identified by the default parallel configuration:

```
j1 = createJob('name','Job A');
j2 = createJob('name','Job B');
j3 = createJob('name','Job C');
submit(j1);submit(j2);submit(j3);
```

Assuming that the default parallel configuration uses a job manager, create an object for that job manager, and promote `Job C` by one position in its queue:

```
jm = findResource();
promote(jm, j3)
```

Examine the new queue sequence:

```
[pjobs, qjobs, rjobs, fjobs] = findJob(jm);
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 labs' 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 directory accessible to all the labs. 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 lab.

## 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` MATLAB function reference pages

`labindex`, `numlabs`, `pmode`, `pload`

## 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`

# resume

## 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`

## 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 “Programming with User Configurations” on page 6-16.

## Remarks

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`

# setJobSchedulerData

## 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 "Managing Jobs" on page 8-48.

## See Also

`getJobSchedulerData`

## 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` MATLAB function reference page

`distributed.spalloc`, `codistributed.spalloc`

# spmd

## 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)".

## Remarks

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`

# submit

## Purpose

Queue job in scheduler

## Syntax

```
submit(obj)
```

## Arguments

| | |
|---|---|
| `obj` | Job object to be queued. |

## Description

`submit(obj)` queues the job object, `obj`, in the scheduler queue. The scheduler used for this job was determined when the job was created.

## Remarks

When a job contained in a scheduler is submitted, 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 `destroy`.

## Examples

Find the job manager named `jobmanager1` using the lookup service on host `JobMgrHost`.

```
jm1 = findResource('scheduler','type','jobmanager', ...
          'name','jobmanager1','LookupURL','JobMgrHost');
```

Create a job object.

```
j1 = createJob(jm1);
```

Add a task object to be evaluated for the job.

```
t1 = createTask(j1, @myfunction, 1, {10, 10});
```

Queue the job object in the job manager.

```
submit(j1);
```

## See Also

`createJob`, `findJob`

## 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 labs 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` MATLAB function reference page

`Composite`, `subsref`

# 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 labs 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` MATLAB function reference page

`Composite`, `subsasgn`

## Purpose

File for user-defined options to run 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.

The file resides in the worker's MATLAB installation at

```
matlabroot/toolbox/distcomp/user/taskFinish.m
```

You add MATLAB code to the file to define task finalization actions to be performed on the worker every time it finishes evaluating a task for this job.

Alternatively, you can create a file called `taskFinish.m` and include it as part of the job's `FileDependencies` property. The version of the file in `FileDependencies` takes precedence over the version in the worker's MATLAB installation.

For further detail, see the text in the installed `taskFinish.m` file.

## See Also

### Functions

`jobStartup`, `poolStartup`, `taskStartup`

### Properties

`FileDependencies`, `PathDependencies`

# taskStartup

## Purpose

File for user-defined options to run 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.

The file resides in the worker's MATLAB installation at

```
matlabroot/toolbox/distcomp/user/taskStartup.m
```

You add MATLAB code to the file to define task initialization actions to be performed on the worker every time it evaluates a task for this job.

Alternatively, you can create a file called `taskStartup.m` and include it as part of the job's `FileDependencies` property. The version of the file in `FileDependencies` takes precedence over the version in the worker's MATLAB installation.

For further detail, see the text in the installed `taskStartup.m` file.

## See Also

### Functions

`jobStartup`, `poolStartup`, `taskFinish`

### Properties

`FileDependencies`, `PathDependencies`

## Purpose

Wait for job to finish or change state

## Syntax

```
wait(obj)
wait(obj, 'state')
wait(obj, 'state', timeout)
```

## Arguments

| | |
|---|---|
| `obj` | Job object whose change in state to wait for. |
| `'state'` | Value of the job object's `State` property to wait for. |
| `timeout` | Maximum time to wait, in seconds. |

## Description

`wait(obj)` blocks execution in the client session until the job identified by the object `obj` reaches the `'finished'` state or fails. This occurs when all the job's tasks are finished processing on remote workers.

`wait(obj, '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(job, 'queued')` for a job already in the `'finished'` state, the call returns immediately.

`wait(obj, '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 distributed or parallel jobs, you cannot use `wait`.

---

## Examples

Submit a job to the queue, and wait for it to finish running before retrieving its results.

```
submit(job);
wait(job, 'finished')
results = getAllOutputArguments(job)
```

Submit a batch job and wait for it to finish before retrieving its variables.

```
job = batch('myScript');
wait(job)
load(job)
```

## See Also

pause, resume, 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 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.

**Note** 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`

14

# Property Reference


| | |
|---|---|
| Job Manager Properties (p. 14-2) | Control job manager objects |
| Scheduler Properties (p. 14-3) | Control scheduler objects |
| Job Properties (p. 14-5) | Control job objects |
| Task Properties (p. 14-6) | Control task objects |
| Worker Properties (p. 14-8) | Control worker objects |

# 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 |
| `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 |
| `State` | Current state of task, job, job manager, or worker |
| `Type` | Type of scheduler object |
| `UserData` | Specify data to associate with object |

# 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 |
| `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 directory 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 |
| `ParallelSubmission-WrapperScript` | 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 |

## Job 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 |
| `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 |

## Task 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 |

## Worker 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 |

15

# Properties — Alphabetical List

# AttemptedNumberOfRetries

## 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

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

## 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 "Managing Jobs" on page 8-48.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | Function handle |

## Values

You can set `CancelJobFcn` to any valid function handle.

## See Also

### Functions

`cancel`, `getJobSchedulerData`, `setJobSchedulerData`

### Properties

`CancelTaskFcn`, `DestroyJobFcn`, `DestroyTaskFcn`

# CancelTaskFcn

## 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 "Managing Jobs" on page 8-48.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | Function handle |

## Values

You can set `CancelTaskFcn` to any valid 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 pathname 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 directory 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

## 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

## 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

## 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

## 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 "Programming with User Configurations" on page 6-16.

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`

# CreateTime

## 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 `CreateTime`.

```
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

## 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 directory 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.

## 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 "Managing Jobs" on page 8-48.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | Function handle |

## Values

You can set `DestroyJobFcn` to any valid function handle.

## See Also

### Functions

`destroy`, `getJobSchedulerData`, `setJobSchedulerData`

### Properties

`CancelJobFcn`, `CancelTaskFcn`, `DestroyTaskFcn`

# DestroyTaskFcn

## 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 "Managing Jobs" on page 8-48.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | Function handle |

## Values

You can set `DestroyTaskFcn` to any valid 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

## 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 information about returned information.

## Characteristics

| | |
|---|---|
| Usage | Task object |
| Read-only | Always |
| Data type | Structure |

## Values

`Error` is empty before an attempt to run a task. `Error` remains empty if the evaluation of a task object's function does not produce an error or if a task does not complete because of cancellation or worker crash.

## See Also

### Properties

`ErrorIdentifier`, `ErrorMessage`, `Function`

# ErrorIdentifier

## Purpose

Task error identifier

## Description

If an error occurs during the task evaluation, `ErrorIdentifier` contains the `identifier` property of the `MException` thrown. `ErrorIdentifier` can also indicate if the task did not complete because of cancellation or worker crash.

## 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 object's function does not produce an error or if the error did not provide an identifier. If a task completes, `ErrorIdentifier` has the same value as the `identifier` field of the `Error` property. If a task does not complete because of cancellation or a worker crash, `ErrorIdentifier` is set to indicate that fact, and the `Error` property is left empty.

## See Also

### Properties

`Error`, `ErrorMessage`, `Function`

## Purpose

Message from task error

## Description

If an error occurs during the task evaluation, `ErrorMessage` contains the `message` property of the `MException` thrown. `ErrorMessage` can also indicate if the task did not complete because of cancellation or worker crash.

## 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's function does not produce an error or if the error did not provide a message. If a task completes, `ErrorMessage` has the same value as the `message` field of the `Error` property. If a task does not complete because of cancellation or a worker crash, `ErrorMessage` is set to indicate that fact, and the `Error` property is left empty.

## Examples

Retrieve the error message from a task object.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
a = [1 2 3 4]; %Note: matrix not square
t = createTask(j, @inv, 1, {a});
submit(j)
get(t,'ErrorMessage')
ans =
Error using ==> inv
Matrix must be square.
```

## See Also

### Properties

`Error`, `ErrorIdentifier`, `Function`

# FailedAttemptInformation

## 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 directory 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 directory 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 directory 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 directory 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.

# FileDependencies

## 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 `FileDependencies` property. For more information on this limit, see "Object Data Size Limitations" on page 6-45. For alternative means of making data available to workers, see "Sharing Code" on page 8-28.

## 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

## 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.

---

**Note** The `FinishedFcn` property is available only when using the MathWorks job manager as your scheduler.

---

## Characteristics

| | |
|---|---|
| Usage | Task object or job object |
| Read-only | Never |
| Data type | Callback |

## 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

## 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.

## 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

## 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 "Managing Jobs" on page 8-48.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | Function handle |

## Values

You can set `GetJobStateFcn` to any valid function handle.

## See Also

### Functions

`get`, `getJobSchedulerData`, `setJobSchedulerData`

### Properties

`State`, `SubmitFcn`

# HasSharedFilesystem

## 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`

## 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

## 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

## 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

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`

## 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`

# JobManager

## 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`

## 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`

## 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`

# LabGrid

## 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`

## 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`

# MatlabCommandToRun

## 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`

## 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

## 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`

## 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`

# MpiexecFileName

## 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`

## 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`

## 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`

# 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`

## 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`

# Orientation

## 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`

## 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.

# OutputArguments

## See Also

### Functions

`createTask`, `getAllOutputArguments`

### Properties

`Function`, `InputArguments`, `NumberOfOutputArguments`

## 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`

# ParallelSubmitFcn

## 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 8-34.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | String |

## 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 "Using the Generic Scheduler Interface" on page 9-8.

## See Also

### Functions

`createParallelJob`, `submit`

### Properties

`MatlabCommandToRun`, `SubmitFcn`

## 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`

# Partition

## 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`

## 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 "Sharing Code" on page 8-28.

## 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`

## 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`

# PreviousTask

## 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`

## 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.

---

**Note** The `QueuedFcn` property is available only when using the MathWorks job manager as your scheduler.

---

## Characteristics

| | |
|---|---|
| Usage | Job object |
| Read-only | Never |
| Data type | Callback |

## 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

## 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

## 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.

---

**Note** The `RunningFcn` property is available only when using the MathWorks job manager as your scheduler.

---

## Characteristics

| | |
|---|---|
| Usage | Task object or job object |
| Read-only | Never |
| Data type | Callback |

## 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`

# ServerName

## 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`

## 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

## 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

## 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 8-34.

## Characteristics

| | |
|---|---|
| Usage | Generic scheduler object |
| Read-only | Never |
| Data type | String |

## 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 "Using the Generic Scheduler Interface" on page 8-33.

## 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

## 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

## 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`

# Timeout

## 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

## 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

## Description

The `UserName` property value is a string indicating the login name of the user who created the job.

## Characteristics

| | |
|---|---|
| Usage | Job object |
| Read-only | Always |
| Data type | String |

## Examples

Examine a job to see who created it.

```
get(job1, 'UserName')
ans =
jsmith
```

# UseSOAJobSubmission

## 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

## 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`

# WorkerMachineOsType

# 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.

**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.

**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 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

## A


arrays
  codistributed 5-4
  local 5-11
  private 5-4
  replicated 5-2
  types of 5-2
  variant 5-3
AttemptedNumberOfRetries property 15-2


## B


batch function 13-2
BlockSize property 15-3
BusyWorkers property 15-4


## C


cancel function 13-5
CancelJobFcn property 15-5
CancelTaskFcn property 15-6
CaptureCommandWindowOutput property 15-7
ccsscheduler object 11-2
clear function 13-7
ClusterMatlabRoot property 15-9
ClusterName property 15-10
ClusterOsType property 15-11
ClusterSize property 15-12
ClusterVersion property 15-13
codistributed arrays
  constructor functions 5-10
  creating 5-7
  defined 5-4
  indexing 5-15
  working with 5-5
codistributed function 13-8
codistributed object 11-4
codistributed.build function 13-10
codistributed.cell function 13-12
codistributed.colon function 13-14
codistributed.eye function 13-16
codistributed.false function 13-18
codistributed.inf function 13-20
codistributed.nan function 13-22
codistributed.ones function 13-24
codistributed.rand function 13-26
codistributed.randn function 13-28
codistributed.spalloc function 13-30
codistributed.speye function 13-32
codistributed.sprand function 13-34
codistributed.sprandn function 13-36
codistributed.true function 13-38
codistributed.zeros function 13-40
codistributor function 13-42
codistributor1d function 13-45
codistributor1d object 11-6
codistributor1d.defaultPartition
  function 13-48
codistributor2dbc function 13-49
codistributor2dbc object 11-7
codistributor2dbc.defaultBlockSize
  property 15-14
codistributor2dbc.defaultLabGrid
  function 13-51
CommandWindowOutput property 15-15
Composite
  getting started 1-10
  outside spmd 3-10
Composite function 13-52
Composite object 11-8
Computer property 15-17
Configuration property 15-18
configurations 6-16
  importing and exporting 6-23
  using in application 6-27
  validating 6-24
  with MATLAB Compiler 6-24
createJob function 13-53
createMatlabPoolJob function 13-55
createParallelJob function 13-57

createTask function 13-60
CreateTime property 15-20
current working directory
    MATLAB worker 6-29
CurrentJob property 15-21
CurrentTask property 15-22

## D

DataLocation property 15-23
defaultParallelConfig function 13-63
demote function 13-65
destroy function 13-67
DestroyJobFcn property 15-25
DestroyTaskFcn property 15-26
dfeval function 13-68
dfevalasync function 13-72
diary function 13-74
Dimension property 15-27
distributed function 13-75
distributed object 11-10
distributed.cell function 13-76
distributed.eye function 13-77
distributed.false function 13-78
distributed.inf function 13-79
distributed.nan function 13-80
distributed.ones function 13-81
distributed.rand function 13-82
distributed.randn function 13-83
distributed.spalloc function 13-84
distributed.speye function 13-85
distributed.sprand function 13-86
distributed.sprandn function 13-87
distributed.true function 13-88
distributed.zeros function 13-89
dload function 13-90
drange operator
    for loop 13-102
dsave function 13-92

## E

EnvironmentSetMethod property 15-28
Error property 15-29
ErrorIdentifier property 15-30
ErrorMessage property 15-31
exist function 13-93

## F

FailedAttemptInformation property 15-32
FileDependencies property 15-33
files
    sharing 8-13
        using an LSF scheduler 8-28
findJob function 13-94
findResource function 13-96
findTask function 13-100
FinishedFcn property 15-36
FinishTime property 15-38
for loop
    distributed 13-102
Function property 15-40
functions
    batch 13-2
    cancel 13-5
    clear 13-7
    codistributed 13-8
    codistributed.build 13-10
    codistributed.cell 13-12
    codistributed.colon 13-14
    codistributed.eye 13-16
    codistributed.false 13-18
    codistributed.inf 13-20
    codistributed.nan 13-22
    codistributed.ones 13-24
    codistributed.rand 13-26
    codistributed.randn 13-28
    codistributed.spalloc 13-30
    codistributed.speye 13-32
    codistributed.sprand 13-34

codistributed.sprandn 13-36
codistributed.true 13-38
codistributed.zeros 13-40
codistributor 13-42
codistributor1d 13-45
codistributor1d.defaultPartition 13-48
codistributor2dbc 13-49
codistributor2dbc.defaultLabGrid 13-51
Composite 13-52
createJob 13-53
createMatlabPoolJob 13-55
createParallelJob 13-57
createTask 13-60
defaultParallelConfig 13-63
demote 13-65
destroy 13-67
dfeval 13-68
dfevalasync 13-72
diary 13-74
distributed 13-75
distributed.cell 13-76
distributed.eye 13-77
distributed.false 13-78
distributed.inf 13-79
distributed.nan 13-80
distributed.ones 13-81
distributed.rand 13-82
distributed.randn 13-83
distributed.spalloc 13-84
distributed.speye 13-85
distributed.sprand 13-86
distributed.sprandn 13-87
distributed.true 13-88
distributed.zeros 13-89
dload 13-90
dsave 13-92
exist 13-93
findJob 13-94
findResource 13-96
findTask 13-100
for
    distributed 13-102
    drange 13-102
gather 13-104
gcat 13-106
get 13-107
getAllOutputArguments 13-109
getCodistributor 13-111
getCurrentJob 13-113
getCurrentJobmanager 13-114
getCurrentTask 13-115
getCurrentWorker 13-116
getDebugLog 13-117
getFileDependencyDir 13-119
getJobSchedulerData 13-120
getLocalPart 13-121
globalIndices 13-122
gop 13-124
gplus 13-126
help 13-127
importParallelConfig 13-128
inspect 13-130
isaUnderlying 13-132
iscodistributed 13-133
isComplete 13-134
isdistributed 13-135
isreplicated 13-136
jobStartup 13-137
labBarrier 13-138
labBroadcast 13-139
labindex 13-141
labProbe 13-142
labReceive 13-143
labSend 13-145
labSendReceive 13-146
length 13-149
load 13-150
matlabpool 13-152
methods 13-157
mpiLibConf 13-159

mpiprofile 13-161
mpiSettings 13-166
numlabs 13-168
parfor 13-169
pause 13-173
pctconfig 13-174
pctRunDeployedCleanup 13-176
pctRunOnAll 13-177
pload 13-178
pmode 13-180
poolStartup 13-183
promote 13-185
psave 13-187
redistribute 13-189
resume 13-190
set 13-191
setJobSchedulerData 13-194
setupForParallelExecution 13-195
size 13-197
sparse 13-198
spmd 13-200
submit 13-202
subsasgn 13-203
subsref 13-204
taskFinish 13-205
taskStartup 13-206
wait 13-207
waitForState 13-209

## G

gather function 13-104
gcat function 13-106
generic scheduler
distributed jobs 8-33
parallel jobs 9-8
genericscheduler object 11-12
get function 13-107
getAllOutputArguments function 13-109
getCodistributor function 13-111
getCurrentJob function 13-113
getCurrentJobmanager function 13-114
getCurrentTask function 13-115
getCurrentWorker function 13-116
getDebugLogp function 13-117
getFileDependencyDir function 13-119
getJobSchedulerData function 13-120
GetJobStateFcn property 15-41
getLocalPart function 13-121
globalIndices function 13-122
gop function 13-124
gplus function 13-126

## H

HasSharedFilesystem property 15-42
help
command-line 6-10
help function 13-127
HostAddress property 15-43
HostName property 15-44

## I

ID property 15-45
IdleWorkers property 15-47
importParallelConfig function 13-128
InputArguments property 15-48
inspect function 13-130
isaUnderlying function 13-132
iscodistributed function 13-133
isComplete function 13-134
isdistributed function 13-135
isreplicated function 13-136

## J

job
creating
example 8-10
creating on generic scheduler

example 8-44
creating on LSF or HPC Server scheduler
example 8-24
life cycle 6-14
local scheduler 8-3
submitting to generic scheduler queue 8-46
submitting to local scheduler 8-5
submitting to LSF or HPC Server scheduler queue 8-26
submitting to queue 8-12
job manager
finding
example 8-3 8-8
job object 11-15
JobData property 15-49
JobDescriptionFile property 15-51
jobmanager object 11-18
JobManager property 15-52
Jobs property 15-53
jobStartup function 13-137
JobTemplate property 15-55


## L


labBarrier function 13-138
labBroadcast function 13-139
LabGrid property 15-56
labindex function 13-141
labProbe function 13-142
labReceive function 13-143
labSend function 13-145
labSendReceive function 13-146
length function 13-149
load function 13-150
localscheduler object 11-21
LSF scheduler 8-20
lsfscheduler object 11-23


## M


MasterName property 15-57
MatlabCommandToRun property 15-58
matlabpool
parfor 2-3
spmd 3-3
matlabpool function 13-152
matlabpooljob object 11-25
MaximumNumberOfRetries property 15-59
MaximumNumberOfWorkers property 15-60
methods function 13-157
MinimumNumberOfWorkers property 15-61
mpiexec object 11-28
MpiexecFileName property 15-62
mpiLibConf function 13-159
mpiprofile function 13-161
mpiSettings function 13-166


## N


Name property 15-63
NumberOfBusyWorkers property 15-65
NumberOfIdleWorkers property 15-66
NumberOfOutputArguments property 15-67
numlabs function 13-168


## O


objects 6-7
ccsscheduler 11-2
codistributed 11-4
codistributor1d 11-6
codistributor2dbc 11-7
Composite 11-8
distributed 11-10
genericscheduler 11-12
job 11-15
jobmanager 11-18
localscheduler 11-21
lsfscheduler 11-23

matlabpooljob 11-25
mpiexec 11-28
paralleljob 11-30
pbsproscheduler 11-33
saving or sending 6-29
simplejob 11-35
simplematlabpooljob 11-38
simpleparalleljob 11-41
simpletask 11-44
task 11-46
torquescheduler 11-49
worker 11-51
Orientation property 15-68
OutputArguments property 15-69


## P


parallel for-loops. *See* parfor-loops
parallel jobs 9-2
supported schedulers 9-4
paralleljob object 11-30
ParallelSubmissionWrapperScript property 15-71
ParallelSubmitFcn property 15-72
Parent property 15-73
parfor function 13-169
parfor-loops 2-1
break 2-11
broadcast variables 2-20
classification of variables 2-14
compared to for-loops 2-5
error handling 2-8
for-drange 2-13
global variables 2-11
improving performance 2-28
limitations 2-9
local vs. cluster workers 2-12
loop variable 2-15
MATLAB path 2-8
nested functions 2-11
nested loops 2-11
nesting with spmd 2-11
nondistributable functions 2-11
persistent variables 2-11
programming considerations 2-8
reduction assignments 2-21
reduction assignments, associativity 2-23
reduction assignments, commutativity 2-24
reduction assignments, overloading 2-25
reduction variables 2-20
release compatibility 2-13
return 2-11
sliced variables 2-16
temporary variables 2-26
transparency 2-9
Partition property 15-74
PathDependencies property 15-75
pause function 13-173
PBS Pro scheduler 8-20
pbsproscheduler object 11-33
pctconfig function 13-174
pctRunDeployedCleanup function 13-176
pctRunOnAll function 13-177
platforms
supported 6-7
pload function 13-178
pmode function 13-180
poolStartup function 13-183
PreviousJob property 15-77
PreviousTask property 15-78
programming
basic session 8-8
guidelines 6-12
local scheduler 8-2
tips 6-29
promote function 13-185
properties
AttemptedNumberOfRetries 15-2
BlockSize 15-3
BusyWorkers 15-4

CancelJobFcn 15-5
CancelTaskFcn 15-6
CaptureCommandWindowOutput 15-7
ClusterMatlabRoot 15-9
ClusterName 15-10
ClusterOsType 15-11
ClusterSize 15-12
ClusterVersion 15-13
codistributor2dbc.defaultBlockSize 15-14
CommandWindowOutput 15-15
Computer 15-17
Configuration 15-18
CreateTime 15-20
CurrentJob 15-21
CurrentTask 15-22
DataLocation 15-23
DestroyJobFcn 15-25
DestroyTaskFcn 15-26
Dimension 15-27
EnvironmentSetMethod 15-28
Error 15-29
ErrorIdentifier 15-30
ErrorMessage 15-31
FailedAttemptInformation 15-32
FileDependencies 15-33
FinishedFcn 15-36
FinishTime 15-38
Function 15-40
GetJobStateFcn 15-41
HasSharedFilesystem 15-42
HostAddress 15-43
HostName 15-44
ID 15-45
IdleWorkers 15-47
InputArguments 15-48
JobData 15-49
JobDescriptionFile 15-51
JobManager 15-52
Jobs 15-53
JobTemplate 15-55
LabGrid 15-56
MasterName 15-57
MatlabCommandToRun 15-58
MaximumNumberOfRetries 15-59
MaximumNumberOfWorkers 15-60
MinimumNumberOfWorkers 15-61
MpiexecFileName 15-62
Name 15-63
NumberOfBusyWorkers 15-65
NumberOfIdleWorkers 15-66
NumberOfOutputArguments 15-67
Orientation 15-68
OutputArguments 15-69
ParallelSubmissionWrapperScript 15-71
ParallelSubmitFcn 15-72
Parent 15-73
Partition 15-74
PathDependencies 15-75
PreviousJob 15-77
PreviousTask 15-78
QueuedFcn 15-79
RcpCommand 15-81
ResourceTemplate 15-82
RestartWorker 15-83
RshCommand 15-84
RunningFcn 15-85
SchedulerHostname 15-87
ServerName 15-88
StartTime 15-89
State 15-91
SubmitArguments 15-94
SubmitFcn 15-96
SubmitTime 15-97
Tag 15-98
Task 15-99
Tasks 15-100
Timeout 15-102
Type 15-104
UserData 15-105
UserName 15-107

UseSOAJobSubmission 15-108
Worker 15-110
WorkerMachineOsType 15-111
psave function 13-187

## Q

QueuedFcn property 15-79

## R

RcpCommand property 15-81
redistribute function 13-189
ResourceTemplate property 15-82
RestartWorker property 15-83
results
local scheduler 8-6
retrieving 8-13
retrieving from job on generic scheduler 8-46
retrieving from job on LSF scheduler 8-27
resume function 13-190
RshCommand property 15-84
RunningFcn property 15-85

## S

saving
objects 6-29
scheduler
generic interface
distributed jobs 8-33
parallel jobs 9-8
HPC Server 8-20
finding, example 8-22
LSF 8-20
finding, example 8-21
PBS Pro 8-20
TORQUE 8-20
SchedulerHostname property 15-87
ServerName property 15-88
set function 13-191
setJobSchedulerData function 13-194
setupForParallelExecution function 13-195
simplejob object 11-35
simplematlabpooljob object 11-38
simpleparalleljob object 11-41
simpletask object 11-44
single program multiple data. *See* spmd
size function 13-197
sparse function 13-198
spmd 3-1
break 3-17
error handling 3-15
getting started 1-10
global variables 3-17
limitations 3-15
MATLAB path 3-15
nested functions 3-16
nested spmd 3-17
nesting with parfor 3-17
persistent variables 3-17
programming considerations 3-15
return 3-17
transparency 3-15
spmd function 13-200
StartTime property 15-89
State property 15-91
submit function 13-202
SubmitArguments property 15-94
SubmitFcn property 15-96
SubmitTime property 15-97
subsasgn function 13-203
subsref function 13-204

## T

Tag property 15-98
task
creating
example 8-11
creating on generic scheduler

example 8-45
creating on LSF scheduler
example 8-25
local scheduler 8-5
task object 11-46
Task property 15-99
taskFinish function 13-205
Tasks property 15-100
taskStartup function 13-206
Timeout property 15-102
TORQUE scheduler 8-20
torquescheduler object 11-49
troubleshooting
programs 6-45
Type property 15-104


## U


user configurations 6-16
UserData property 15-105
UserName property 15-107
UseSOAJobSubmission property 15-108


## W


wait function 13-207
waitForState function 13-209
Windows HPC Server scheduler 8-20
worker object 11-51
Worker property 15-110
WorkerMachineOsType property 15-111