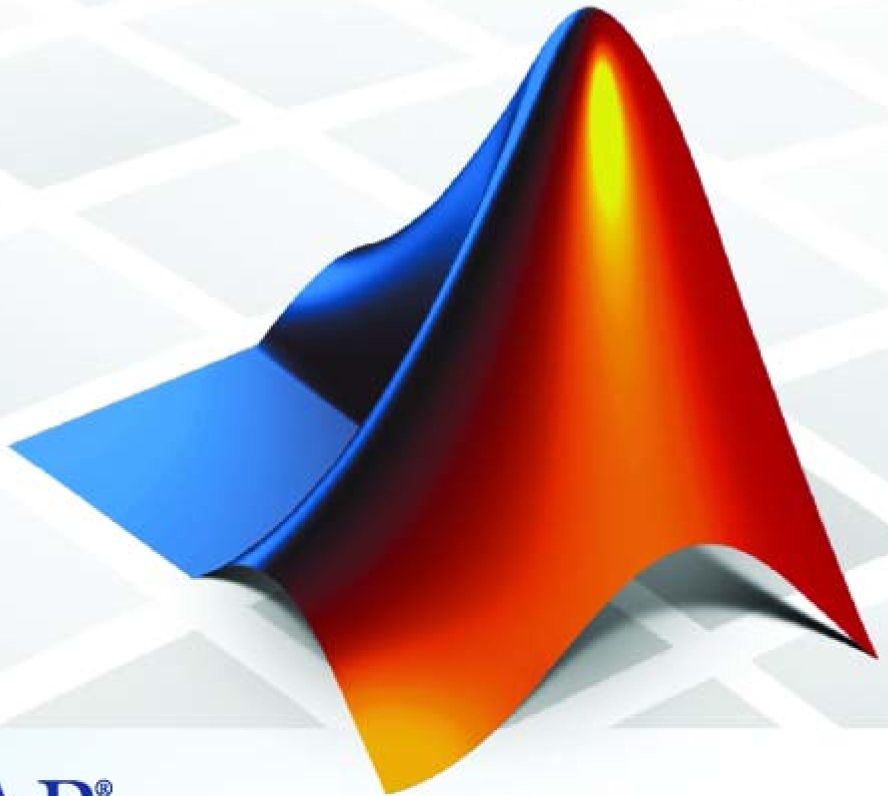
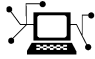
Distributed Computing Toolbox 3
User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

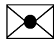


www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

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.

Distributed Computing Toolbox User's Guide

© COPYRIGHT 2004–2007 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

Other product or brand names are 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)

Contents

1 Getting Started

What Are the Distributed Computing Products? **1-2**
Determining Product Installation and Versions **1-3**

Toolbox and Engine Components **1-4**
Job Managers, Workers, and Clients **1-4**
Local Scheduler **1-6**
Third-Party Schedulers **1-6**
Components on Mixed Platforms or Heterogeneous Clusters **1-7**
MATLAB Distributed Computing Engine Service **1-8**
Components Represented in the Client **1-8**

Using Distributed Computing Toolbox **1-9**
Example: Evaluating a Basic Function **1-9**
Example: Programming a Basic Job with a Local Scheduler **1-9**

Getting Help **1-11**
Command-Line Help **1-11**
Help Browser **1-12**

2 Programming Overview

Program Development Guidelines **2-2**

Life Cycle of a Job **2-4**

Programming with User Configurations **2-6**
Defining Configurations **2-6**

Exporting and Importing Configurations 2-12
Applying Configurations in Client Code 2-12

Programming Tips and Notes 2-15
Saving or Sending Objects 2-15
Current Working Directory of a MATLAB Worker 2-15
Using clear functions 2-16
Running Tasks That Call Simulink 2-16
Using the pause Function 2-16
Transmitting Large Amounts of Data 2-16
Interrupting a Job 2-16
IPv6 on Macintosh 2-17
Speeding Up a Job 2-17

Using the Parallel Profiler 2-18
Introduction 2-18
Collecting Parallel Profile Data 2-18
Viewing Parallel Profile Data 2-19

Troubleshooting and Debugging 2-29
Object Data Size Limitations 2-29
File Access and Permissions 2-31
No Results or Failed Job 2-33
Connection Problems Between the Client and Job Manager 2-34

3 Parallel for-Loops (parfor)

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

Programming Considerations 3-7
MATLAB Path 3-7

Error Handling 3-7
Limitations 3-8
Performance Considerations 3-10
Compatibility with Earlier Versions of MATLAB 3-11

Advanced Topics **3-12**
About Programming Notes 3-12
Classification of Variables 3-12
Improving Performance 3-26

4 Interactive Parallel Mode (pmode)

Introduction **4-2**

Getting Started with Interactive Parallel Mode **4-3**

Parallel Command Window **4-11**

Running pmode on a Cluster **4-17**

Plotting in pmode **4-18**

Limitations and Unexpected Results **4-20**
Distributing Nonreplicated Arrays 4-20
Using Graphics in pmode 4-21

Troubleshooting **4-22**
Hostname Resolution 4-22
Socket Connections 4-22

5 Evaluating Functions in a Cluster

Evaluating Functions Synchronously **5-2**
Scope of dfeval **5-2**
Arguments of dfeval **5-3**
Example — Using dfeval **5-4**

Evaluating Functions Asynchronously **5-8**

6 Programming Distributed Jobs

Using a Local Scheduler **6-2**
Creating and Running Jobs with a Local Scheduler **6-2**
Local Scheduler Behavior **6-6**

Using a Job Manager **6-7**
Creating and Running Jobs with a Job Manager **6-7**
Sharing Code **6-12**
Managing Objects in the Job Manager **6-14**

Using a Fully Supported Third-Party Scheduler **6-18**
Creating and Running Jobs with an LSF or CCS Scheduler **6-18**
Sharing Code **6-25**
Managing Objects **6-27**

Using the Generic Scheduler Interface **6-30**
Overview **6-30**
MATLAB Client Submit Function **6-31**
Example — Writing the Submit Function **6-35**
MATLAB Worker Decode Function **6-36**
Example — Writing the Decode Function **6-38**
Example — Programming and Running a Job in the Client **6-39**
Supplied Submit and Decode Functions **6-44**
Summary **6-45**

7 Programming Parallel Jobs

Introduction **7-2**

Using a Supported Scheduler **7-4**
Coding the Task Function **7-4**
Coding in the Client **7-5**

Using the Generic Scheduler Interface **7-7**
Introduction **7-7**
Coding in the Client **7-7**

Further Notes on Parallel Jobs **7-10**
Number of Tasks in a Parallel Job **7-10**
Avoiding Deadlock and Other Dependency Errors **7-10**

8 Parallel Math

Array Types **8-2**
Introduction **8-2**
Nondistributed Arrays **8-2**
Distributed Arrays **8-4**

Working with Distributed Arrays **8-5**
How MATLAB Distributes Arrays **8-5**
Creating a Distributed Array **8-7**
Local Arrays **8-10**
Obtaining Information About the Array **8-11**
Changing the Dimension of Distribution **8-13**
Restoring the Full Array **8-14**
Indexing into a Distributed Array **8-15**

Using a for-Loop Over a Distributed Range (for-drange) **8-17**
Parallelizing a for-Loop **8-17**
Distributed Arrays in a for-drange Loop **8-18**

Using MATLAB Functions on Distributed Arrays 8-20

9 Objects — By Category

Scheduler Objects 9-2

Job Objects 9-2

Task Objects 9-3

Worker Objects 9-3

10 Objects — Alphabetical List

11 Functions — By Category

General Toolbox Functions 11-2

Job Manager Functions 11-3

Scheduler Functions 11-3

Job Functions 11-4

Task Functions 11-4

Toolbox Functions Used in Parallel Jobs and pmode 11-5

Toolbox Functions Used in MATLAB Workers 11-7

12 Functions — Alphabetical List

13 Properties — By Category

Job Manager Properties 13-2

Scheduler Properties 13-3

Job Properties 13-4

Task Properties 13-6

Worker Properties 13-7

14 Properties — Alphabetical List

Glossary

Index

Getting Started

This chapter provides information you need to get started with Distributed Computing Toolbox and MATLAB® Distributed Computing Engine. The sections are as follows.

What Are the Distributed Computing Products? (p. 1-2)	Overview of Distributed Computing Toolbox and MATLAB Distributed Computing Engine, and their capabilities
Toolbox and Engine Components (p. 1-4)	Descriptions of the parts and configurations of a distributed computing setup
Using Distributed Computing Toolbox (p. 1-9)	Introduction to Distributed Computing Toolbox programming with a basic example
Getting Help (p. 1-11)	Explanation of how to get help on toolbox functions

What Are the Distributed Computing Products?

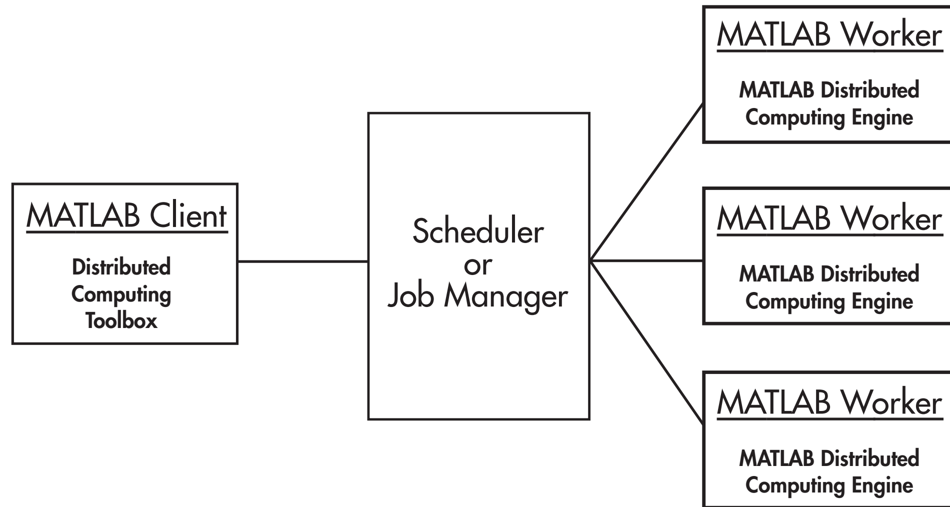
Distributed Computing Toolbox and MATLAB Distributed Computing Engine enable you to coordinate and execute independent MATLAB operations simultaneously on a cluster of computers, speeding up execution of large MATLAB jobs.

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 Distributed Computing Toolbox to perform the definition of jobs and tasks. MATLAB Distributed Computing Engine 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 engine'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 Windows CCS or Platform LSF.

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



Basic Distributed Computing Configuration

Determining Product Installation and Versions

To determine if Distributed Computing Toolbox 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.

You can run the `ver` command as part of a task in a distributed application to determine what version of MATLAB Distributed Computing Engine is installed on a worker machine. Note that the toolbox and engine must be the same version.

Toolbox and Engine Components

In this section...

"Job Managers, Workers, and Clients" on page 1-4

"Local Scheduler" on page 1-6

"Third-Party Schedulers" on page 1-6

"Components on Mixed Platforms or Heterogeneous Clusters" on page 1-7

"MATLAB Distributed Computing Engine Service" on page 1-8

"Components Represented in the Client" on page 1-8

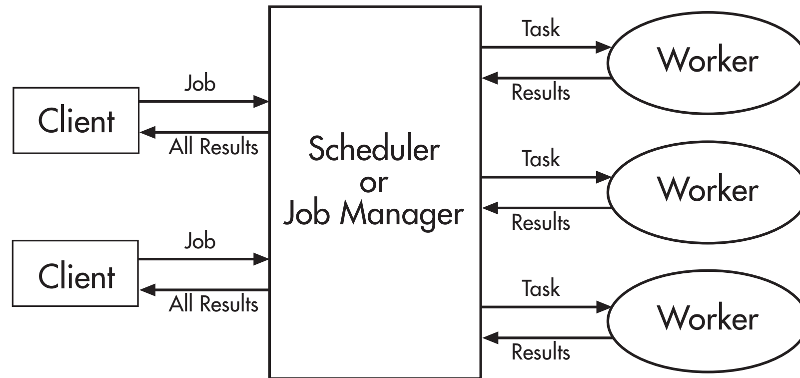
Job Managers, Workers, and Clients

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.

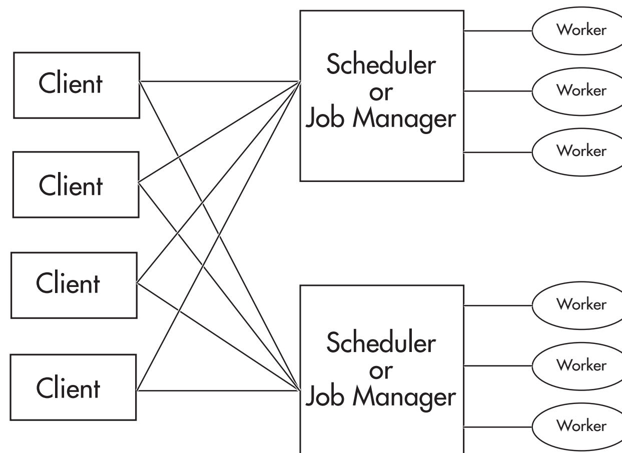
A MATLAB Distributed Computing Engine setup usually includes many workers that can all execute tasks simultaneously, speeding up execution of large MATLAB jobs. It is generally not important which worker executes a specific task. The workers evaluate tasks one at a time, returning the results to the 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 Distributed 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 Distributed Computing Toolbox is the ability to run a local scheduler and up to four workers on the client machine, so that you can run distributed and parallel jobs without requiring a remote cluster or MATLAB Distributed Computing Engine. 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 Windows CCS, Platform Computing LSF, 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 distributed computing jobs the only cluster scheduling management you need?

  The MathWorks job manager is designed specifically for MathWorks distributed 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 distributed 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

Distributed Computing Toolbox and MATLAB Distributed Computing Engine are supported on Windows, UNIX, and Macintosh platforms. 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.

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.

MATLAB Distributed Computing Engine Service

If you are using the MathWorks job manager, every machine that hosts a worker or job manager session must also run the MATLAB Distributed Computing Engine (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 Engine 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 Distributed Computing Toolbox

In this section...
“Example: Evaluating a Basic Function” on page 1-9
“Example: Programming a Basic Job with a Local Scheduler” on page 1-9

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 5, “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 Distributed 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 6-7 or "Creating and Running Jobs with an LSF or CCS Scheduler" on page 6-18.)

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

2. Create a job. Create job `j` on the scheduler. (For more information, see "Create a Job" on page 6-9.)

   ```
   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 6-10.)

   ```
   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 four at one time. (For more information, see "Submit a Job to the Job Queue" on page 6-11.)

   ```
   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 6-11.)

   ```
   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

In this section...
"Command-Line Help" on page 1-11 "Help Browser" on page 1-12

Command-Line Help

You can get command-line help on the object functions in Distributed Computing Toolbox 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 `jobmanager`, `job`, and `task`.

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
```

Note You must enter the property or function name with lowercase letters, even though function names are case sensitive in other situations.

2

Programming Overview

This chapter provides information you need for programming with Distributed Computing Toolbox. The specifics of evaluating functions in a cluster, programming distributed jobs, and programming parallel jobs are covered in later chapters. This chapter describes features common to all the programming options. The sections are as follows.

Program Development Guidelines (p. 2-2)	Suggested method for program development
Life Cycle of a Job (p. 2-4)	Stages of a job from creation to completion
Programming with User Configurations (p. 2-6)	How to employ configurations for parameters and properties in your program
Programming Tips and Notes (p. 2-15)	Provides helpful hints for good programming practice
Using the Parallel Profiler (p. 2-18)	Describes how to use the parallel profile to determine the calculation and communications time for each lab
Troubleshooting and Debugging (p. 2-29)	Describes common programming errors and how to avoid them

Program Development Guidelines

When writing code for Distributed Computing Toolbox, 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 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 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 interactive parallel mode (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.

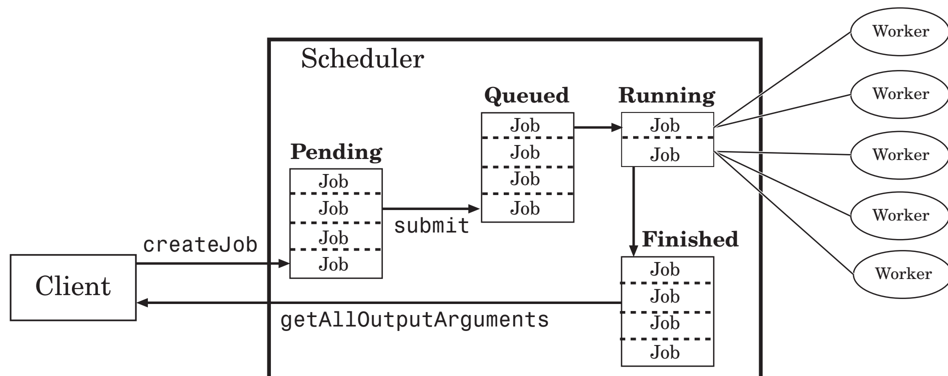
6. **Run the distributed job on only one cluster node.** Run your distributed job with one task to verify that remote distribution is working between your client and the cluster, and to verify file and path dependencies.
7. **Run the distributed or parallel job on multiple cluster nodes.** Scale up your 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 Distributed Computing Toolbox. 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 Distributed Computing Toolbox. The job's first state is `pending`. This is when you define the job by adding tasks to it.

Job Stage	Description
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 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.

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

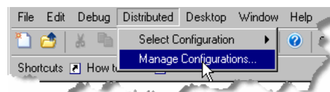
In this section...
"Defining Configurations" on page 2-6
"Exporting and Importing Configurations" on page 2-12
"Applying Configurations in Client Code" on page 2-12

Defining Configurations

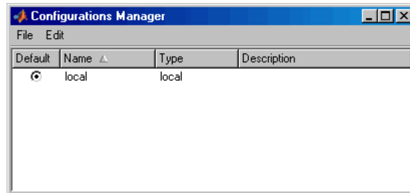
Configurations allow you to define certain parameters and properties, then have your settings applied when creating objects in the MATLAB client. The functions that support the use of configurations are

- `createJob`
- `createParallelJob`
- `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 **Distributed** pull-down menu on the MATLAB desktop. Click **Distributed > 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.

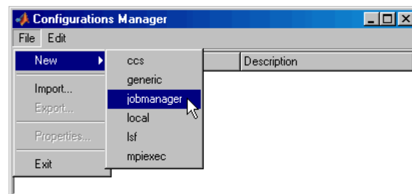


The following example provides instructions on how to create and modify configurations using the Configurations Manager and its menus and dialog boxes.

Example — Creating and Modifying User Configurations

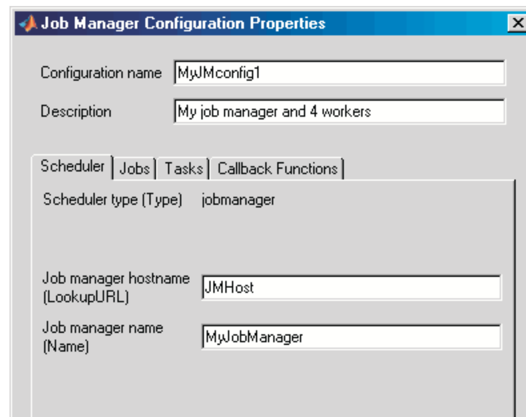
Suppose you want to create a configuration to set several properties for some jobs being run by a job manager.

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

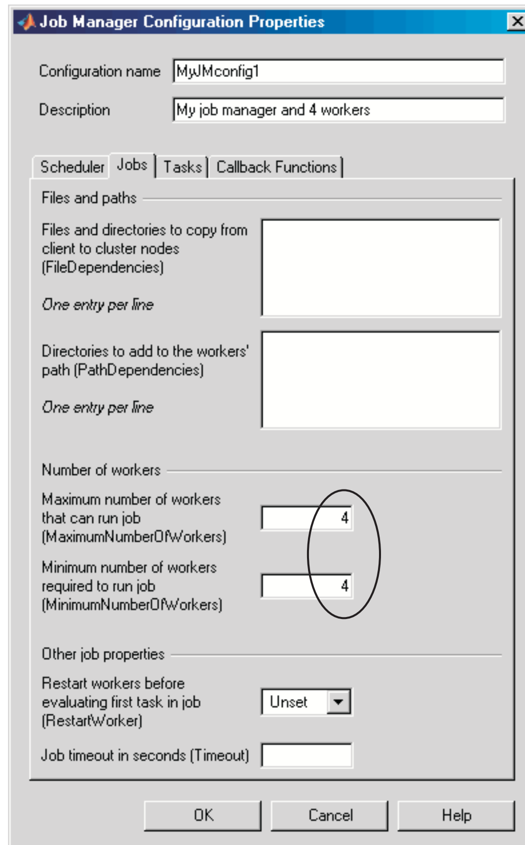


This opens a new Job Manager Configuration Properties dialog box.

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.



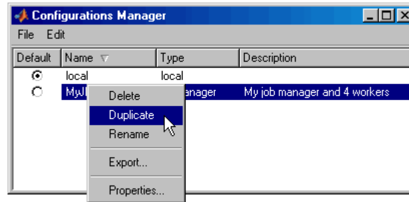
3 In the **Jobs** tab, enter 4 and 4 for the maximum and minimum number of workers. This specifies that for jobs using this configuration, they require at least four workers and use no more than four workers. Therefore, the job runs on exactly four workers, even if it has to wait until four workers are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

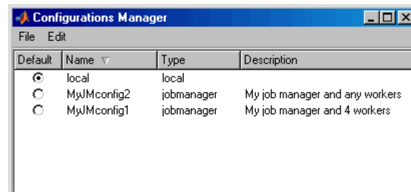
a In the Configurations Manager, right-click the configuration `MyJMconfig1` in the list and select **Duplicate**.



The duplicate configuration is created with a default name, already highlighted for you to edit.

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

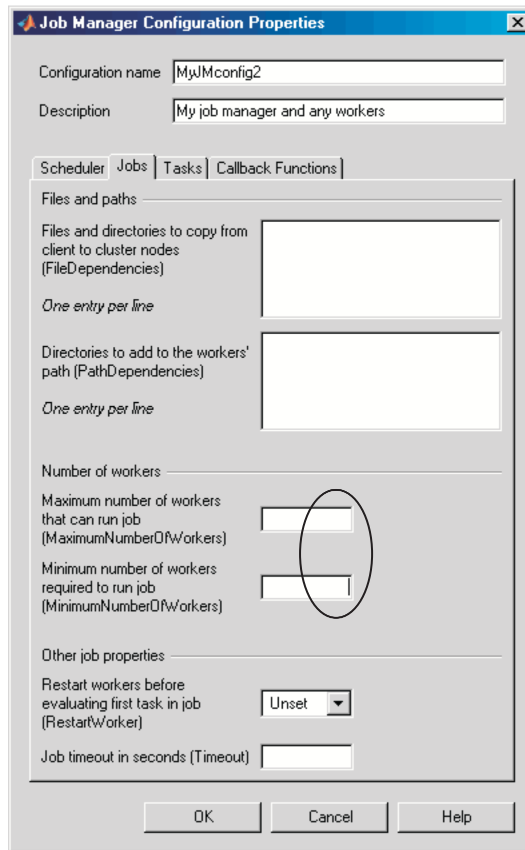
c Click twice, slowly (not a rapid double-click), in the description field of the new configuration. You can now edit the description field to change its text to `My job manager and any workers`.



So far, the new configuration has a new name and description, but all its properties are identical to those of the original `MyJMConfig1` configuration.

6 Double-click the `MyJMconfig2` entry in the configurations list. This opens the properties dialog box for that configuration. The new name and description are already set in the top fields.

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



8 Click **OK** to save the configuration and close the dialog box.

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

Configurations are stored as part of your MATLAB preferences, so they are generally available on an individual user basis. To make a 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 distributed computing cluster can share common configurations.

To export a configuration:

1. In the Configuration 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 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.

Applying Configurations in Client Code

In the MATLAB client where you create and define your distributed 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. Currently, `pmode` and `matlabpool` support default configurations.

There are several ways to specify which of your configurations should be used as the default configuration:

- In the MATLAB desktop, click **Distributed > 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. For example,

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

This command finds the scheduler defined by the settings of the configuration named `our_jobmanager` . 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 third-party schedulers, settable object properties can be defined in the configuration and applied after `findResource` has created the scheduler object. For example,

```
lsfsched = findResource('scheduler', 'type', 'lsf');
set (lsfsched, 'configuration', 'my_lsf_config');
```

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.

Writing Scheduler-Independent 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,

```
sched = findResource('scheduler', 'configuration', 'MyConfig');
set(sched, 'Configuration', 'MyConfig');
job1 = createJob(sched, 'Configuration', 'MyConfig');
createTask(..., '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.

Programming Tips and Notes

In this section...

"Saving or Sending Objects" on page 2-15

"Current Working Directory of a MATLAB Worker" on page 2-15

"Using clear functions" on page 2-16

"Running Tasks That Call Simulink" on page 2-16

"Using the pause Function" on page 2-16

"Transmitting Large Amounts of Data" on page 2-16

"Interrupting a Job" on page 2-16

"IPv6 on Macintosh" on page 2-17

"Speeding Up a Job" on page 2-17

Saving or Sending Objects

Do not use the `save` or `load` function on Distributed 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 distributed computing object between distributed 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 Distributed 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 6-15.

Running Tasks That Call Simulink

The first task that runs on a worker session that uses Simulink® 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 machines, `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`.

IPv6 on Macintosh

To allow multicast access between different distributed computing processes run by different users on the same Macintosh computer, IPv6 addressing is disabled for MATLAB with Distributed Computing Toolbox on a Macintosh.

Note Though DCT/MDCE Version 3 continues to support multicast communications between its processes, multicast is not recommended and might not be supported in future releases.

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 2-18

"Collecting Parallel Profile Data" on page 2-18

"Viewing Parallel Profile Data" on page 2-19

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 M-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
```

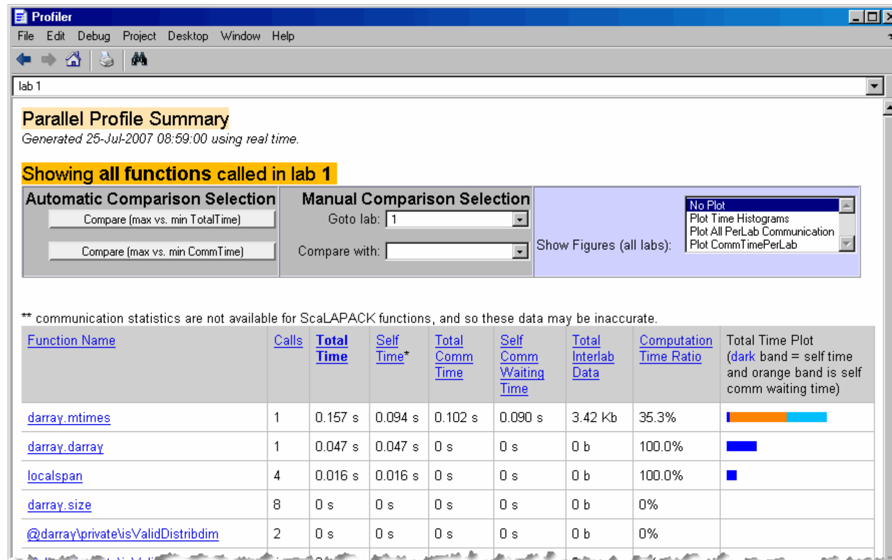
The remainder of this section is an example that illustrates some of the features of the parallel profile viewer. This example executes in a pmode session running on four local labs. Initiate pmode by typing in the MATLAB Command Window:

```
pmode start local 4
```

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

```
P>> R1 = rand(16, darray)
P>> R2 = rand(16, darray)
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
Total Interlab Data	The amount of data transferred to and from this lab for this function

Column Header	Description
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 `darray.mtimes` looks like this:

Profiler
File Edit Debug Project Desktop Window Help
lab 1

darray.mtimes(1 call, 0.157 sec, 1.71 Kb sent, 1.71 Kb rec, 0.090 s wait, 0.012 s act comm)
Total computation took 35.29% of total time.

Generated 25-Jul-2007 09:05:01 using real time.
Showing **this function's** statistics on lab **1**

Automatic Comparison Selection	Manual Comparison Selection	
Compare (max vs. min TotalTime)	Goto lab: 1	Show Figures (all labs): No Plot / Plot Time Histograms / Plot All PerLab Communication / Plot CommTimePerLab
Compare (max vs. min CommTime)	Compare with:	

M-function in file C:\Work\Ami\matlab\toolbox\distcomp\parallel\ops\@darray\mtimes.m
[Copy to new window for comparing multiple runs]

Please note that the code displayed is taken from the client, and might have changed since execution on the cluster. Only valid M-functions will be shown.

Refresh
☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show M-Lint results ☑ Show file coverage ☑ Show function listing

Sort busy lines and graph according to time

Parents (calling functions)
No parent

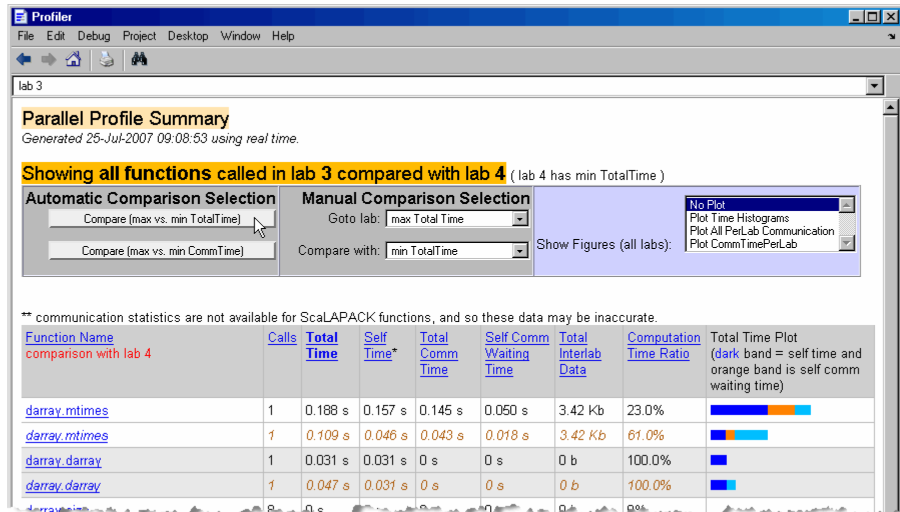
Lines where the most time was spent.

Line Number	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
144	`Aloc = labSendReceive(to, from...`	3	0.094 s	1.71 Kb	1.71 Kb	0.090 s	0.012 s	59.9%	
148	`C = darray(Cloc,2,Bpart);`	1	0.047 s	0 b	0 b	0 s	0 s	29.9%	
138	`k = localspan(Apart,labindex);`	1	0.016 s	0 b	0 b	0 s	0 s	10.2%	

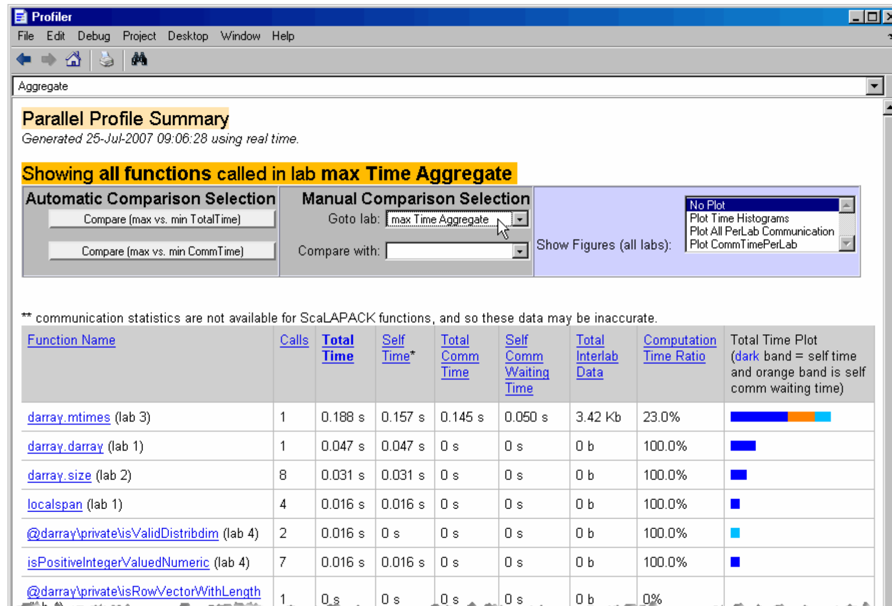
The code that is displayed in the report is taken from the client. If the code has changed on the client since the parallel job ran on the labs, or if the labs are running a different version of the functions, the display might not accurately reflect what actually executed.

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.

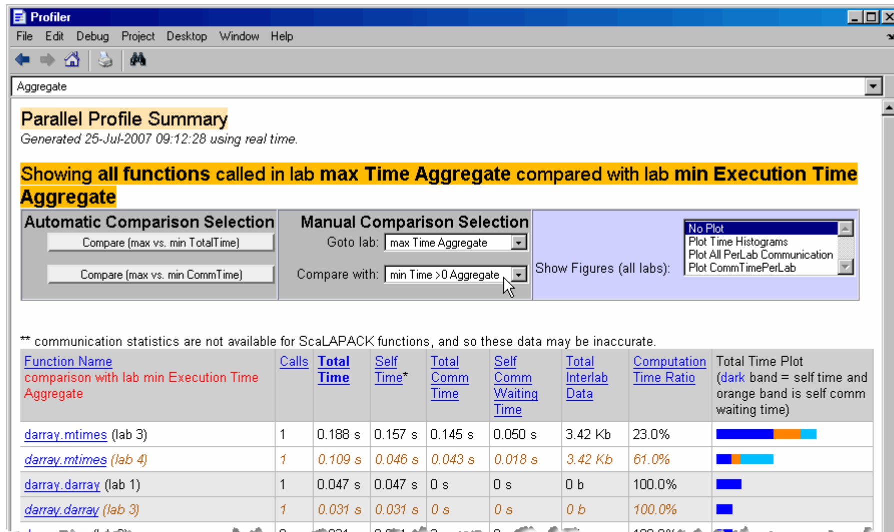
The following summary report shows the result of using the **Automatic Comparison Selection** of **Compare (max vs. min TotalTime)**. The comparison shows data from lab 3 compared to lab 4 because these are the labs that spend the most versus least amount of time executing the code.



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.

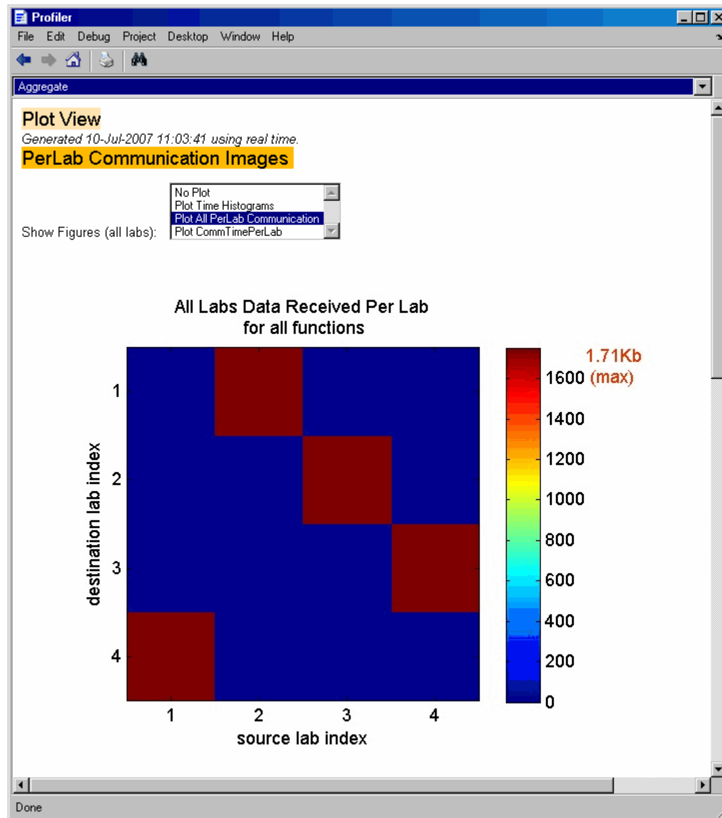


The next figure shows a summary report for the labs that spend the most versus least time for each function. A **Manual Comparison Selection** of **max Time Aggregate** against **min Time >0 Aggregate** generated this summary. Both aggregate settings indicate that the profiler should consider data from all labs for all functions, for both maximum and minimum. This report lists the data for `darray.mtimes` from labs 3 and 4 because they spent the maximum and minimum times on this function. Likewise, data for `darray.darray` is listed from labs 1 and 3.

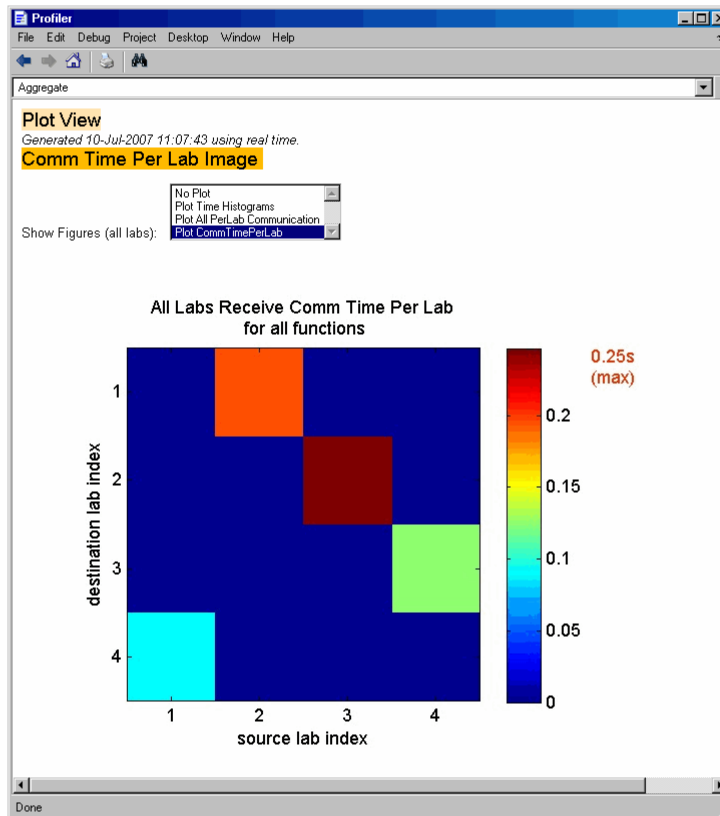


Click on a function name in the summary listing of a comparison to get a detailed comparison. The detailed comparison for `darray.mtimes` looks like this, displaying line-by-line data from both labs:

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 distributed arrays.

Troubleshooting and Debugging

In this section...

"Object Data Size Limitations" on page 2-29

"File Access and Permissions" on page 2-31

"No Results or Failed Job" on page 2-33

"Connection Problems Between the Client and Job Manager" on page 2-34

Object Data Size Limitations

By default, the size limit of data transfers among the distributed computing objects is approximately 50 MB, determined by the Java Virtual Machine (JVM) memory allocation limit. You can increase the amount of JVM memory available to the distributed computing processes (clients, job manager, and workers).

MATLAB Clients and Workers

You can find the current maximum JVM memory limit by typing the command

```
java.lang.Runtime.getRuntime.maxMemory
ans =
    98172928
```

MATLAB clients and MATLAB workers allow up to approximately half of the JVM memory limit for large data transfers. In the default case, half of the approximately 100-MB limit is about 50 MB.

To increase the limit, create a file named `java.opts` that includes the `-Xmx` option, specifying the amount of memory you want to give the JVM.

For example, to increase the JVM memory allocation limit to 200 MB, use the following syntax in the `java.opts` file:

```
-Xmx200m
```

This increased limit allows approximately 100 MB of data to be transferred with distributed computing objects.

Note To avoid virtual memory thrashing, never set the `-Xmx` option to more than 66% of the physical RAM available.

For MATLAB clients on UNIX or Macintosh systems, place the `java.opts` file in a directory where you intend to start MATLAB, and move to that directory before starting MATLAB.

For MATLAB clients on Windows systems

1. Create the `java.opts` file in a directory where you intend to start MATLAB.
2. Create a shortcut to MATLAB.
3. Right-click the shortcut and select **Properties**.
4. In the Properties dialog box, specify the name of the directory in which you created the `java.opts` file as the MATLAB startup directory.

For computers running MATLAB workers, place the modified `java.opts` file in

```
matlabroot/toolbox/distcomp/bin
```

Job Managers

For job managers, the Java memory allocation limit is set in the `mdce_def` file.

On Windows systems, this file can be found at

```
matlabroot/toolbox/distcomp/bin/mdce_def.sh
```

On UNIX and Macintosh systems, this file can be found at

```
matlabroot\toolbox\distcomp\bin\mdce_def.bat
```

The parameter in this file controlling the Java memory limit is `JOB_MANAGER_MAXIMUM_MEMORY`. You should set this limit to four times the value you need for data transfers in your job. For example, to accommodate data transfers of 100 MB, modify the line for UNIX or Macintosh to read

```
JOB_MANAGER_MAXIMUM_MEMORY="400m"
```

Or for Windows, to read

```
set JOB_MANAGER_MAXIMUM_MEMORY=400m
```

Note Although you can increase the amount of data that you can transfer between objects, it is probably more efficient to have the distributed computing processes directly access large data sets in a shared file system. See "Directly Accessing Files" on page 6-12.

File Access and Permissions

Ensuring That Windows Workers Can Access Files

By default, a worker on a Windows node 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 MDCE under a different user with rights to log on as a service. See the section "Setting the User" in the MATLAB Distributed Computing Engine 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:

- LSF might send e-mails with error messages.
- CCS, LSF, 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

- MATLAB failed to start due to licensing errors, is not on the default path on the worker, 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

Detailed instructions for 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 `dctconfig` to change the hostname that the job manager will use for contacting the client.

3

Parallel for-Loops (parfor)

Getting Started with parfor (p. 3-2)	The basic concept of `parfor`-loops and how to begin programming them
Programming Considerations (p. 3-7)	Requirements, limitations, version compatibility, and other considerations in programming `parfor`-loops
Advanced Topics (p. 3-12)	Detailed information about variable classification and other topics to help with optimization and error handling

Getting Started with parfor

In this section...

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

Introduction

The basic concept of a parallel `for`-loop (`parfor`-loop) in MATLAB 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. If the number of workers is equal to the number of loop iterations, each iteration defines a task for the workers. Because each iteration is independent, there is no guarantee that the tasks are synchronized in any way, nor is there any need for this. If there are more iterations than workers, each task comprises more than one loop iteration.

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: 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 control all of this is by the configuration you use for your cluster. For a description of how to manage and use configurations, see "Programming with User Configurations" on page 2-6.

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

```
matlabpool
```

This command by default starts four MATLAB worker sessions 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. The parentheses in the `parfor` statement are necessary, so be sure to include them:

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

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

Differences Between for-Loops and parfor-Loops

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

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

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

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

Although the elements of `A` come out the same in both of these examples, the value of `d` does not. In the `for`-loop above on the left, the iterations execute in sequence, so afterward `d` has the value it held in the last iteration of the loop. In the `parfor`-loop on the right, the iterations execute in parallel, not in sequence, so it would be impossible to assign `d` a definitive value at the end of the loop. This also applies to the loop variable, `i`. Therefore, `parfor`-loop is 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.

Programming Considerations

In this section...

"MATLAB Path" on page 3-7
"Error Handling" on page 3-7
"Limitations" on page 3-8
"Performance Considerations" on page 3-10
"Compatibility with Earlier Versions of MATLAB" on page 3-11

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 `dctRunOnAll` 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` and `lasterror` are unspecified at the end of the `parfor` if they are used within the loop body.

Limitations

Unambiguous Variable Names

You cannot have names in a parfor-loop that are ambiguous as to whether they refer to a variable or function at the time the code is read. (See "Naming Variables" in the MATLAB documentation.) For example, in the following code, if f is not a function on the path when the code is read, nor clearly defined as a variable in the code, f(5) could refer either to the fifth element of the array f, or to the function f with an argument of 5.

```
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 (i = 1:4)
    eval('X');
end
```

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.

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.

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` nor `persistent` variable declarations.

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 3-26.

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 3-26.

Compatibility with Earlier Versions of MATLAB

In versions of MATLAB prior to 7.5, the name `parfor` designated a more limited style of parallel `for` than what is available in MATLAB 7.5 and later. This style was intended for use with distributed arrays inside a parallel job. In version 7.5, MATLAB supports both the pre-7.5 style and the `parfor` described in this chapter. Because of this change, the `parfor` keyword has a different meaning in MATLAB 7.5 than it did in pre-7.5 versions, and you should be careful to use it appropriately.

To help avoid the possibility of your accidentally using the `parfor` keyword to define the earlier style of a `parfor`-loop, there is an additional difference in syntax between the two commands. The range of a `parfor` expression must be enclosed within parentheses. If you do happen to use `parfor` without the parentheses, MATLAB issues a warning message. To avoid this warning, convert the pre-7.5 style `parfor` to an ordinary `for`-loop that uses `drange` to define the range.

Functionality	Syntax Prior to MATLAB 7.5	Syntax in MATLAB 7.5 and Later

Parallel loop for distributed arrays inside a parallel job	`parfor i = range`
	`  loop body`
	`    .`
	`    .`
	`end`
		`for i = drange(range)`
		`  loop body`
		`    .`
		`    .`
		`end`

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

Note The M-Lint utility catches improper use of the `parfor` keyword in the MATLAB Editor, highlighting the invalid syntax. If you right-click the highlighted text, you have the option to replace it with the currently supported syntax.

Advanced Topics

In this section...

"About Programming Notes" on page 3-12
"Classification of Variables" on page 3-12
"Improving Performance" on page 3-26

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 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 3-12
- "Loop Variable" on page 3-13
- "Sliced Variables" on page 3-14
- "Broadcast Variables" on page 3-17
- "Reduction Variables" on page 3-17
- "Temporary Variables" on page 3-24

Overview

When a name in a `parfor`-loop is recognized as referring to a variable, it is classified into one of the following categories. A `parfor`-loop generates an

error if it contains any variables that cannot be uniquely categorized or if any variables violate their category restrictions.

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

Each of these variable classifications appears in this code fragment:

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


Loop Variable

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

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

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

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

Sliced Variables

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

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

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

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

- Type of First-Level Indexing — The first level of indexing is either parentheses, `()`, or braces, `{}`.
- Fixed Index Listing — Within the first-level parenthesis or braces, the list of indices is the same for all occurrences of a given variable.
- Form of Indexing — Within the list of indices for the variable, exactly one index involves the loop variable.
- Shape of Array — In assigning to a sliced variable, the right-hand side of the assignment is not `[]` or `''` (these operators indicate deletion of elements).

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

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

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

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

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

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

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

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

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

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

Form of Indexing. Within the list of indices for a sliced variable, one of these indices is of the form `i`, `i+k`, `i-k`, `k+i`, or `k-i`, where `i` is the loop variable and

k is a constant or a simple (nonindexed) 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 (i = 1:10)
   a = i;
   z = z + i;
   b(i) = r(i);
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 (i = 1:n)
   if someCondition
      A(i) = 32;
   else
      A(i) = 17;
   end
   loop code that uses A(i)
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.

<table>
<tr><td><code>X = X + expr</code></td><td><code>X = expr + X</code></td></tr>
<tr><td><code>X = X - expr</code></td><td>See Associativity in Reduction Assignments in “Further Considerations with Reduction Variables” on page 3-20</td></tr>
<tr><td><code>X = X .* expr</code></td><td><code>X = expr .* X</code></td></tr>
<tr><td><code>X = X * expr</code></td><td><code>X = expr * X</code></td></tr>
<tr><td><code>X = X & expr</code></td><td><code>X = expr & X</code></td></tr>
<tr><td><code>X = X | expr</code></td><td><code>X = expr | X</code></td></tr>
<tr><td><code>X = [X, expr]</code></td><td><code>X = [expr, X]</code></td></tr>
<tr><td><code>X = [X; expr]</code></td><td><code>X = [expr; X]</code></td></tr>
<tr><td><code>X = {X, expr}</code></td><td><code>X = {expr, X}</code></td></tr>
<tr><td><code>X = {X; expr}</code></td><td><code>X = {expr; X}</code></td></tr>
<tr><td><code>X = min(X, expr)</code></td><td><code>X = min(expr, X)</code></td></tr>
<tr><td><code>X = max(X, expr)</code></td><td><code>X = max(expr, X)</code></td></tr>
<tr><td><code>X = union(X, expr)</code></td><td><code>X = union(expr, X)</code></td></tr>
<tr><td><code>X = intersect(X, expr)</code></td><td><code>X = intersect(expr, X)</code></td></tr>
</table>

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 3-17, the only other (and more general) form of a reduction assignment is

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

Required (static): `f` can be a function or a variable. If it is a variable, it must not be affected by the `parfor` body (in other words, it is a broadcast variable).

If `f` is a variable, then for all practical purposes its value at run time is a function handle. However, this is not strictly required; as long as the right-hand side can be evaluated, the resulting value is stored in `X`.

The `parfor`-loop below on the left will not execute correctly because the statement `f = @times` causes `f` to be classified as a temporary variable and

therefore is cleared at the beginning of each iteration. The `parfor` on the right is correct, because it does not assign to `f` inside the loop:

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

Note that the operators `&&` and `||` are not listed in the table in "Reduction Variables" on page 3-17. 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)
```

All the special cases listed in the table in "Reduction Variables" on page 3-17 have a corresponding function that is (perhaps approximately) associated with it, with the exception of the minus operator (`-`). The assignment `X = X - expr` can conceptually be written as `X = X + (-expr)`, and MATLAB achieves this effect for you. (Technically, the function associated with 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.

Interactive Parallel Mode (pmode)

This chapter describes the interactive parallel mode (pmode) of MATLAB in the following sections.

Introduction (p. 4-2)	Introduces the concept of parallel computing with MATLAB
Getting Started with Interactive Parallel Mode (p. 4-3)	Provides a quick tutorial to begin using the interactive parallel mode of MATLAB
Parallel Command Window (p. 4-11)	Describes the pmode interface
Running pmode on a Cluster (p. 4-17)	Describes how to run pmode as a parallel job on a cluster using a configuration
Plotting in pmode (p. 4-18)	Describes how to plot when working in pmode
Limitations and Unexpected Results (p. 4-20)	Provides information on common problems with pmode
Troubleshooting (p. 4-22)	Suggestions for solving problems you might encounter in pmode

Introduction

The interactive parallel mode (pmode) of MATLAB 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.

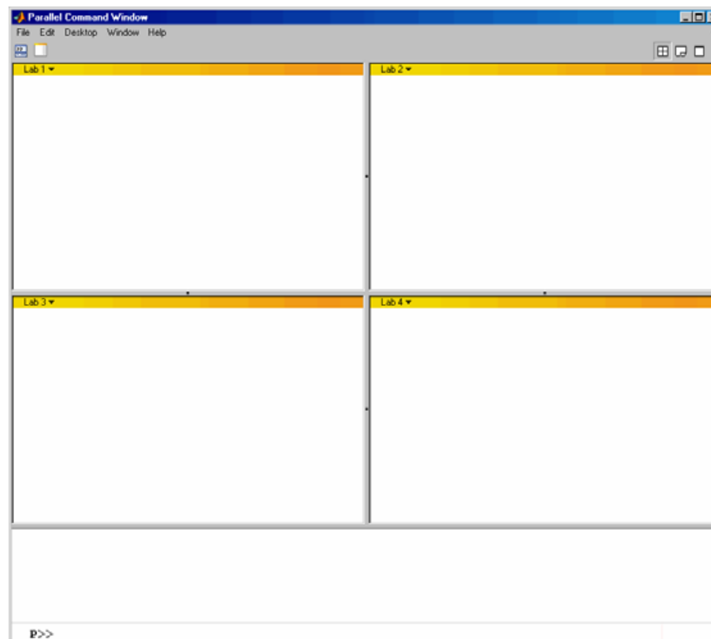
Getting Started with Interactive Parallel Mode

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 parallel mode (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.

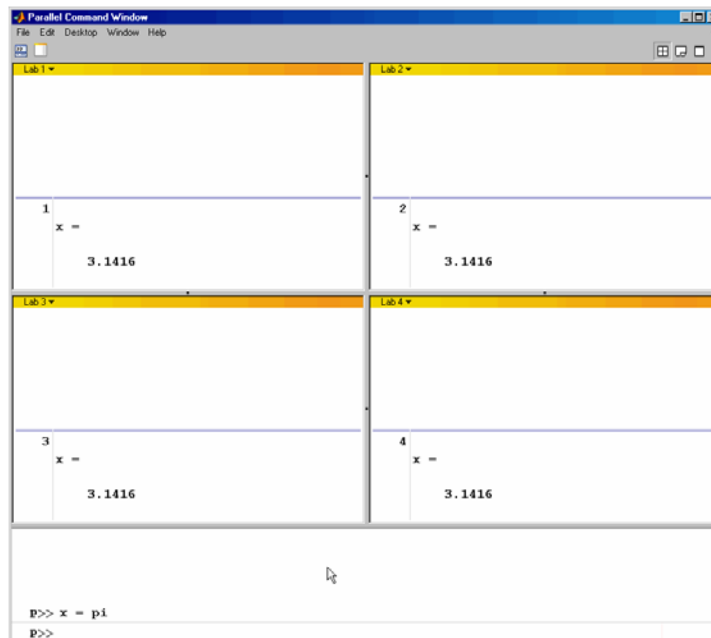
   You can control where the command history appears. For this exercise, the position is set by clicking **Window > History Position > Above Prompt** , but you can set it according to your own preference.

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.

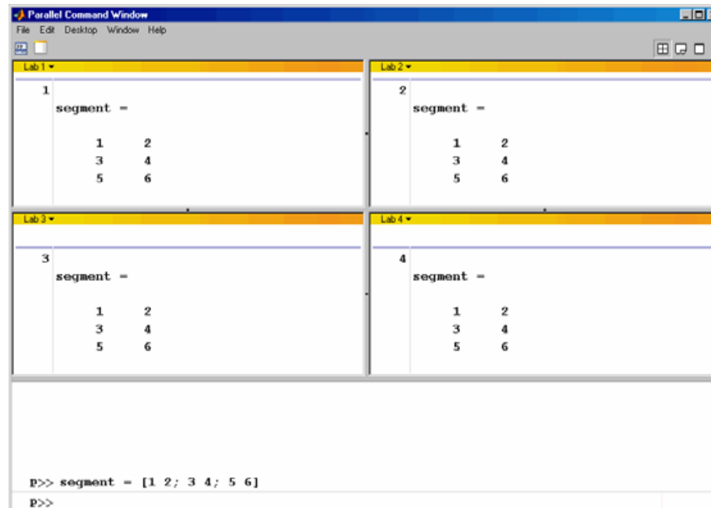
```
P>> x = labindex
```

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. Aggregate the array segments into a coherent array, distributed among the labs, with the `darray` function.

```
P>> whole = darray(segment, 2)
```

This combines four separate 3-by-2 arrays into one 3-by-8 distributed array. The second argument value of `2` indicates that the array is distributed along its second dimension, or columns. On each lab, `segment` provided the data for the local portion of the `whole` array, so `segment` and `local(whole)` appear the same on each lab.

```
Parallel Command Window
File Edit Desktop Window Help

Lab 1
    1
     local(whole) =
         11    12
         13    14
         15    16

Lab 2
    2
     local(whole) =
         21    22
         23    24
         25    26

Lab 3
    3
     local(whole) =
         31    32
         33    34
         35    36

Lab 4
    4
     local(whole) =
         41    42
         43    44
         45    46

P>> segment = [1 2; 3 4; 5 6]
P>> segment = segment + 10*labindex
P>> whole = darray(segment, 2)
P>>
```

9 Now, when you operate on the distributed 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 distributed array allows for operations on its entirety, you can use the `local` function to access the portion of a distributed array on a particular lab.

```
P>> section = local(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
```

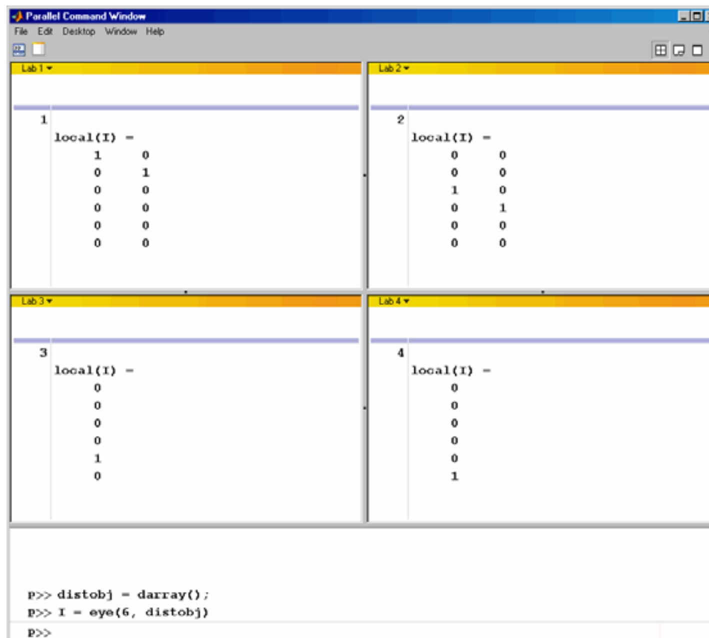
To see the array, type its name.

```
combined
```

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

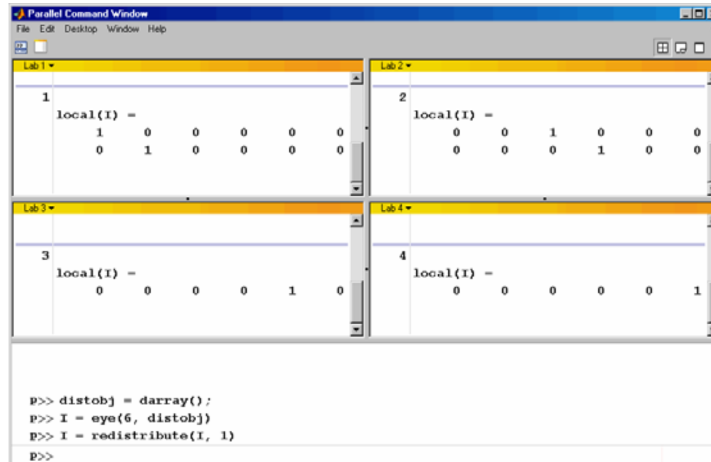
```
P>> distobj = darray();
P>> I = eye(6, distobj)
```

Calling the `darray` function without arguments causes the default distribution, which is by columns 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` indicates to distribute along the first dimension (rows).

```
P>> I = redistribute(I, 1)
```



15 Exit pmode and return to normal MATLAB.

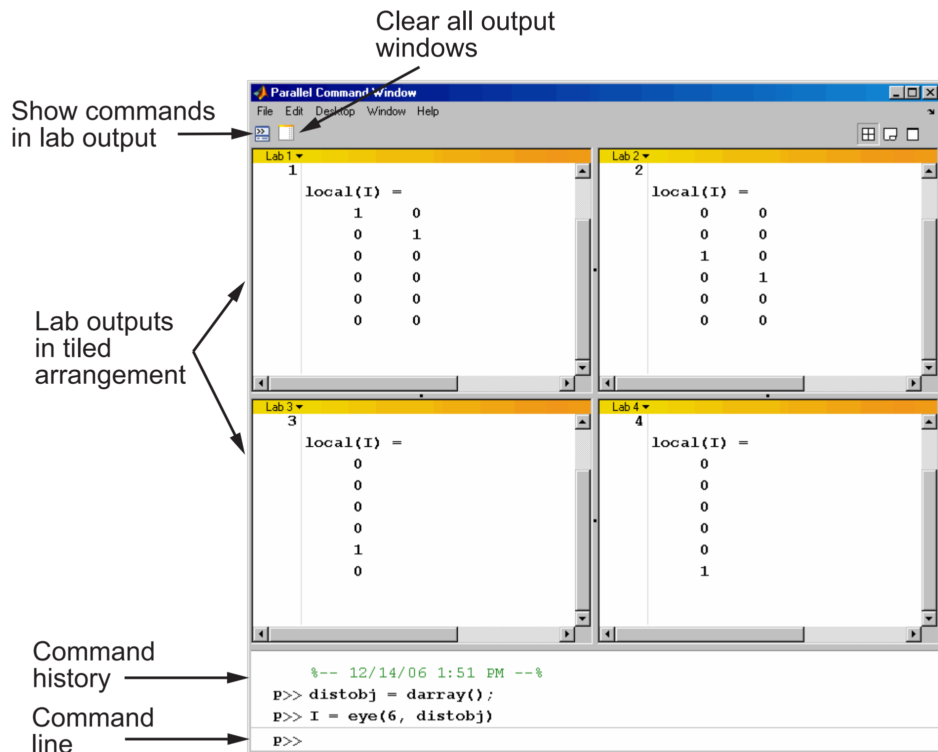
```
P>> pmode exit
```

Parallel Command Window

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

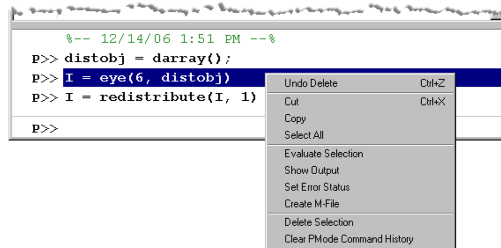
```
pmode start local 4
```

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.

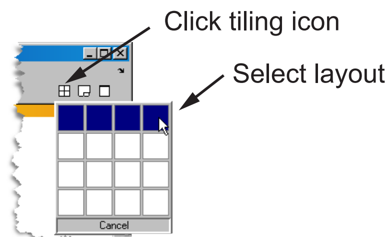


The Parallel Command Window offers much of the same functionality as the MATLAB desktop, including command line, output, and command history.

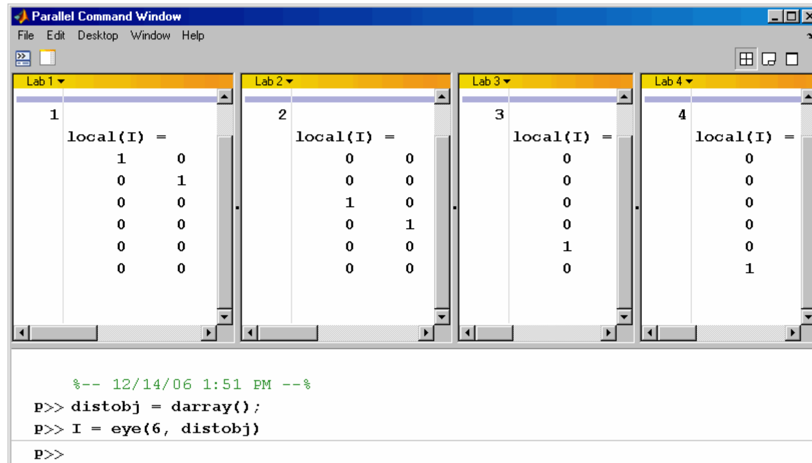
When you select one or more lines in the command history and right-click, you see the following context menu.



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.



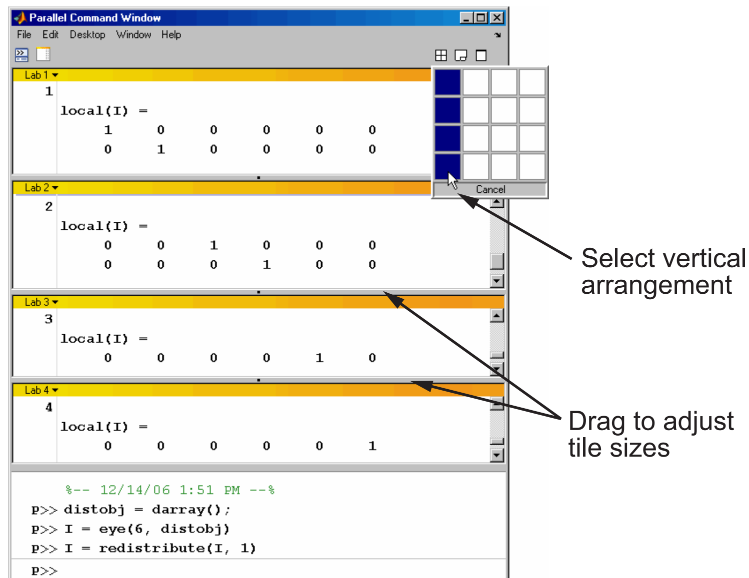
This arrangement results in the following figure, which might be more convenient for viewing data distributed by columns.



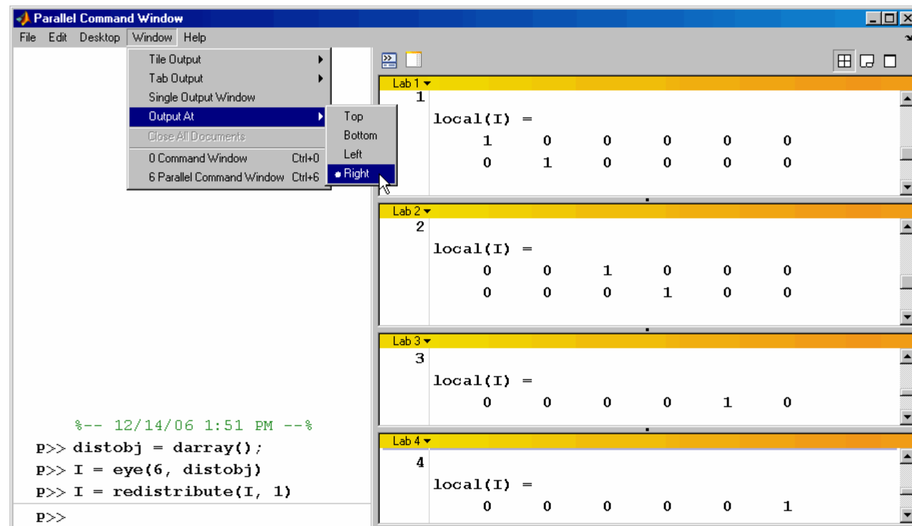
Alternatively, if the data is distributed by rows, you might want to stack the lab tiles vertically. For the following figure, the data is reformatted with the command

```
I = redistribute(I, 1)
```

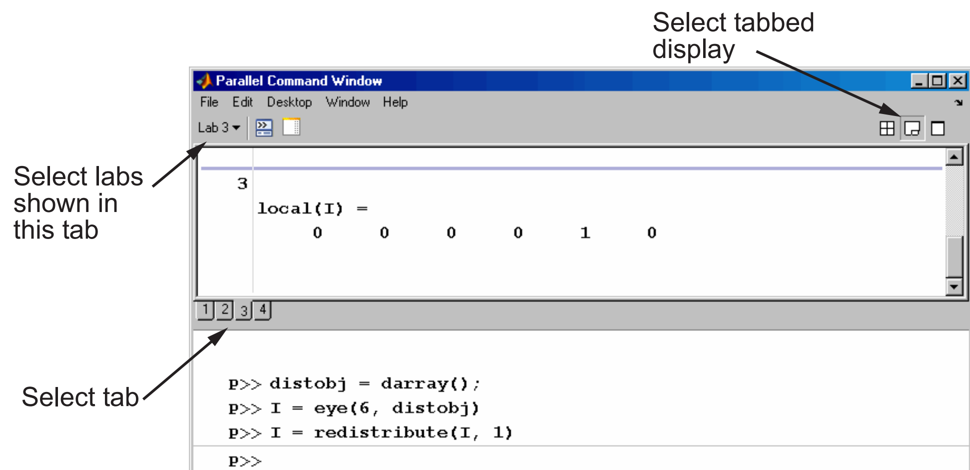
When you rearrange the tiles, you see the following.



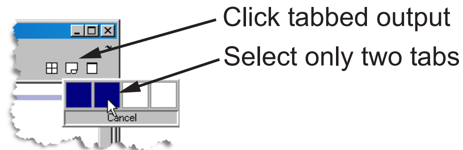
You can control the relative positions of the command window and the lab output. The following figure shows how to set the output to display beside the input, rather than above it.



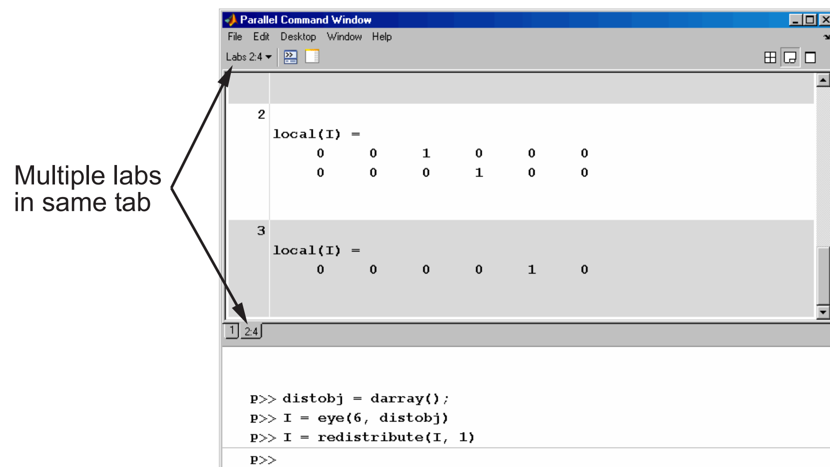
You can choose to view the lab outputs by tabs.



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. 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 2-6.

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 distributed arrays, one way to plot a distributed 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 distributed array of 0s. With four labs, each lab has a 1-by-25 segment of the whole array.

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

  1: local(D) is 1-by-25
  2: local(D) is 1-by-25
  3: local(D) is 1-by-25
  4: local(D) is 1-by-25
```

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 distributed 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

In this section...
"Distributing Nonreplicated Arrays" on page 4-20
"Using Graphics in pmode" on page 4-21

Distributing Nonreplicated Arrays

The `distribute` function is intended for use only on "Replicated Arrays" on page 8-2. When executing the `distribute` function, each lab creates a local segment of the distributed array based on a portion of the array in its workspace. The following simple example illustrates the result of using `distribute` on "Variant Arrays" on page 8-3.

First, create a variant array, whose value depends on `labindex`.

```
P>> x = labindex + (0:1)
  1: x =
  1:      1     2
  2: x =
  2:      2     3
```

Notice that the content of `x` differs on the two labs. When you distribute this 1-by-2 array, each lab gets only one element. With the `distribute` function, lab 1 takes for its local portion of the array the first element of the array in its own workspace; while lab 2 takes the second element of the array in its own workspace.

```
P>> distribute(x)
  1: local(ans) =
  1:      1
  2: local(ans) =
  2:      3
```

The result is the distributed array `[1 3]`. This is neither of the original arrays.

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

Because the labs running a pmode job do not have displays, you cannot use Simulink 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...

"Hostname Resolution" on page 4-22
"Socket Connections" on page 4-22

Hostname Resolution

If a lab cannot resolve the hostname of the computer running the MATLAB client, use `dctconfig` 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 `dctconfig` 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 `dctconfig` 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.

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. Distributed Computing Toolbox 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 (p. 5-2)	Evaluating a function in the cluster while the MATLAB client is blocked
Evaluating Functions Asynchronously (p. 5-8)	Evaluating a function in the cluster in the background, while the MATLAB client continues

Evaluating Functions Synchronously

In this section...

"Scope of dfeval" on page 5-2

"Arguments of dfeval" on page 5-3

"Example — Using dfeval" on page 5-4

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 MATLAB is blocked until the evaluation is complete and the results are assigned to the designated variable. So you provide the necessary information, while Distributed Computing Toolbox 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

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 Distributed Computing Toolbox. 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 2-6, 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.

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 (p. 6-2)	Programming a distributed job using a local scheduler and workers on the client machine
Using a Job Manager (p. 6-7)	Programming a distributed job using the job manager as a scheduler
Using a Fully Supported Third-Party Scheduler (p. 6-18)	Programming a distributed job using a Windows CCS or Platform LSF scheduler to distribute the tasks
Using the Generic Scheduler Interface (p. 6-30)	Programming a distributed job using a generic third-party scheduler to distribute the tasks

Using a Local Scheduler

In this section...

"Creating and Running Jobs with a Local Scheduler" on page 6-2

"Local Scheduler Behavior" on page 6-6

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 Windows machine 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 Distributed Computing Toolbox using a local scheduler:

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

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)
                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: []
       Configuration: ''
```

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 =
    distcomp.task: 5-by-1
```

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 four 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
```

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 by the local scheduler, the scheduler starts a MATLAB worker for each task in the job, but only up to four at one time. If your job has more than four tasks, the scheduler waits for one of the current four tasks to complete before starting another MATLAB worker to evaluate the next task. 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 four 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 6-7
"Sharing Code" on page 6-12
"Managing Objects in the Job Manager" on page 6-14

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 Distributed Computing Toolbox using a MathWorks job manager:

- "Find a Job Manager" on page 6-7
- "Create a Job" on page 6-9
- "Create Tasks" on page 6-10
- "Submit a Job to the Job Queue" on page 6-11
- "Retrieve the Job's Results" on page 6-11

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)
                   Name: 'MyJobManager'
               Hostname: 'bonanza'
            HostAddress: {'123.123.123.123'}
                   Type: 'jobmanager'
          ClusterOsType: 'pc'
                   Jobs: [0x1 double]
                  State: 'running'
          Configuration: ''
               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)
                         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)
                   Name: 'MyJobManager'
```

```
             Hostname: 'bonanza'
          HostAddress: {'123.123.123.123'}
                 Type: 'jobmanager'
        ClusterOsType: 'pc'
                 Jobs: [1x1 distcomp.job]
                State: 'running'
        Configuration: ''
             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 6-12.

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.

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 6-12
- "Passing Data Between Sessions" on page 6-13
- "Passing M-Code for Startup and Finish" on page 6-14

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 Engine run by setting

the MDCEUSER value in the mdce_def file before starting the services. For Windows 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 Engine 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 M-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.

The default maximum amount of data that can be sent in a single call for setting properties is approximately 50 MB. This limit applies to the OutputArguments property as well as to data passed into a job. If the limit is exceeded, you get an error message. For information on how to increase this limit, see "Object Data Size Limitations" on page 2-29.

Passing M-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 M-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.
- `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 M-code you want the worker to execute at the indicated times.

Alternatively, you can create your own versions of these M-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.

For further details on these M-files, see the `jobStartup`, `taskStartup`, and `taskFinish` reference pages.

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 6-15
- "Recovering Objects" on page 6-15
- "Resetting Callback Properties" on page 6-16
- "Permanently Removing Objects" on page 6-16

What Happens When the Client Session Ends

When you close the client session of Distributed Computing Toolbox, all of the objects in the workspace are cleared. However, the objects in MATLAB Distributed Computing Engine 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 Distributed Computing Toolbox can access any of the objects in MATLAB Distributed Computing Engine, 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 6-17
- “Starting a Job Manager from a Clean State” on page 6-17

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 Engine System Administrator’s Guide.

Using a Fully Supported Third-Party Scheduler

In this section...

"Creating and Running Jobs with an LSF or CCS Scheduler" on page 6-18

"Sharing Code" on page 6-25

"Managing Objects" on page 6-27

Creating and Running Jobs with an LSF or CCS Scheduler

If your network already uses a Load Sharing Facility (LSF) or Windows Compute Cluster Server (CCS), you can use Distributed Computing Toolbox 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 Distributed Computing Toolbox for jobs distributed to workers by a fully supported third-party scheduler.

This section assumes you have LSF or CCS installed and running on your network. For more information about LSF, see `http://www.platform.com/Products/`. For more information about CCS, see `http://www.microsoft.com/windowsserver2003/ccs/default.mspx`.

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

- "Find an LSF Scheduler" on page 6-19
- "Find a CCS Scheduler" on page 6-20
- "Create a Job" on page 6-21
- "Create Tasks" on page 6-23
- "Submit a Job to the Job Queue" on page 6-23
- "Retrieve the Job's Results" on page 6-24

Find an LSF Scheduler

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

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

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

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', '\\apps\data\project_55')
set(sched, 'HasSharedFilesystem', true)
set(sched, 'ClusterMatlabRoot', '\\apps\matlab\')
```

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)
                    Type: 'lsf'
            DataLocation: '\\apps\data\project_55'
     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]
                  Configuration: ''
```

Find a CCS Scheduler

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

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

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

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
- The operating system of the cluster
- The name of the scheduler host

```
set(sched, 'DataLocation', '\\apps\data\project_106')
set(sched, 'HasSharedFilesystem', true)
set(sched, 'ClusterMatlabRoot', '\\apps\matlab\')
set(shced, 'ClusterOsType', 'pc')
set(sched, 'SchedulerHostname', 'server04')
```

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.

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)
                     Type: 'ccs'
             DataLocation: '\\apps\data\project_106'
      HasSharedFilesystem: 1
                     Jobs: [0x1 double]
         ClusterMatlabRoot: '\\apps\matlab\'
            ClusterOsType: 'pc'
                 UserData: []
              ClusterSize: Inf
        SchedulerHostname: 'server04'
            Configuration: ''
```

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)
                 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: []
        Configuration: ''
```

This output varies only slightly between LSF and CCS jobs, but is quite different from a job that uses a job manager. For example, jobs on LSF or CCS 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 6-25.

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.

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 LSF 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 6-25
- "Passing Data Between Sessions" on page 6-26
- "Passing M-Code for Startup and Finish" on page 6-26

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 M-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 M-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 M-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.
- `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 M-code you want the worker to execute at the indicated times.

Alternatively, you can create your own versions of these M-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.

For further details on these M-files, see the `jobStartup`, `taskStartup`, and `taskFinish` reference pages.

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 6-28
- "Recovering Objects" on page 6-28

- "Destroying Jobs" on page 6-29

What Happens When the Client Session Ends?

When you close the client session of Distributed Computing Toolbox, 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 Distributed Computing Toolbox 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. These objects refer to jobs listed in the scheduler, whose data is found in the specified `DataLocation`.

```
sched = findResource('scheduler', 'type', 'LSF');
set(sched, 'DataLocation', '/apps/data/project_88');
```

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', '/apps/data/project_88');
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...

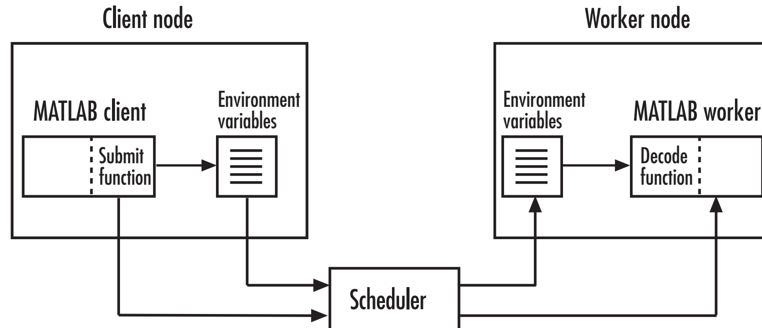
"Overview" on page 6-30
"MATLAB Client Submit Function" on page 6-31
"Example — Writing the Submit Function" on page 6-35
"MATLAB Worker Decode Function" on page 6-36
"Example — Writing the Decode Function" on page 6-38
"Example — Programming and Running a Job in the Client" on page 6-39
"Supplied Submit and Decode Functions" on page 6-44
"Summary" on page 6-45

Overview

Distributed Computing Toolbox 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 Distributed 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})
```

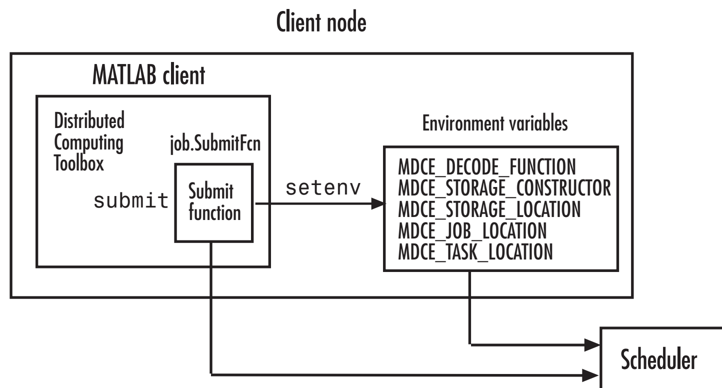
In this example, the submit function requires five arguments: the three defaults, along with the numeric value of `time_limit` and the string value of `testlocation`. The function's declaration line might look like this:

```
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 6-36.

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

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 6-31.

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 properties of your distributed computing 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.

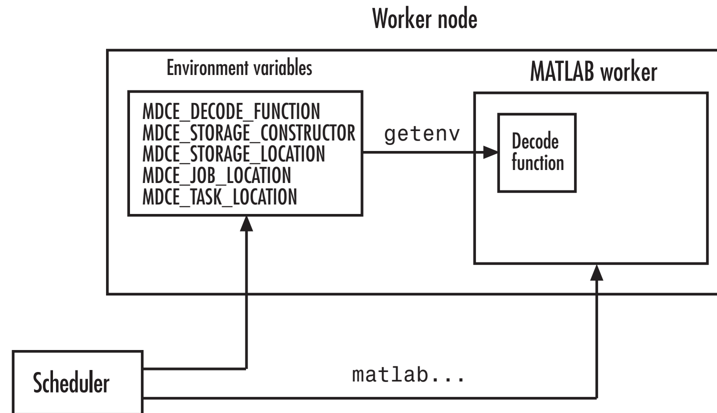
Note If you are not familiar with your network scheduler, ask your system administrator for help.

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 6-31. 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', '\\apps\data\project_101')
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)
                   Type: 'generic'
           DataLocation: '\\apps\data\project_101'
    HasSharedFilesystem: 1
                   Jobs: [0x1 double]
      ClusterMatlabRoot: '\\apps\matlab\'
          ClusterOsType: 'pc'
               UserData: []
            ClusterSize: Inf
     MatlabCommandToRun: 'worker'
              SubmitFcn: []
      ParallelSubmitFcn: []
          Configuration: ''
```

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)
                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)
                     Type: 'generic'
             DataLocation: '\\apps\data\project_101'
      HasSharedFilesystem: 1
                     Jobs: [1x1 distcomp.simplejob]
        ClusterMatlabRoot: '\\apps\matlab\'
            ClusterOsType: 'pc'
                 UserData: []
              ClusterSize: Inf
       MatlabCommandToRun: 'worker'
                SubmitFcn: @mysubmitfunc
        ParallelSubmitFcn: []
            Configuration: ''
```

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 (pbs), LSF (lsf), generic UNIX (ssh), Sun Grid Engine (sge), and mpiexec on Windows

(`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.

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.

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 (p. 7-2)	Explains the difference between distributed and parallel jobs
Using a Supported Scheduler (p. 7-4)	Explains how to program a parallel job using a job manager, local scheduler, or other supported scheduler
Using the Generic Scheduler Interface (p. 7-7)	Explains how to program a parallel job using the generic scheduler interface to work with any scheduler
Further Notes on Parallel Jobs (p. 7-10)	Provides useful information for programming parallel jobs

Introduction

A parallel job consists of only a single task that runs simultaneously on several workers. 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.
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...

"Coding the Task Function" on page 7-4
"Coding in the Client" on page 7-5

Coding the Task Function

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, CCS, LSF, or mpiexec).

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 common file, 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)
set(sched, '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 2-6. 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 7-4) 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.

```
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. This example runs on four labs (the maximum available with a local scheduler), which can be established in the configuration, or can be set 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 7-7

"Coding in the Client" on page 7-7

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 6-30 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. 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 2-6.

   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 7-8.

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 (pbs), LSF (lsf), generic UNIX (ssh), Sun Grid Engine (sge), and mpiexec on Windows (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 7-10

"Avoiding Deadlock and Other Dependency Errors" on page 7-10

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 distributed 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.

8

Parallel Math

This chapter describes the distribution of data across several labs, and the functionality provided for operations on that data in parallel jobs and the interactive parallel mode of MATLAB. The sections are as follows.

Array Types (p. 8-2)	Describes the various types of arrays used in parallel jobs, including pmode
Working with Distributed Arrays (p. 8-5)	Describes how to use distributed arrays for calculation
Using a for-Loop Over a Distributed Range (for-drange) (p. 8-17)	Describes how to program a parallel `for`-loop with distributed arrays
Using MATLAB Functions on Distributed Arrays (p. 8-20)	MATLAB functions that operate on distributed arrays

Array Types

In this section...
"Introduction" on page 8-2
"Nondistributed Arrays" on page 8-2
"Distributed Arrays" on page 8-4

Introduction

All built-in data types and data structures supported by MATLAB 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 (p. 8-2)

Variant Arrays (p. 8-3)

Private Arrays (p. 8-4)

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 at the pmode prompt the value assigned to this variable, all labs respond by showing the same array.

```
P>> A = magic(3)

  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 it to the same variable on all labs. If you display at the pmode prompt the value assigned to this variable, all labs respond by showing their version of the array.

```
P>> A = magic(3) + labindex-1

  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.

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

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:

```
P>> if labindex >= 3; A = magic(3) + labindex - 1; 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
```

Distributed Arrays

With replicated and variant arrays, the full content of the array is stored in the workspace of each lab. *Distributed 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.

```
P>> A = [11:20; 21:30; 31:40];
P>> D = distribute(A, 2)


      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 distributed arrays, see "Working with Distributed Arrays" on page 8-5.

Working with Distributed Arrays

In this section...

"How MATLAB Distributes Arrays" on page 8-5

"Creating a Distributed Array" on page 8-7

"Local Arrays" on page 8-10

"Obtaining Information About the Array" on page 8-11

"Changing the Dimension of Distribution" on page 8-13

"Restoring the Full Array" on page 8-14

"Indexing into a Distributed Array" on page 8-15

How MATLAB Distributes Arrays

When you distribute an array to a number of labs, MATLAB 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 columns.

```
A = zeros(80, 1000);
D = distribute(A)
  1: local(D) is 80-by-250
  2: local(D) is 80-by-250
  3: local(D) is 80-by-250
  4: local(D) is 80-by-250
```

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 Distributed Array

MATLAB displays the local segments of a distributed array as follows for lab 1 and lab 2. Each lab displays that part of the array that is stored in its workspace. This part of the array is said to be *local* to that lab. The lab index appears at the left.

```
1: local(D) =
1:     11    12
1:     21    22
1:     31    32
1:     41    42
2: local(D) =
2:     13    14
2:     23    24
2:     33    34
2:     43    44
```

When displaying larger distributed arrays, MATLAB prints out only the sizes of the local segments.

```
1: local(D) is 4-by-250
2: local(D) is 4-by-250
3: local(D) is 4-by-250
4: local(D) is 4-by-250
```

Note When displayed, a distributed array can look the same as a smaller variant array. For example, on a configuration with four labs, a 4-by-20 distributed array might appear to be the same size as a 4-by-5 variant array because both are displayed as 4-by-5 in each lab window. You can tell the difference either by finding the size of the array or by using the `isdarray` 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 functions called `distribdim` and `partition` that you can use to determine the exact distribution of an array. See "Indexing Functions" on page 8-15 for more information on `dcolon`.

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 Distributed Array

You can create a distributed array in any of the following ways:

- "Partitioning a Larger Array" on page 8-8 — 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 8-9 — Start with smaller variant or replicated arrays stored on each lab, and combine them so that each array becomes a segment of a larger distributed array. This method saves on memory as it lets you build a distributed array from smaller pieces.

- "Using MATLAB Constructor Functions" on page 8-10 — Use any of the MATLAB constructor functions like `rand` or `zeros` with the `darray()` argument. These functions offer a quick means of constructing a distributed array of any size in just one step.

Partitioning a Larger Array

If you have a large array already in memory that you want MATLAB to process more quickly, you can partition it into smaller segments and distribute these segments to all of the labs using the `distribute` 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 variable `A`:

```
P>> A = [11:18; 21:28; 31:38; 41:48]
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 `distribute` function to construct a single 4-by-8 matrix `D` that is distributed along the second dimension of the array:

```
P>> D = distribute(A, 2)
1: local(D) =  | 2: local(D) =  | 3: local(D) =  | 4: local(D) =
    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
```

Note that 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.

```
P>> whos
  Name      Size           Bytes  Class

  A         4x8              256  double array
  D         4x8              460  darray object
```

See the `distribute` function reference page for syntax and usage information.

Building from Smaller Arrays

The `distribute` function is less useful when you are using distribution to reduce the amount of memory required to store data. This is because you have to first construct the full array and then partition it into distributed segments. To save on memory, you can construct the smaller pieces on each lab first, and then combine them into a single array that is distributed across the labs using the `darray` function.

This example creates a 4-by-250 variant array A on each of four labs and then uses `darray` to distribute these segments across four labs, creating a 4-by-1000 distributed array. Here is the variant array, `A`:

```
P>> A = [1:250; 251:500; 501:750; 751:1000] + 250 * (labindex - 1);

       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 across the first (or vertical) dimension. The array is now 16-by-250, with a 4-by-250 segment residing on each lab:

```
P>> D = darray(A, 1)
1: local(D) is 4-by-250
2: local(D) is 4-by-250
3: local(D) is 4-by-250
4: local(D) is 4-by-250

P>> whos
  Name          Size           Bytes  Class

  A              4x250          8000  double array
  D             16x250          8396  distributedarray object
```

You could also use replicated arrays in the same fashion, if you wanted to create a distributed array whose segments were all identical to start with. See the `darray` function reference page for syntax and usage information.

Using MATLAB Constructor Functions

MATLAB provides several array constructor functions that you can use to build distributed 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 darray, `dist`.

Constructor Functions. The distributed constructor functions are listed here. Use the `dist` argument (created by the `darray` function: `dist=darray()`) to specify over which dimension to distribute the array. See the individual reference pages for these functions for further syntax and usage information.

```
cell(m, n, ..., dist)
eye(m, ..., classname, dist)
false(m, n, ..., dist)
Inf(m, n, ..., classname, dist)
NaN(m, n, ..., classname, dist)
ones(m, n, ..., classname, dist)
rand(m, n, ..., dist)
randn(m, n, ..., dist)
sparse(m, n, dist)
speye(m, ..., dist)
sprand(m, n, density, dist)
sprandn(m, n, density, dist)
true(m, n, ..., dist)
zeros(m, n, ..., classname, dist)
```

Local Arrays

That part of a distributed 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 distributed array segment is called a *local array*.

Creating Local Arrays from a Distributed Array

The `local` function copies the segments of a distributed array to a separate variant array. This example makes a local copy `L` of each segment of distributed 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:

```
P>> A = [1:80; 81:160; 161:240];
P>> D = distribute(A, 2);

P>> size(D)
ans =
     3    80

P>> L = local(D);
P>> size(L)
ans =
     3    20
```

Each lab recognizes that the distributed array `D` is 3-by-80. However, notice that the size of the local portion, `L`, is 3-by-20 on each lab, because the 80 columns of `D` are distributed over four labs.

Creating a Distributed from Local Arrays

Use the `darray` function to perform the reverse operation. This function, described in "Building from Smaller Arrays" on page 8-9, 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 of a new distributed array `X`.

```
P>> X = darray(L, 2);
P>> size(X)
ans =
     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 distributed arrays.

Determining Whether an Array Is Distributed

The `isdarray` function returns a logical `1` (`true`) if the input array is distributed, and logical `0` (`false`) otherwise. The syntax is

```
P>> TF = isdarray(D)
```

where `D` is any MATLAB array.

Determining the Dimension of Distribution

The `distribdim` function returns a number that represents the dimension of distribution of a distributed array, and the `partition` function returns a vector that describes how the array is partitioned along its dimension of distribution.

The syntax is

```
P>> distribdim(D)
P>> partition(D)
```

where `D` is any distributed array. For a 250-by-10 matrix distributed across four labs by columns,

```
P>> D = ones(250, 10, darray())
  1: local(D) is 250-by-3
  2: local(D) is 250-by-3
  3: local(D) is 250-by-2
  4: local(D) is 250-by-2
P>> dim = distribdim(D)
  1: dim = 2
P>> part = partition(D)
  1: part = [3 3 2 2]
```

The `distribdim(D)` return value of `2` means the array is distributed by columns (dimension 2); and the `partition(D)` return value of `[3 3 2 2]` means that the first three columns reside in the lab 1, the next three columns in lab 2, the next two columns in lab 3, and the final two columns in lab 4.

Other Array Functions

Other functions that provide information about standard arrays also work on distributed arrays and use the same syntax.

- `ndims` — Returns the number of dimensions.
- `size` — Returns the size of each dimension.
- `length` — Returns the length of a specific dimension.
- `isa` — Returns information about a number of array characteristics.
- `is*` — All functions that have names beginning with `'is'`, such as `ischar` and `issparse`.

numel Not Supported on Distributed Arrays. For a distributed array, the `numel` function does not return the number of elements, but instead always returns a value of `1`.

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.

Construct an 8-by-16 distributed array `D` of random values having distributed columns:

```
P>> D = rand(8, 16, darray());

P>> size(local(D))
ans =
     8     4
```

Create a new array from this one that has distributed rows:

```
P>> X = redistribute(D, 1);

P>> size(local(X))
ans =
     2    16
```

Restoring the Full Array

You can restore a distributed 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:

```
P>> A = [11:20; 21:30; 31:40; 41:50]
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

P>> D = distribute(A, 2)
        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
                   |                |            |
P>> size(local(D))
  1: ans =
  1:      4     3
  2: ans =
  2:      4     3
  3: ans =
  3:      4     2
  4: ans =
  4:      4     2
```

Restore the undistributed segments to the full array form by gathering the segments:

```
P>> X = gather(D)
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

P>> size(X)
ans =
     4     8
```

Indexing into a Distributed Array

While indexing into a nondistributed array is fairly straightforward, distributed 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 distributed 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 varies with each array. For a 200-by-1000 array that has been distributed by columns over four labs, this starting index would be 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 revealed 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 distributed arrays, MATLAB provides a distributed version of the `colon` operator, called `dcolon`. This actually is a function, not a symbolic operator like `colon`.

Indexing Functions

dcolon. The `dcolon` function returns a distributed vector of length `L` that maps the subscripts of an equivalent array residing on the same lab configuration. An equivalent array is an array for which the distributed dimension is also of length `L`. For example, the subscripts of a 50-element `dcolon` vector are as follows:

```
 [1:13]  for Lab 1
[14:26]  for Lab 2
```

```
[27:38]  for Lab 3
[39:50]  for Lab 4
```

This vector shows how MATLAB would distribute 50 rows, columns, or any dimension of an array in a configuration having the same number of labs (four in this case). A 50-row, 10-column array, for example, with the rows distributed over four labs

```
D = rand(50, 10, darray('1d',1))
```

will have rows 1 through 13 stored on lab 1, rows 14 through 26 on lab 2, rows 27 through 38 on lab 3, and rows 39 through 50 on lab 4.

The command syntax for `dcolon` is as follows. The `step` input argument is optional:

```
P>> V = dcolon(first, step, last)
```

Inputs to `dcolon` are shown below. Each input must be a real scalar integer value.

Input Argument	Description
`first`	Number of the first subscript in this dimension.
`step`	Size of the interval between numbers in the generated sequence. Optional; the default is 1.
`last`	Number of the last subscript in this dimension.

To use `dcolon` to index into the 50-by-10 distributed array in the previous example, first generate the vector `V` that shows how the 50-row dimension is partitioned. Then you can use the elements of this vector to derive the range of rows that apply to particular segments of the array.

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

In this section...

"Parallelizing a for-Loop" on page 8-17

"Distributed Arrays in a for-drange Loop" on page 8-18

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 the interactive parallel mode 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 pmode or in a parallel job:

```
results = zeros(1, numDataSets, darray());
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 distributed 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.

Distributed 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 distributed array. Any calculations in such a loop that require a lab to access portions of a distributed 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 distributed arrays, one an identity matrix, the other set to zeros, distributed across four labs.

```
D = eye(8,8,darray())
```

```
E = zeros(8,8,darray())
```

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 2 of the 8 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.

Using MATLAB Functions on Distributed Arrays

Many functions in MATLAB are enhanced so that they operate on distributed 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 distributed array. To see if any function has different behavior when used with a distributed array, type

```
help darray/function_name
```

For example,

```
help darray/normest
```

The following table lists the enhanced MATLAB functions that operate on distributed 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, 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

Objects — By Category

Scheduler Objects (p. 9-2)	Representing job manager, local scheduler, or third-party scheduler
Job Objects (p. 9-2)	Representing different types of jobs
Task Objects (p. 9-3)	Representing different types of tasks
Worker Objects (p. 9-3)	Representing MATLAB worker sessions

Scheduler Objects

ccsscheduler	Access Windows Compute Cluster 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

Job Objects

job	Define 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
simpleparalleljob	Define parallel job behavior and properties when using local or third-party scheduler

Task Objects

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

Worker Objects

worker	Access information about MATLAB worker session

10

Objects — Alphabetical List

ccsscheduler

Purpose Access Windows Compute Cluster Server scheduler

Constructor `findResource`

Container Hierarchy

Parent	None
Children	`simplejob` and `simpleparalleljob` objects

Description A ccsscheduler object provides access to your network's Windows Compute Cluster Server 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
`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
`SchedulerHostname`	Name of host running CCS scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`genericscheduler`, `jobmanager`, `lsfscheduler`, `mpiexec`

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 M-code submit function on the client and the M-code decode function on the worker node.

Methods

`createJob`	Create job object in scheduler and client
`createParallelJob`	Create parallel job object
`findJob`	Find job objects stored in 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
`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`

job

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
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`submit`	Queue job in scheduler
`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 M-file function to execute when job is submitted to job manager queue
RestartWorker	Specify whether to restart MATLAB workers before evaluating job tasks
RunningFcn	Specify M-file function 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

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 MDCE product, and its use as a scheduler is optional.

Methods

`createJob`	Create job object in scheduler and client
`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

localscheduler

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
`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
`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 on LSF

Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterName`	Name of 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 LSF master node
`ParallelSubmission-WrapperScript`	Script LSF scheduler runs to start labs
`SubmitArguments`	Specify additional arguments to use when submitting job to LSF or mpiexec scheduler
`Type`	Type of scheduler object
`UserData`	Specify data to associate with object

See Also

`ccsscheduler`, `genericscheduler`, `jobmanager`, `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
`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 LSF 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`

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
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`submit`	Queue job in scheduler
`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 M-file function to execute when job is submitted to job manager queue
RestartWorker	Specify whether to restart MATLAB workers before evaluating job tasks
RunningFcn	Specify M-file function 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`

simplejob

Purpose

Define job behavior and properties when using local or third-party scheduler

Constructor

`createJob`

Container Hierarchy

Parent	`localscheduler`, `ccsscheduler`, `genericscheduler`, `lsfscheduler`, or `mpiexec` 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
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`submit`	Queue job in scheduler
`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`

simpleparalleljob

Purpose

Define parallel job behavior and properties when using local or third-party scheduler

Constructor

`createParallelJob`

Container Hierarchy

Parent	`localscheduler`, `ccsscheduler`, `genericscheduler`, `lsfscheduler`, or `mpiexec` object
Children	`task` 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
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`submit`	Queue job in scheduler
`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`

Purpose

Define task behavior and properties when using third-party scheduler

Constructor

`createTask`

Container Hierarchy

Parent	`simplejob` 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`

Purpose

Define task behavior and properties when using job manager

Constructor

`createTask`

Container Hierarchy

Parent	`job` 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

`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

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
NumberOfOutputArguments	Number of arguments returned by task function
OutputArguments	Data returned from execution of task
Parent	Parent object of job or task
RunningFcn	Specify M-file function 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 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 distributed by a job manager. Only worker sessions started with the `startworker` script can be represented by a worker object.

Methods

None

Properties

`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`

Functions — By Category

General Toolbox Functions (p. 11-2)	Toolbox functions not specific to particular object type
Job Manager Functions (p. 11-3)	Operate on job manager object
Scheduler Functions (p. 11-3)	Operate on various schedulers
Job Functions (p. 11-4)	Operate on job object
Task Functions (p. 11-4)	Operate on task object
Toolbox Functions Used in Parallel Jobs and pmode (p. 11-5)	Execute within parallel job code
Toolbox Functions Used in MATLAB Workers (p. 11-7)	Execute within MATLAB worker session

General Toolbox Functions

`clear`	Remove objects from MATLAB workspace
`dctconfig`	Configure settings for Distributed Computing Toolbox client session
`dctRunOnAll`	Run command on client and all workers in matlabpool
`defaultParallelConfig`	Default parallel computing configuration
`dfeval`	Evaluate function using cluster
`dfevalasync`	Evaluate function asynchronously using cluster
`findResource`	Find available distributed computing resources
`get`	Object properties
`help`	Help for toolbox functions in Command Window
`inspect`	Open Property Inspector
`jobStartup`	M-file for user-defined options to run when job starts
`length`	Length of object array
`matlabpool`	Start parallel language worker pool
`methods`	List functions of object class
`parfor`	Execute block of code in parallel
`pmode`	Interactive parallel mode
`set`	Configure or display object properties
`size`	Size of object array

`taskFinish`	M-file for user-defined options to run when task finishes
`taskStartup`	M-file for user-defined options to run when task starts

Job Manager Functions

`createJob`	Create job object in scheduler and client
`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

Scheduler Functions

`createJob`	Create job object in scheduler and client
`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
`mpiLibConf`	Location of MPI implementation

`mpiSettings`	Configure options for MPI communication
`setupForParallelExecution`	Set options for submitting parallel jobs on LSF

Job Functions

`cancel`	Cancel job or task
`createTask`	Create new task in job
`destroy`	Remove job or task object from parent and memory
`findTask`	Task objects belonging to job object
`getAllOutputArguments`	Output arguments from evaluation of all tasks in job object
`submit`	Queue job in scheduler
`waitForState`	Wait for object to change state

Task Functions

`cancel`	Cancel job or task
`destroy`	Remove job or task object from parent and memory
`waitForState`	Wait for object to change state

Toolbox Functions Used in Parallel Jobs and pmode

`cell`	Create distributed cell array
`darray`	Create distributed array from local data
`dcolon`	Distributed colon operation
`dcolonpartition`	Default partition for distributed array
`distribdim`	Distributed dimension of distributed array
`distribute`	Distribute replicated array
`eye`	Create distributed identity matrix
`false`	Create distributed false array
`for`	For-loop over distributed range
`gather`	Convert distributed array into replicated array
`gcat`	Global concatenation
`gop`	Global operation across all labs
`gplus`	Global addition
`Inf`	Create distributed array of `Inf` values
`isdarray`	True for distributed array
`isreplicated`	True for replicated array
`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
`local`	Local portion of distributed array
`localspan`	Index range of local segment of distributed array
`mpiprofile`	Profile parallel communication and execution times
`NaN`	Create distributed array of NaN values
`numlabs`	Total number of labs operating in parallel on current job
`ones`	Create distributed array of 1s
`partition`	Partition of distributed array
`pload`	Load file into parallel session
`psave`	Save data from parallel job session
`rand`	Create distributed array of uniformly distributed pseudo-random numbers
`randn`	Create distributed array of normally distributed random values
`redistribute`	Distribute array along different dimension
`sparse`	Create distributed sparse matrix
`speye`	Create distributed sparse identity matrix
`sprand`	Create distributed sparse array of uniformly distributed pseudo-random values
`sprandn`	Create distributed sparse array of normally distributed random values

`true`	Create distributed true array
`zeros`	Create distributed array of 0s

Toolbox Functions Used in MATLAB Workers

`getCurrentJob`	Job object whose task is currently being evaluated
`getCurrentJobmanager`	Job manager object that distributed 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

Functions — Alphabetical List

cancel

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

Create distributed cell array

Syntax

```
D = cell(n, dist)
D = cell(m, n, p, ..., dist)
D = cell([m, n, p, ...], dist)
```

Description

`D = cell(n, dist)` creates an n-by-n distributed array of underlying class cell. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `cell`.

`D = cell(m, n, p, ..., dist)` and `D = cell([m, n, p, ...], dist)` create an m-by-n-by-p-by-... distributed array of underlying class cell. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

Examples

With four labs,

```
D = cell(1000,darray())
```

creates a 1000-by-1000 distributed cell array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = cell(10, 10, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed cell array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of D.

See Also

`cell` MATLAB function reference page

`eye`, `false`, `Inf`, `NaN`, `ones`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

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`

createJob

Purpose

Create job object in scheduler and client

Syntax

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

Arguments

`obj`	The job object.
`scheduler`	The job manager object representing the job manager service that will execute the job, or the scheduler object representing the scheduler on the cluster that will distribute the job.
`p1`, *`p2`*	Object properties configured at object creation.
`v1`, `v2`	Initial values for corresponding object properties.

Description

`obj = createJob(scheduler)` creates a job object at the data location for the identified scheduler, or in the job manager.

`obj = createJob(..., 'p1', v1, 'p2', v2, ...)` creates a job object with the specified property values. 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 with the property values specified in the configuration `ConfigurationName`. For details about defining and applying configurations, see "Programming with User Configurations" on page 2-6.

Examples

Construct a job object.

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

Add tasks to the job.

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

Run the job.

```
submit(obj);
```

Retrieve job results.

```
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`

createParallelJob

Purpose Create parallel job object

Syntax

```
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`, using either a job manager or mpiexec scheduler.
`p1`, *`p2`*	Object properties configured at object creation.
`v1`, `v2`	Initial values for corresponding object properties.

Description

`pjob = createParallelJob(scheduler)` creates a parallel job object at the data location for the identified scheduler, or in the job manager. 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(..., 'p1', v1, 'p2', v2, ...)` creates a parallel job object with the specified property values. 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.

`pjob = createParallelJob(..., 'configuration', 'ConfigurationName',...)` creates a parallel 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 2-6.

Examples

Construct a parallel job object in a job manager queue.

```
jm = findResource('scheduler','type','jobmanager');
pjob = createParallelJob(jm,'Name','testparalleljob');
```

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);
```

Retrieve 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`

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}`.

`t = createTask(..., 'p1',v1,'p2',v2,...)` adds a task object with the specified property values. 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 2-6.

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);
```

Get the output from the task evaluation.

```
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

darray

Purpose

Create distributed array from local data

Syntax

```
A = darray()
A = darray('1d')
A = darray('1d', dim)
A = darray('1d', dim, part)
A = darray('2d')
A = darray('2d', lbgrid)
A = darray('2d', lbgrid, blksize)
A = darray(L)
A = darray(L, '1d')
A = darray(L, dim)
A = darray(L, D)
A = darray(L, dim, part)
A = darray(L, '1d', dim, part)
A = darray(L, '2d')
A = darray(L, '2d', lbgrid)
A = darray(L, '2d', lbgrid, blksize)
A = darray(L, '2d', lbgrid, blksize, siz)
```

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 `'2d'`, 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.

`A = darray()`, with no arguments, returns a primitive distributed array with zero-valued or empty parameters, which can then be used as an argument to other darray functions to indicate that the function is to create a distributed array. For example,

```
zeros(..., darray())
randn(..., darray())
```

`A = darray('1d')` is the same as `A = darray()`.

`A = darray('1d', dim)` also forms a primitive distributed array with `distribdim(A) = dim` and `partition(A) = dcolonpartition`.

`A = darray('1d', dim, part)` also forms a primitive distributed array with `distribdim(A) = dim` and `partition(A) = part`.

`A = darray('2d')` forms a primitive `'2d'` distributed array.

`A = darray('2d', lbgrid)` forms a primitive `'2d'` distributed array with `labgrid(A) = lbgrid` and `blocksize(A) = defaultblocksize(numlabs)`.

`A = darray('2d', lbgrid, blksize)` forms a primitive `'2d'` distributed array with `labgrid(A) = lbgrid` and `blocksize(A) = blksize`.

`A = darray(L)` forms a `'1d'` distributed array with `local(A) = L`. The darray `A` is created as if you had concatenated the `L` from all the labs together. The distribution scheme of `A` is derived from the sizes of the `L` arrays.

`A = darray(L, '1d')` is the same as `A = darray(L)`.

`A = darray(L, dim)` forms a `'1d'` distributed array with `distribdim(A) = dim`. `szl = size(L)` must be the same on all labs, except possibly for `szl(dim)`. Communication between labs is required to determine overall size and partition. `dim` might be larger than `gop(@max, ndims(L))`.

`A = darray(L, D)` forms a distributed array with the same distribution scheme as the darray `D`.

`A = darray(L, dim, part)` forms a `'1d'` distributed array with `local(A) = L`, `distribdim(A) = dim` and `partition(A) = part`. `szl = size(L)` must be the same on all labs, except for `szl(dim)` which must equal `part(labindex)`. No communication is required between labs to achieve this.

`A = darray(L, '1d', dim, part)` is the same as `A = darray(L,dim,part)`.

`A = darray(L, '2d')` forms a `'2d'` distributed array with `local(A) = L`, `labgrid(A) = defaultlabgrid(numlabs)`, and `blocksize(A) = defaultblocksize(numlabs)`. Communication is required between the labs to determine `size(A)`.

`A = darray(L, '2d', lbgrid)` forms a `'2d'` distributed array with `local(A) = L`, `labgrid(A) = lbgrid`, `blocksize(A) = defaultblocksize(numlabs)`. Communication is required between the labs to determine `size(A)`.

`A = darray(L, '2d', lbgrid, blksize)` forms a `'2d'` distributed array with `local(A) = L`, `labgrid(A) = lbgrid`, `blocksize(A) = blksize`. Communication is required between the labs to determine `size(A)`.

`A = darray(L, '2d', lbgrid, blksize, siz)` forms a `'2d'` distributed array with `local(A) = L`, `labgrid(A) = lbgrid`, `blocksize(A) = blksize`, and `size(A) = siz`. No communication is required between the labs to achieve this.

Examples

Create a 3-dimensional array with distribution dimension `2` and partition `[1 2 1 2 ...]`.

```
if mod(labindex, 2)
    L = rand(2,1,4)
else
    L = rand(2,2,4)
end
A = darray(L)
```

On 4 labs, create a 20-by-5 array `A` distributed by rows (over its first dimension) with an even partition.

```
L = magic(5) + labindex;
dim = 1;
A = darray(L, dim);
```

The second `dim` input to `darray` is required here to override the default distribution dimension.

See Also `distribdim`, `distribute`, `local`, `partition`, `redistribute`

dcolon

Purpose

Distributed colon operation

Syntax

```
dcolon(a,d,b)
dcolon(a,b)
```

Description

`dcolon` is the basis for parallel for-loops and the default distribution of distributed arrays.

`dcolon(a,d,b)` partitions the vector `a:d:b` into `numlabs` contiguous subvectors of equal, or nearly equal length, and creates a distributed array whose local portion on each lab is the `labindex`-th subvector.

`dcolon(a,b)` uses `d = 1`.

Examples

Partition the vector `1:10` into four subvectors among four labs.

```
P>> C=dcolon(1,10)
  1: 1: local(C) =
  1:      1     2     3
  2: 2: local(C) =
  2:      4     5     6
  3: 3: local(C) =
  3:      7     8
  4: 4: local(C) =
  4:      9    10
```

See Also

`colon` MATLAB function reference page

`darray`, `dcolonpartition`, `localspan`, `for`, `partition`

Purpose

Default partition for distributed array

Syntax

```
P = dcolonpartition(n)
```

Description

`P = dcolonpartition(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 distributed arrays.

Examples

If `numlabs = 4`,

```
P>> P = dcolonpartition(10)
  1: P =
  1:      3     3     2     2
  2: P =
  2:      3     3     2     2
  3: P =
  3:      3     3     2     2
  4: P =
  4:      3     3     2     2
```

See Also

`darray`, `dcolon`, `distribute`, `localspan`, `partition`

dctconfig

Purpose

Configure settings for Distributed Computing Toolbox client session

Syntax

```
dctconfig('p1', v1, ...)
config = dctconfig('p1', v1, ...)
config = dctconfig()
```

Arguments

p1	Property to configure. Supported properties are `'port'`, `'hostname'`, and `'pmodeport'`.
`v1`	Value for corresponding property.
`config`	Structure of configuration value.

Description

`dctconfig('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 `'port'`, the specified value is used to set the port for the client session of Distributed Computing Toolbox. This is useful in environments where the choice of ports is limited. By default, the client session searches for an available port to communicate with the other sessions of MATLAB Distributed Computing Engine. In networks where you are required to use specific ports, you use `dctconfig` to set the client's port.

If the property is `'hostname'`, the specified value is used to set the hostname for the client session of Distributed Computing Toolbox. 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.

If the property is `'pmodeport'`, the specified value is used to set the port for communications with the labs in a pmode session.

`config = dctconfig('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 = dctconfig()`, without any input arguments, returns all the current values as a structure to `config`. If you have not set any values, these are the defaults.

Examples

View the current settings for hostname and ports.

```
config = dctconfig()
config =
        port: 27370
    hostname: 'machine32'
   pmodeport: 27371
```

Set the current client session port number to `21000` with hostname `fdm4`.

```
dctconfig('hostname', 'fdm4', 'port', 21000');
```

Set the client hostname to a fully qualified domain name.

```
dctconfig('hostname', 'desktop24.subnet6.mathworks.com');
```

dctRunOnAll

Purpose

Run command on client and all workers in matlabpool

Syntax

```
dctRunOnAll command
```

Description

`dctRunOnAll 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 `dctRunOnAll` 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 Linux and your client is running Windows, an `addpath` argument with Linux-style paths will warn on the Windows client.

Examples

Clear all loaded functions on all labs:

```
dctRunOnAll clear functions
```

Change the directory on all workers to the project directory:

```
dctRunOnAll cd /opt/projects/c1456
```

Add some directories to the paths of all the labs:

```
dctRunOnAll addpath({'/usr/share/path1' '/usr/share/path2'})
```

See Also

`matlabpool`

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 be `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.

Note that 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.

See Also

`findResource`, `matlabpool`, `pmode`

demote

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.

See Also

`createJob`, `findJob`, `promote`, `submit`

destroy

Purpose

Remove job or task object from parent and memory

Syntax

```
destroy(obj)
```

Arguments

`obj`	Job or task object 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`

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 Distributed Computing Toolbox. `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` will be three cell arrays of 10 elements each.

`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 2-6. 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 5-2.

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 = dfeval(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 = dfeval(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 2-6. Note that 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 5-8.

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`

distribdim

Purpose Distributed dimension of distributed array

Syntax

```
dim = distribdim(dist)
```

Description `dim = distribdim(dist)` returns the distribution dimension of a distributed array. If `dim` is `-1`, the distribution dimension is unspecified.

See Also `darray`, `localspan`, `partition`

distribute

Purpose

Distribute replicated array

Syntax

```
D = distribute(X)
D = distribute(X, dim)
D = distribute(X, dist)
```

Description

`D = distribute(X)` distributes `X` on its last nonsingleton dimension using the default `dcolon`-based distributor. `X` must be a replicated array, namely it must have the same value on all labs. `size(D)` is the same as `size(X)`.

`D = distribute(X, dim)` distributes `X` over dimension `dim`. `dim` must be between `1` and `ndims(X)`.

`D = distribute(X, dist)` for distributor `dist` distributes `X` accordingly. `dist` should specify distribution dimension (`dim`) and partition (`PAR`); but if it does not, `dim` is taken to be the last nonsingleton dimension and `PAR` is taken to be `dcolonpartition(size(X, dim))`.

If `X` is a replicated array, `X = gather(distribute(X))` returns the original replicated array, `X`.

Using `distribute` on a replicated array is not the most memory-efficient way of creating a distributed array. Use the `darray` or the `zeros(m, n, ..., dist)` family of functions instead.

Remarks

`distribute` is intended for use only with replicated arrays that are identical on all labs. A function such as `rand` generates a different (variant) array on each lab.

`gather` essentially performs the inverse of `distribute`.

Examples

```
D = distribute(magic(numlabs));
D = distribute(cat(3, pascal(4), hilb(4), magic(4), eye(4)), 3);
```

See Also

`darray`, `dcolonpartition`, `gather`

Purpose

Create distributed identity matrix

Syntax

```
D = eye(n, dist)
D = eye(m, n, dist)
D = eye([m, n], dist)
D = eye(..., classname, dist)
```

Description

`D = eye(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = eye(m, n, dist)` and `D = eye([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = eye(..., classname, dist)` optionally 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

With four labs,

```
D = eye(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = eye(10, 10, 'uint16', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed 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

cell, false, Inf, NaN, ones, rand, randn, sparse, speye, sprand, sprandn, true, zeros

Purpose

Create distributed false array

Syntax

```
F = false(n, dist)
F = false(m, n, dist)
F = false([m, n], dist)
```

Description

`F = false(n, dist)` creates an n-by-n distributed array of underlying class logical. `F` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `F` is distributed by its second dimension. If `PAR` is unspecified, then `F` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `false`.

`F = false(m, n, dist)` and `F = false([m, n], dist)` create an m-by-n distributed array of underlying class logical. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `F`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

Examples

With four labs,

```
F = false(1000, darray())
```

creates a 1000-by-1000 distributed double array `F`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `F`.

```
F = false(10, 10, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed logical array `F`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `F`.

See Also

`false` MATLAB function reference page

`cell`, `eye`, `Inf`, `NaN`, `ones`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

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`

Purpose

Find available distributed computing resources

Syntax

```
out = findResource('scheduler','type','SchedType')
out = findResource('scheduler','type','jobmanager', ...
                              'LookupURL','host:port')
out = findResource('scheduler','type','SchedType', ...,
                              'p1', v1, 'p2', v2,...)
out = findResource('scheduler', ...
                              'configuration', 'ConfigurationName')
out = findResource('worker')
out = findResource('worker','LookupURL','host:port')
out = findResource('worker', ..., 'p1', v1, 'p2', v2,...)
```

Arguments

`out`	Object or array of objects returned.
`'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'`, `'ccs'`, `'LSF'`, `'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.
`'configuration'`	Literal string to indicate usage of a configuration.
`'ConfigurationName'`	Name of configuration to use.

findResource

Description

`out = findResource('`**`scheduler`**`','`**`type`**`','`*`SchedType`*`') out = findResource('`**`worker`**`')` return an array, `out`, containing objects representing all available distributed computing schedulers of the given type, or workers. *`SchedType`* can be `'jobmanager'`, `'local'`, `'ccs'`, `'LSF'`, `'mpiexec'`, or any string starting with `'generic'`. A `'local'` scheduler will queue 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. For third-party schedulers, job data is stored in the location specified by the scheduler object's `DataLocation` property.

`out = findResource('`**`scheduler`**`','`**`type`**`','`**`jobmanager`**`',`
`...   '`**`LookupURL`**`','host:port')`
`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 only necessary 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.

Note **`LookupURL`** is ignored 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'`.

`out = findResource('`**`scheduler`**`', ... '`**`configuration`**`', 'ConfigurationName')` returns an array, `out`, of schedulers whose property names and property values match those defined by the parameters in the configuration `ConfigurationName`. For details about defining and applying configurations, see "Programming with User Configurations" on page 2-6.

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 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`

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`

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 `dcolon` 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`, `darray`, `distribute`, `gather`, and `redistribute`.

It is possible to access portions of distributed arrays that are local to each lab, but it is not possible to access other portions of distributed 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 distributed array.

```
r = zeros(1, 40, darray());
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, darray());
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`

Purpose

Convert distributed array into replicated array

Syntax

```
X = gather(D)
X = gather(D, lab)
```

Description

`X = gather(D)` is a replicated array formed from the distributed array `D`.

`D = distribute(gather(D))` returns the original distributed array `D`.

`X = gather(D, lab)` converts a distributed array `D` 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 distributed array in the workspaces of all the labs on which it executes, not on the MATLAB client. If you want to transfer a distributed array into the client workspace, first gather it, then move it from a lab to the client with `pmode lab2client`. See the `pmode` reference page for more details.

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`.

As `gather` performs the inverse of `distribute`, be aware that if you use `distribute` on a nonreplicated array, `gather` does not return the original. For example, `gather(distribute(rand(n,m)))` does not return the original random matrix, because `rand` generates a different matrix on each lab in the first place, therefore the original matrix is variant, not replicated.

Examples

Distribute a magic square across your labs, then gather the matrix onto every lab. This code returns `M = magic(n)` on all labs.

```
D = distribute(magic(n))
M = gather(D)
```

Gather all of the data in D onto lab 1, so that it can be saved from there.

```
D = distribute(magic(n));
out = gather(D, 1);
if labindex == 1
    save data.mat out;
end
```

See Also distribute, pmode

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

getAllOutputArguments

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`

getCurrentJob

Purpose

Job object whose task is currently being evaluated

Syntax

```
job = getCurrentJob
```

Arguments

`job`	The job object that contains the task currently being evaluated by the worker session.

Description

`job = getCurrentJob` returns the job object that is the `Parent` of the task currently being evaluated by the worker session.

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`

Purpose

Job manager object that distributed current task

Syntax

```
jm = getCurrentJobmanager
```

Arguments

`jm`	The job manager object that distributed 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 distributed by a third-party scheduler instead of a job manager, `getCurrentJobmanager` returns a `distcomp.taskrunner` object.

See Also

`getCurrentJob`, `getCurrentTask`, `getCurrentWorker`, `getFileDependencyDir`

getCurrentTask

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`

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`

getDebugLog

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, LSF, or CCS 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`.

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

Complete the initialization of the scheduler object by setting all the necessary properties on it.

```
set(mpiexecObj, '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

getFileDependencyDir

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`

Purpose

Global operation across all labs

Syntax

`gop(@F, x)`

Arguments

`F`	Function to operate across labs.
`x`	Argument to function `F`, should be same variable on all labs.

Description

`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))
```

Examples

Calculate the sum of all labs' value for `x`.

```
gop(@plus,x)
```

Find the maximum value of `x` among all the labs.

```
gop(@max,x)
```

Perform the horizontal concatenation of `x` from all labs.

```
gop(@horzcat,x)
```

Calculate the 2-norm of `x` from all labs.

```
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`

help

Purpose Help for toolbox functions in Command Window

Syntax

```
help class/function
```

Arguments

class	A Distributed 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 Distributed 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`

Purpose

Create distributed array of `Inf` values

Syntax

```
D = Inf(n, dist)
D = Inf(m, n, dist)
D = Inf([m, n], dist)
D = Inf(..., classname, dist)
```

Description

`D = Inf(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = Inf(m, n, dist)` and `D = Inf([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = Inf(..., classname, dist)` optionally specifies the class of the distributed array `D`. Valid choices are the same as for the regular `Inf` function: `'double'` (the default), and `'single'`.

Examples

With four labs,

```
D = Inf(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = Inf(10, 10, 'single', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed single array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`Inf` MATLAB function reference page

`cell`, `eye`, `false`, `NaN`, `ones`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

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.

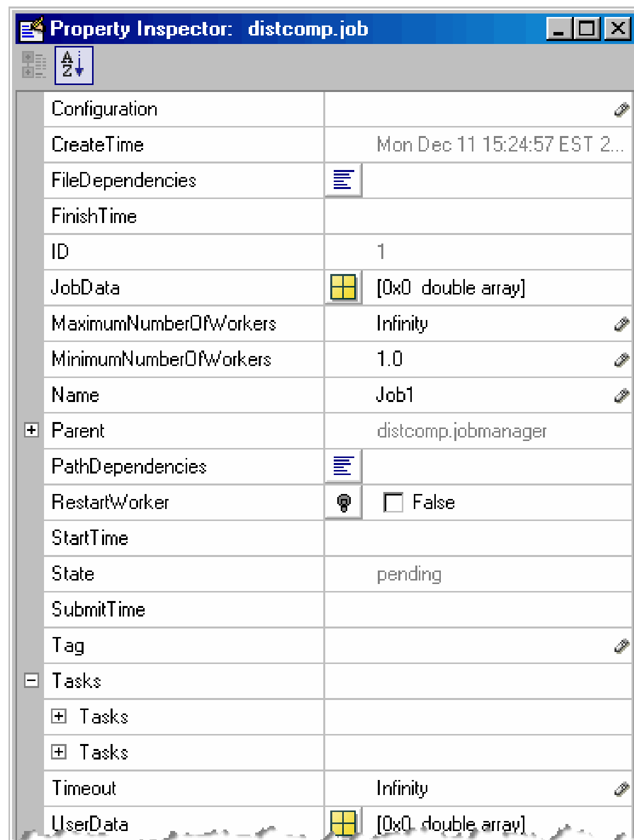
The Property Inspector does not automatically update its display. To refresh the Property Inspector, open it again.

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`

Purpose True for distributed array

Syntax

```
tf = isdarray(X)
```

Description

`tf = isdarray(X)` returns `true` for a distributed array, or `false` otherwise. For a description of a distributed array, see "Array Types" on page 8-2.

Examples

```
L = ones(100, 1)
D = ones(100, 1, darray())
isdarray(L) % returns false
isdarray(D) % returns true
```

See Also

`darray`, `distribute`, `zeros`

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 8-2.

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 distributed array is not replicated.

Examples

```
A = magic(3);
t = isreplicated(A); % returns t = true
B = magic(labindex);
f = isreplicated(B); % returns f = false
```

See Also `isdarray`

jobStartup

Purpose

M-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 function M-file resides in the worker's MATLAB installation at

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

You add M-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

`taskFinish`, `taskStartup`

Properties

`FileDependencies`

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.

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`

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`

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`

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.

See Also

`labBarrier`, `labindex`, `labProbe`, `labSend`

labSend

Purpose

Send data to another lab

Syntax

```
labSend(data, destination)
labSend(data, destination, tag)
```

Arguments

`data`	Data sent to the other lab; any MATLAB data type.
`destination`	`labindex` of receiving lab.
`tag`	Nonnegative integer to identify data.

Description

`labSend(data, destination)` sends the data to the specified destination, with a tag of `0`.

`labSend(data, destination, tag)` sends the data to the specified destination with the specified `tag`. `data` can be any MATLAB data type. `destination` identifies the `labindex` of the receiving lab, and must be either a scalar or a vector of integers between `1` and `numlabs`; it cannot be `labindex` (i.e., the current lab). `tag` can be any integer from `0` to `32767`.

Remarks

This function might return before the corresponding `labReceive` completes in the receiving lab.

See Also

`labBarrier`, `labindex`, `labProbe`, `labReceive`, `numlabs`

Purpose

Simultaneously send data to and receive data from another lab

Syntax

```
received = labSendReceive(labTo, labFrom, data)
received = labSendReceive(labTo, labFrom, data, tag)
```

Arguments

`data`	Data on the sending lab that is sent to the receiving lab; any MATLAB data type.
`received`	Data accepted on the receiving lab.
`labTo`	`labindex` of the lab to which data is sent.
`labFrom`	`labindex` of the lab from which data is received.
`tag`	Nonnegative integer to identify data.

Description

`received = labSendReceive(labTo, labFrom, data)` sends `data` to the lab whose `labindex` is `labTo`, and receives `received` from the lab whose `labindex` is `labFrom`. `labTo` and `labFrom` must be scalars. This function is conceptually equivalent to the following sequence of calls:

```
labSend(data, labTo);
received = labReceive(labFrom);
```

with the important exception that both the sending and receiving of data happens concurrently. This can eliminate deadlocks that might otherwise occur if the equivalent call to `labSend` would block.

If `labTo` is an empty array, `labSendReceive` does not send data, but only receives. If `labFrom` is an empty array, `labSendReceive` does not receive data, but only sends.

`received = labSendReceive(labTo, labFrom, data, tag)` uses the specified tag for the communication. `tag` can be any integer from `0` to `32767`.

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)
  1: mydata =
  1:      1
  2: mydata =
  2:      1     3
  2:      4     2
  3: mydata =
  3:      8     1     6
  3:      3     5     7
  3:      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)
  1: otherdata =
  1:      8     1     6
  1:      3     5     7
  1:      4     9     2
  2: otherdata =
  2:      1
  3: otherdata =
  3:      1     3
  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)
  1: otherdata =
  1:      []
  2: otherdata =
  2:      1
  3: otherdata =
  3:      1     3
  3:      4     2
```

See Also

`labBarrier`, `labindex`, `labProbe`, `labReceive`, `labSend` `numlabs`

length

Purpose Length of object array

Syntax

```
length(obj)
```

Arguments

`obj`	An object or an array of objects.

Description

`length(obj)` returns the length of `obj`. It is equivalent to the command `max(size(obj))`.

Examples

Examine how many tasks are in the job `j1`.

```
length(j1.Tasks)
ans =
     9
```

See Also

`size`

Purpose

Local portion of distributed array

Syntax

```
L = local(A)
```

Description

`L = local(A)` returns the local portion of a distributed array.

Examples

With four labs

```
A = magic(4)
D = distribute(A, 1)
L = local(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

`darray`, `distribute`, `partition`

localspan

Purpose

Index range of local segment of distributed array

Syntax

```
K = localspan(D)
[e, f] = localspan(D)
K = localspan(D, lab)
[e, f] = localspan(D, lab)
```

Description

The local span of a distributor is the index range in the distributed dimension for a distributed array on a particular lab.

`K = localspan(D)` returns a vector `K`, so that `local(D) = D(..., K, ...)` on the current lab.

`[e, f] = localspan(D)` returns two integers `e` and `f` so that `local(D) = D(..., e:f, ...)` on the current lab.

`K = localspan(D, lab)` returns a vector `K` so that `local(D) = D(..., K, ...)` on the specified lab.

`[e, f] = localspan(D, lab)` returns two integers `e` and `f` so that `local(D) = D(..., e:f, ...)` on the specified lab.

In all of the above syntaxes, if the partition is unspecified, then `K`, `e`, and `f` are all -1.

Examples

```
dist = darray('1d', 2, [6 6 5 5])
On lab 1, K = localspan(dist) returns K = 1:6.
On lab 2, [e, f] = localspan(dist) returns e = 7, f = 12.
K = localspan(dist, 3) returns K = 13:17.
[e, f] = localspan(dist, 4) returns e = 18, f = 22.
```

See Also

`distribdim`, `local`, `partition`

matlabpool

Purpose

Start parallel language worker pool

Syntax

```
matlabpool
matlabpool open
matlabpool open poolsize
matlabpool open conf
matlabpool open conf poolsize
matlabpool conf poolsize
matlabpool close
matlabpool close force
matlabpool close force conf
```

Description

`matlabpool` enables the parallel language features within the MATLAB language (e.g., `parfor`) by starting a parallel job which connects this MATLAB client with a number of labs.

`matlabpool` or `matlabpool` **`open`** starts a worker pool using the default configuration with the pool size specified by that configuration. You can also specify the pool size using `matlabpool` **`open`** `poolsize`, but note that most schedulers have a maximum number of processes that they can start (4 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`** `conf` or `matlabpool` **`open`** `conf poolsize` starts a worker pool using the Distributed Computing Toolbox user configuration `conf` 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.

`matlabpool conf poolsize` is the same as `matlabpool open conf poolsize` and is 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`** `conf` destroys all parallel jobs being run under the scheduler specified in the configuration `conf`.

`matlabpool` can be invoked as either a command or a function. For example, the following are equivalent:

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

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 Windows while the MATLAB workers are all running on Linux machines, the same argument to `addpath` cannot work on both. In this situation, you can use the function `dctRunOnAll` to assure that a command runs on all the workers.

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
```

See Also

`dctRunOnAll`, `parfor`

methods

Purpose

List functions of object class

Syntax

```
methods(obj)
out = methods(obj)
```

Arguments

`obj`	An object or an array of objects.
`out`	Cell array of strings.

Description

`methods(obj)` returns the names of all methods for the class of which `obj` is an instance.

`out = methods(obj)` returns the names of the methods as a cell array of strings.

Examples

Create 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`

mpiLibConf

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 an M-file called `mpiLibConf`, and place it on the MATLAB path. The recommended location is *`matlabroot`*`/toolbox/distcomp/user`.

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, the MPI library must support the following additional MPI-2 functions:

- `MPI_Open_port`
- `MPI_Comm_accept`
- `MPI_Comm_connect`

Examples

View the current MPI implementation library.

```
mpiLibConf
    mpich2.dll
```

Purpose

Profile parallel communication and execution times

Syntax

```
mpiprofile
mpiprofile on <options>
mpiprofile off
mpiprofile resume
mpiprofile clear
mpiprofile status
mpiprofile reset
mpiprofile info
mpiprofile viewer
mpiprofile('viewer', <profinfoarray>)
```

Description

`mpiprofile` enables or disables the parallel profiler data collection on a MATLAB worker running a parallel job. `mpiprofile` aggregates statistics on execution time and communication times. The statistics are collected in a manner similar to running the `profile` command on each MATLAB worker. By default, the parallel profiling extensions include array fields that collect information on communication with each of the other labs. This command in general should be executed in pmode or as part of a task in a parallel job.

`mpiprofile on <options>` starts the parallel profiler and clears previously recorded profile statistics.

`mpiprofile` takes the following options:

Option	Description
-detail mmex -detail builtin	This option specifies the set of functions for which profiling statistics are gathered. -detail mmex (the default) records information about M-functions, M-subfunctions, and MEX-functions. -detail builtin additionally records information about built-in functions such as eig or labReceive.
-messagedetail default -messagedetail simplified	This option specifies the detail at which communication information is stored. -messagedetail default collects information on a per lab instance. -messagedetail simplified turns off collection for PerLab data fields, which reduces the profiling overhead. If you have a very large cluster, you might want to use this option; however you will not get all the detailed inter-lab communication plots in the viewer. See mpiprofile info below.
-history -nohistory -historysize <size>	mpiprofile supports these options in the same way as the standard profile. No other profile options are supported by mpiprofile. Note that these three options have no effect on the data that is 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 example below.

`mpiprofile` does not accept `-timer clock` options, as the communication timer clock must be real.

For more information and examples on using the parallel profiler, see "Using the Parallel Profiler" on page 2-18.

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, darray)*rand(100, darray));
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` (the default is `'off'` for performance reasons). 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.

`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

```
% in "jobStartup.m" for a parallel job
mpiSettings('DeadlockDetection', 'on');
myLogFname = sprintf('%s_%d.log', tempname, labindex);
mpiSettings('MessageLoggingDestination', 'File', myLogFname);
mpiSettings('MessageLogging', 'on');
```

NaN

Purpose

Create distributed array of NaN values

Syntax

```
D = NaN(n, dist)
D = NaN(m, n, dist)
D = NaN([m, n], dist)
D = NaN(..., classname, dist)
```

Description

`D = NaN(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = NaN(m, n, dist)` and `D = NaN([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = NaN(..., classname, dist)` optionally specifies the class of the distributed array `D`. Valid choices are the same as for the regular `NaN` function: `'double'` (the default), and `'single'`.

Examples

With four labs,

```
D = NaN(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = NaN(10, 10, 'single', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed single array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`NaN` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `ones`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

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

Create distributed array of 1s

Syntax

```
D = ones(n, dist)
D = ones(m, n, dist)
D = ones([m, n], dist)
D = ones(..., classname, dist)
```

Description

`D = ones(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = ones(m, n, dist)` and `D = ones([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = ones(..., classname, dist)` optionally 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

With four labs,

```
D = ones(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = ones(10, 10, 'uint16', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed uint16 array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`ones` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

Purpose

Execute block of code 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 statements that execute in parallel on a cluster of labs. `initval` and `endval` must evaluate to finite integer values, or the range must evaluate to a value that can be obtained by such an expression, that is, an ascending row vector of consecutive integers.

The following table lists some ranges that are not valid.

Invalid parfor Range	Reason Range Not Valid
`parfor (i = 1:2:25)`	`1, 3, 5,...` are not consecutive.
`parfor (i = -7.5:7.5)`	`-7.5, -6.5,...` are not integers.
`A = [3 7 -2 6 4 -4 9 3 7];` `parfor (i = find(A>0))`	The resulting range, `1, 2, 4,...`, has nonconsecutive integers.
`parfor (i = [5;6;7;8])`	`[5;6;7;8]` is a column vector, not a row vector.

You can enter a `parfor`-loop on multiple lines, but if you put more than one segment of the loop statement on the same line, separate the segments with commas or semicolons:

```
parfor (i = range); <loop body>; end
```

`parfor (loopvar = initval:endval, M), statements, end` uses `M` to specify the maximum number of MATLAB workers that will evaluate statements in the body of the `parfor`-loop. `M` must be a nonnegative integer. By default, MATLAB uses as many workers as it finds available. If you specify an upper limit, MATLAB employs no more than that number, even if additional workers are available. Use the `matlabpool` command to make workers available for a `parfor`-loop.

If you request more resources than are available, MATLAB uses the maximum number available at the time of the call. If no workers are available, 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 be executed in any order.

Note Because of independence of iteration order, execution of parfor does not guarantee deterministic results.

For a detailed description of `parfor`-loops, see Chapter 3, "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 will depend only upon the number of iterations for which p(i) is true. As long as p(i) depends only upon i, the value of s will be 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

partition

Purpose

Partition of distributed array

Syntax

```
PAR = partition(dist)
```

Description

`PAR = partition(dist)` returns the partition of a distributed array, describing how the array is distributed among the labs.

Examples

```
partition(darray('1d', 2, [3 3 2 2]))
```

returns `[3 3 2 2]` .

See Also

`distribdim`, `localspan`

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`

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 distributed 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 distributed. Then save the data.

```
clear all;
rep = speye(numlabs);
var = magic(labindex);
D = eye(numlabs,darray());
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)
isdarray(D)
```

See Also

`load`, `save` MATLAB function reference pages

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

pmode

Purpose

Interactive parallel mode

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 four labs.

`pmode start conf numlabs` starts pmode using the Distributed 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 distributed 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 distributed 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

promote

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.

See Also

`createJob`, `demote`, `findJob`, `submit`

psave

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 distributed. Then save the data.

```
clear all;
rep = speye(numlabs);
var = magic(labindex);
D = eye(numlabs,darray());
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)
isdarray(D)
```

See Also

`load`, `save` MATLAB function reference pages

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

Purpose

Create distributed array of uniformly distributed pseudo-random numbers

Syntax

```
D = rand(n, dist)
D = rand(m, n, dist)
D = rand([m, n], dist)
D = rand(..., classname, dist)
```

Description

`D = rand(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = rand(m, n, dist)` and `D = rand([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = rand(..., classname, dist)` optionally specifies the class of the distributed array `D`. 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` 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,

```
D = rand(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = rand(10, 10, 'uint16', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed uint16 array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`rand` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `ones`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

randn

Purpose

Create distributed array of normally distributed random values

Syntax

```
D = randn(n, dist)
D = randn(m, n, dist)
D = randn([m, n], dist)
D = randn(..., classname, dist)
```

Description

`D = randn(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = randn(m, n, dist)` and `D = randn([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = randn(..., classname, dist)` optionally specifies the class of the distributed array `D`. 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 `randn` 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,

```
D = randn(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = randn(10, 10, 'uint16', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed uint16 array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`randn` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `ones`, `rand`, `sparse`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

redistribute

Purpose

Distribute array along different dimension

Syntax

```
D2 = redistribute(D1)
D2 = redistribute(D1, dim)
D2 = redistribute(D1, dim, part)
D2 = redistribute(D1, D3)
```

Description

`D2 = redistribute(D1)` redistributes a distributed array `D1` with its default distribution scheme. The distribution dimension `dim` is the last nonsingleton dimension and the partition is that specified by `dcolonpartition(size(D1,dim))` along the size of `D1` in the distribution dimension.

`D2 = redistribute(D1, dim)` redistributes a distributed array `D1` along dimension dim. The partition is that specified by `dcolonpartition(size(D1, dim))`. `dim` must be between `1` and `ndims(D1)`.

`D2 = redistribute(D1, dim, part)` redistributes a distributed array D1 along dimension `dim` using partition `part`.

`D2 = redistribute(D1, D3)` redistributes a distributed array `D1` using the same distribution scheme as `D3`.

Examples

Redistribute an array according to the distribution of another array. First, create a magic square distributed by columns.

```
M = distribute(magic(10), darray('1d', 2, [1 2 3 4]));
```

Create a pascal matrix distributed by rows (first dimension).

```
P = distribute(pascal(10), 1);
```

Redistribute the pascal matrix according to the distribution (partition) of the magic square.

```
R = redistribute(P, M);
```

See Also

`darray`, `dcolonpartition`, `distribdim`, `distribute`, `partition`

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 2-6.

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`

Purpose

Set options for submitting parallel jobs on LSF

Syntax

```
setupForParallelExecution(lsf_sched, 'pc')
setupForParallelExecution(lsf_sched, 'pcNoDelegate')
setupForParallelExecution(lsf_sched, 'unix')
```

Arguments

`lsf_sched`	LSF scheduler object.
`'pc'`, `'pcNoDelegate'`, `'unix'`	Setting for parallel execution.

Description

`setupForParallelExecution(lsf_sched, 'pc')` sets up the scheduler to expect Windows PC worker machines, 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 LSF schedules your job to run only on PC workers. For example, including `'-R type==NTX86'` in your `SubmitArguments` causes the scheduler to select only 32-bit Windows workers.

`setupForParallelExecution(lsf_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(lsf_sched, 'unix')` sets up the scheduler to expect UNIX worker machines, and selects the default wrapper script for UNIX workers. You still need to supply `SubmitArguments` that ensure LSF schedules your job to run only on UNIX workers. For example, including `'-R type==LINUX64'` in your `SubmitArguments` causes the scheduler to select only 64-bit Linux 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 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 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`

sparse

Purpose

Create distributed sparse matrix

Syntax

```
D = sparse(m, n, dist)
```

Description

`D = sparse(m, n, dist)` creates an m-by-n sparse distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)` and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its last nonsingleton dimension, or its second dimension if `m` and `n` are both 1 (`D` is scalar). If `PAR` is unspecified, then `D` uses `dcolonpartition` over the size in dimension `dim` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `sparse`.

Note To create a sparse distributed array of underlying class logical, first create an array of underlying class double and then cast it using the `logical` function:

```
logical(sparse(m, n, dist))
```

Examples

With four labs,

```
D = sparse(1000, 1000, darray())
```

creates a 1000-by-1000 distributed sparse double array `D`. `D` is distributed by its second dimension (columns), and each lab contains a 1000-by-250 local piece of `D`.

```
D = sprand(10, 10, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed sparse double array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`sparse` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `ones`, `rand`, `randn`, `speye`, `sprand`, `sprandn`, `true`, `zeros`

Purpose

Create distributed sparse identity matrix

Syntax

```
D = speye(n, dist)
D = speye(m, n, dist)
D = speye([m, n], dist)
```

Description

`D = speye(n, dist)` creates an n-by-n sparse distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `par=partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist=darray()`) as input to `speye`.

`D = speye(m, n, dist)` and `D = speye([m, n], dist)` create an m-by-n sparse distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` may be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D` and `PAR` is provided by `dcolonpartition` over the size in that dimension.

Note To create a sparse distributed array of underlying class logical, first create an array of underlying class double and then cast it using the `logical` function:

```
logical(speye(m, n, dist))
```

Examples

With four labs,

```
D = speye(1000, darray())
```

creates a 1000-by-1000 sparse distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = speye(10, 10, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 sparse distributed double array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`speye` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `ones`, `rand`, `randn`, `sparse`, `sprand`, `sprandn`, `true`, `zeros`

sprand

Purpose

Create distributed sparse array of uniformly distributed pseudo-random values

Syntax

```
D = sprand(m, n, density, dist)
```

Description

`D = sprand(m, n, density, dist)` creates an m-by-n sparse distributed array with approximately `density*m*n` uniformly distributed nonzero double entries. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, `D` is distributed by its second dimension. If `PAR` is unspecified, `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `sprandn`.

Remarks

When you use `sprand` 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,

```
D = sprand(1000, 1000, .001, darray())
```

creates a 1000-by-1000 sparse distributed double array `D` with approximately 1000 nonzeros. `D` is distributed by its second dimension (columns), and each lab contains a 1000-by-250 local piece of `D`.

```
D = sprand(10, 10, .1, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed double array `D` with approximately 10 nonzeros. `D` is distributed by its columns, and each lab contains a 10-by-`labindex` local piece of `D`.

See Also

sprand MATLAB function reference page

cell, eye, false, Inf, NaN, ones, rand, randn, sparse, speye, sprandn, true, zeros

sprandn

Purpose

Create distributed sparse array of normally distributed random values

Syntax

```
D = sprandn(m, n, density, dist)
```

Description

`D = sprandn(m, n, density, dist)` creates an m-by-n sparse distributed array with approximately `density*m*n` normally distributed nonzero double entries. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, `D` is distributed by its second dimension. If `PAR` is unspecified, `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `sprandn`.

Remarks

When you use `sprandn` 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,

```
D = sprandn(1000, 1000, .001, darray())
```

creates a 1000-by-1000 sparse distributed double array `D` with approximately 1000 nonzeros. `D` is distributed by its second dimension (columns), and each lab contains a 1000-by-250 local piece of `D`.

```
D = sprandn(10, 10, .1, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed double array `D` with approximately 10 nonzeros. `D` is distributed by its columns, and each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`sprandn` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `ones`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `true`, `zeros`

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

M-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 function M-file resides in the worker's MATLAB installation at

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

You add M-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`, `taskStartup`

Properties

`FileDependencies`

taskStartup

Purpose

M-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 function M-file resides in the worker's MATLAB installation at

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

You add M-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`, `taskFinish`

Properties

`FileDependencies`

Purpose

Create distributed true array

Syntax

```
T = true(n, dist)
T = true(m, n, dist)
T = true([m, n], dist)
```

Description

`T = true(n, dist)` creates an n-by-n distributed array of underlying class logical. `T` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `T` is distributed by its second dimension. If `PAR` is unspecified, then `T` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `true`.

`T = true(m, n, dist)` and `T = true([m, n], dist)` create an m-by-n distributed array of underlying class logical. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `T`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

Examples

With four labs,

```
T = true(1000, darray())
```

creates a 1000-by-1000 distributed double array `T`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `T`.

```
T = true(10, 10, darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed logical array `T`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `T`.

See Also

true MATLAB function reference page

cell, eye, false, Inf, NaN, ones, rand, randn, sparse, speye, sprand, sprandn, zeros

waitForState

Purpose

Wait for object to change state

Syntax

```
waitForState(obj)
waitForState(obj, 'state')
waitForState(obj, 'state', 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.

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.

Note Simulink models cannot run while MATLAB 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

Purpose

Create distributed array of 0s

Syntax

```
D = zeros(n, dist)
D = zeros(m, n, dist)
D = zeros([m, n], dist)
D = zeros(..., classname, dist)
```

Description

`D = zeros(n, dist)` creates an n-by-n distributed array of underlying class double. `D` is distributed by dimension `dim`, where `dim = distribdim(dist)`, and with partition `PAR`, where `PAR = partition(dist)`. If `dim` is unspecified, then `D` is distributed by its second dimension. If `PAR` is unspecified, then `D` uses `dcolonpartition(n)` as its partition. The easiest way to do this is to use a default distributor where both `dim` and `PAR` are unspecified (`dist = darray()`) as input to `eye`.

`D = zeros(m, n, dist)` and `D = zeros([m, n], dist)` create an m-by-n distributed array of underlying class double. The distribution dimension `dim` and partition `PAR` can be specified by `dist` as above, but if they are not specified, `dim` is taken to be the last nonsingleton dimension of `D`, and `PAR` is provided by `dcolonpartition` over the size in that dimension.

`D = zeros(..., classname, dist)` optionally 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

With four labs,

```
D = zeros(1000, darray())
```

creates a 1000-by-1000 distributed double array `D`, distributed by its second dimension (columns). Each lab contains a 1000-by-250 local piece of `D`.

```
D = zeros(10, 10, 'uint16', darray('1d', 2, 1:numlabs))
```

creates a 10-by-10 distributed uint16 array `D`, distributed by its columns. Each lab contains a 10-by-`labindex` local piece of `D`.

See Also

`zeros` MATLAB function reference page

`cell`, `eye`, `false`, `Inf`, `NaN`, `ones`, `rand`, `randn`, `sparse`, `speye`, `sprand`, `sprandn`, `true`

Properties — By Category

Job Manager Properties (p. 13-2)	Control job manager objects
Scheduler Properties (p. 13-3)	Control scheduler objects
Job Properties (p. 13-4)	Control job objects
Task Properties (p. 13-6)	Control task objects
Worker Properties (p. 13-7)	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

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterName`	Name of 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
`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
`MasterName`	Name of LSF master node
`MatlabCommandToRun`	MATLAB command that generic scheduler runs to start lab
`MpiexecFileName`	Specify pathname of executable mpiexec command
`ParallelSubmission-WrapperScript`	Script LSF scheduler runs to start labs
`ParallelSubmitFcn`	Specify function to run when parallel job submitted to generic scheduler
`SchedulerHostname`	Name of host running CCS scheduler

SubmitArguments	Specify additional arguments to use when submitting job to LSF 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
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 M-file function to execute when job is submitted to job manager queue
`RestartWorker`	Specify whether to restart MATLAB workers before evaluating job tasks
`RunningFcn`	Specify M-file function 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 LSF or mpiexec scheduler
`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

Task 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
`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
`NumberOfOutputArguments`	Number of arguments returned by task function
`OutputArguments`	Data returned from execution of task
`Parent`	Parent object of job or task
`RunningFcn`	Specify M-file function 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

`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

14

Properties — Alphabetical List

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 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.

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`

ClusterMatlabRoot

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 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`

ClusterOsType

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 CCS scheduler object
Data type	String

Values

The valid values for this property are `'pc'`, `'unix'`, and`'mixed'`.

- For CCS, the setting is always `'pc'`.
- A value of `'mixed'` is valid only for distributed jobs with 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, CCS, mpiexec, or generic scheduler must all be the same platform.
- For parallel jobs with an LSF 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 or local scheduler object
Data type	Double

Values

For job managers and local schedulers, this property is read-only. The value for a job manager represents the number of workers registered with that job manager. The value for a local scheduler is `4`.

For third-party schedulers (LSF, CCS, mpiexec, or generic), this property settable, and its value specifies the maximum number of workers or labs that this scheduler can start for running a job. For parallel jobs running on a third-party scheduler, the job's `MaximumNumberOfWorkers` property value should not exceed the value of `ClusterSize`.

See Also

Properties

`BusyWorkers`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

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

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 2-6.

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`

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`

DataLocation

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.

Examples

Set the `DataLocation` property for a UNIX 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 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

Error contains a structure which is the output from execution of the lasterror command if an error occurs during the task evaluation. The structure contains the following fields:

Field Name	Description
message	Character array containing the text of the error message.
identifier	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the identifier field is an empty character array.
stack	Structure providing information on the location of the error. The structure has fields file, name, and line, and is the same as the structure returned by the dbstack function. If lasterror returns no stack information, stack is a 0-by-1 structure having the same three fields.

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

`ErrorIdentifier` contains the identifier output from execution of the `lasterror` command if an error occurs during the task evaluation, or an identifier indicating that the task did not complete.

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`

ErrorMessage

Purpose

Message from task error

Description

`ErrorMessage` contains the message output from execution of the `lasterror` command if an error occurs during the task evaluation, or a message indicating that the task did not complete.

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 an 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`

FileDependencies

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, and stores it on the job manager machine.

The first time a worker evaluates a task for a particular job, the job manager passes to the worker the zip file of the files and directories in the `FileDependencies` property. On the worker, 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 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.)

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.

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

The default limitation on the size of data transfers via the `FileDependencies` property is approximately 50 MB. For information on increasing this limit, see "Object Data Size Limitations" on page

2-29. For alternative means of making data available to workers, see "Sharing Code" on page 6-25.

Examples

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'
```

See Also

Functions

`getFileDependencyDir`, `jobStartup`, `taskFinish`, `taskStartup`

Properties

`PathDependencies`

FinishedFcn

Purpose

Specify callback to execute after task or job runs

Description

The callback will be executed in the local MATLAB session, that is, the session that sets the property, the MATLAB client.

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 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	Never
Data type	Logical

Values

The value of `HasSharedFilesystem` can be set to `true` (or logical `1`) or `false` (or logical `0`). When you perform `get` on the property, the value returned is logical `1` or logical `0`.

See Also

Properties

`DataLocation`, `FileDependencies`, `PathDependencies`

HostAddress

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

`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

`HostAddress`

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`

IdleWorkers

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`

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` will be 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.

See Also

Functions

`createJob`, `createTask`

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`.

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 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`

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`

MinimumNumberOfWorkers

Purpose

Specify minimum number of workers to perform job tasks

Description

With `MinimumNumberOfWorkers` you specify the minimum number of workers to perform the evaluation of the job's tasks. When the job is queued, it will not run until at least this many workers are simultaneously available.

If `MinimumNumberOfWorkers` workers are available to the job manager, but some of the task dispatches fail due to network or node failures, such that the number of tasks actually dispatched is less than `MinimumNumberOfWorkers`, the job will be canceled.

Characteristics

Usage	Job object
Read-only	After job is submitted
Data type	Double

Values

The default value is `1`. You can set the value anywhere from `1` up to or equal to the value of the `MaximumNumberOfWorkers` property.

Examples

Set the minimum number of workers to perform a job.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
set(j, 'MinimumNumberOfWorkers', 6);
```

In this example, when the job is queued, it will not begin running tasks until at least six workers are available to perform task evaluations.

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

Purpose Specify pathname of executable mpiexec command

Description `MpiexecFileName` specifies which `mpiexec` command is executed to run your jobs.

Characteristics

Usage	mpiexec scheduler object
Read-only	Never
Data type	String

Remarks See your network administrator to find out which `mpiexec` you should run. The default value of the property points the `mpiexec` included in your MATLAB installation.

See Also

Functions

`mpiLibConf`, `mpiSettings`

Properties

`SubmitArguments`

Name

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 Engine System Administrator's Guide. This is reflected in the `Name` property of the object that represents the service. You can use the name of the job manager or worker service to search for the particular service when creating an object with the `findResource` function.

You can configure `Name` as a descriptive name for a job object at any time before the job is submitted to the queue.

Characteristics

Usage	Job manager object, job object, or worker object
Read-only	Always for a job manager or worker object; after job object is submitted
Data type	String

Values

By default, a job object is constructed with a `Name` created by concatenating the `Name` of the job manager with `_job`.

Examples

Construct a job manager object by searching for the name of the service you want to use.

```
jm = findResource('scheduler','type','jobmanager', ...
          'Name','MyJobManager');
```

Construct a job and note its default `Name`.

```
j = createJob(jm);
get(j, 'Name')
ans =
    MyJobManager_job
```

Change the job's `Name` property and verify the new setting.

```
set(j,'Name','MyJob')
get(j,'Name')
ans =
    MyJob
```

See Also

Functions

`findResource`, `createJob`

NumberOfBusyWorkers

Purpose

Number of workers currently running tasks

Description

The `NumberOfBusyWorkers` property value indicates how many workers are currently running tasks for the job manager.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Double

Values

The value of `NumberOfBusyWorkers` can range from 0 up to the total number of workers registered with the job manager.

Examples

Examine the number of workers currently running tasks for a job manager.

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

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfIdleWorkers`

Purpose

Number of idle workers available to run tasks

Description

The `NumberOfIdleWorkers` property value indicates how many workers are currently available to the job manager for the performance of job tasks.

If the `NumberOfIdleWorkers` is equal to or greater than the `MinimumNumberOfWorkers` of the job at the top of the queue, that job can start running.

Characteristics

Usage	Job manager object
Read-only	Always
Data type	Double

Values

The value of `NumberOfIdleWorkers` can range from 0 up to the total number of workers registered with the job manager.

Examples

Examine the number of workers available to a job manager.

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

See Also

Properties

`BusyWorkers`, `ClusterSize`, `IdleWorkers`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`

NumberOfOutputArguments

Purpose

Number of arguments returned by task function

Description

When you create a task with the `createTask` function, you define how many output arguments are expected from the task function.

Characteristics

Usage	Task object
Read-only	While task is running or finished
Data type	Double

Values

A matrix is considered one argument.

Examples

Create a task and examine its `NumberOfOutputArguments` property.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
get(t,'NumberOfOutputArguments')
ans =
     1
```

This example returns a 2-by-4 matrix, which is a single argument. The `NumberOfOutputArguments` value is set by the `createTask` function, as the argument immediately after the task function definition; in this case, the `1` following the `@rand` argument.

See Also

Functions

`createTask`

Properties

`OutputArguments`

Purpose Data returned from execution of task

Description `OutputArguments` is a 1-by-N cell array in which each element corresponds to each output argument requested from task evaluation. If the task's `NumberOfOutputArguments` property value is `0`, or if the evaluation of the task produced an error, `the` cell array is empty.

Characteristics

Usage	Task object
Read-only	Always
Data type	Cell array

Values The forms and values of the output arguments are totally dependent on the task function.

Examples Create a job with a task and examine its result after running the job.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
submit(j)
```

When the job is finished, retrieve the results as a cell array.

```
result = get(t, 'OutputArguments')
```

Retrieve the results from all the tasks of a job.

```
alltasks = get(j, 'Tasks')
allresults = get(alltasks, 'OutputArguments')
```

Because each task returns a cell array, `allresults` is a cell array of cell arrays.

See Also

Functions

createTask, getAllOutputArguments

Properties

Function, InputArguments, NumberOfOutputArguments

Purpose Script LSF scheduler runs to start labs

Description `ParallelSubmissionWrapperScript` identifies the script for the LSF scheduler to run when starting labs for a parallel job.

Characteristics

Usage	LSF 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 6-31.

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 7-7.

See Also

Functions

`createParallelJob`, `submit`

Properties

`MatlabCommandToRun`, `SubmitFcn`

Parent

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`

PathDependencies

Purpose

Specify directories to add to MATLAB worker path

Description

`PathDependencies` identifies directories to be added to the path of MATLAB worker sessions for this 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 and Windows 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 6-25.

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`

QueuedFcn

Purpose

Specify M-file function to execute when job is submitted to job manager queue

Description

`QueuedFcn` specifies the M-file function to execute when a job is submitted to a job manager queue.

The callback will be executed in the local MATLAB session, that is, the session that sets the property.

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`

RestartWorker

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`

Purpose

Specify M-file function to execute when job or task starts running

Description

The callback will be executed in the local MATLAB session, that is, the session that sets the property.

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`

SchedulerHostname

Purpose

Name of host running CCS scheduler

Description

`SchedulerHostname` indicates the name of the node on which the CCS scheduler is running.

Characteristics

Usage	CCS scheduler object
Read-only	Never
Data type	String

Values

`SchedulerHostname` is a string of the full name of the scheduler node.

See Also

Properties

`ClusterOsType`

StartTime

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 Distributed 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.
- `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`

Purpose

Specify additional arguments to use when submitting job to LSF or mpiexec scheduler

Description

`SubmitArguments` is simply a string that is passed via the `bsub` command to the LSF scheduler at submit time, or passed to the `mpiexec` command if using an mpiexec scheduler.

Characteristics

Usage	LSF or mpiexec scheduler object
Read-only	Never
Data type	String

Values

LSF

Useful `SubmitArguments` values might be `'-m "machine1 machine2"'` to indicate that your LSF scheduler should use only the named machines to run the job, or `'-R "type==LINUX64"'` to use only Linux 64-bit machines. Note that by default the LSF 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"'`.

mpiexec

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.

Value	Description
-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 6-31.

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 6-30.

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

Tasks contained in job object

Description

The `Tasks` property contains an array of all the task objects in a job, whether the tasks are pending, running, or finished. Tasks are always returned in the order in which they were created.

Characteristics

Usage	Job object
Read-only	Always
Data type	Array of task objects

Examples

Examine the `Tasks` property for a job object, and use the resulting array of objects to set property values.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
createTask(j, ...)
.
.
.
createTask(j, ...)
alltasks = get(j, 'Tasks')
alltasks =
     distcomp.task: 10-by-1
set(alltasks, 'Timeout', 20);
```

The last line of code sets the `Timeout` property value to `20` seconds for each task in the job.

See Also

Functions

createTask, destroy, findTask

Properties

Jobs

Purpose

Specify time limit to complete task or job

Description

Timeout holds a double value specifying the number of seconds to wait before giving up on a task or job.

The time for timeout begins counting when the task `State` property value changes from the `Pending` to `Running`, or when the job object `State` property value changes from `Queued` to `Running`.

When a task times out, the behavior of the task is the same as if the task were stopped with the `cancel` function, except a different message is placed in the task object's `ErrorMessage` property.

When a job times out, the behavior of the job is the same as if the job were stopped using the `cancel` function, except all pending and running tasks are treated as having timed out.

Characteristics

Usage	Task object or job object
Read-only	While running
Data type	Double

Values

The default value for `Timeout` is large enough so that in practice, tasks and jobs will never time out. You should set the value of `Timeout` to the number of seconds you want to allow for completion of tasks and jobs.

Examples

Set a job's `Timeout` value to 1 minute.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
set(j, 'Timeout', 60)
```

See Also

Functions

submit

Properties

ErrorMessage, State

Type

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.

UserData

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
```

UserName

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
```

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`

WorkerMachineOsType

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 with 32-bit Linux.

See Also

Properties

`HostAddress`, `HostName`

Glossary

CHECKPOINTBASE

The name of the parameter in the `mdce_def` file that defines the location of the job manager and worker checkpoint directories.

checkpoint directory

Location where job manager checkpoint information and worker checkpoint information is stored.

client

The MATLAB session that defines and submits the job. This is the MATLAB session in which the programmer usually develops and prototypes applications. Also known as the MATLAB client.

client computer

The computer running the MATLAB client.

cluster

A collection of computers that are connected via a network and intended for a common purpose.

coarse-grained application

An application for which run time is significantly greater than the communication time needed to start and stop the program. Coarse-grained distributed applications are also called embarrassingly parallel applications.

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 array

An array partitioned into segments, with each segment residing in the workspace of a different lab.

distributed computing
: Computing with distributed applications, running the application on several nodes simultaneously.

distributed computing demos
: Demonstration programs that use Distributed Computing Toolbox, 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 Distributed Computing Toolbox, 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 Engine. For a list of products that are not eligible for use with Distributed Computing Toolbox, 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.

MATLAB client
: See client.

MathWorks job manager
: See job manager.

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.

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 process 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
  distributed 8-4
  local 8-10
  private 8-4
  replicated 8-2
  types of 8-2
  variant 8-3

B

BusyWorkers property 14-2

C

cancel function 12-2
CaptureCommandWindowOutput property 14-3
CCS scheduler 6-18
ccsscheduler object 10-2
cell function 12-4
clear function 12-5
ClusterMatlabRoot property 14-5
ClusterName property 14-6
ClusterOsType property 14-7
ClusterSize property 14-8
CommandWindowOutput property 14-9
Configuration property 14-11
configurations 2-6
createJob function 12-6
createParallelJob function 12-8
createTask function 12-11
CreateTime property 14-13
current working directory
  MATLAB worker 2-15
CurrentJob property 14-14
CurrentTask property 14-15

D

darray function 12-14
DataLocation property 14-16
dcolon function 12-18
dcolonpartition function 12-19
dctconfig function 12-20
dctRunOnAll function 12-22
defaultParallelConfig function 12-23
demote function 12-25
destroy function 12-26
dfeval function 12-27
dfevalasync function 12-31
distribdim function 12-33
distribute function 12-34
distributed arrays
  constructor functions 8-10
  creating 8-7
  defined 8-4
  indexing 8-15
  working with 8-5
drange operator
  for loop 12-47

E

EnvironmentSetMethod property 14-18
Error property 14-19
ErrorIdentifier property 14-20
ErrorMessage property 14-21
eye function 12-35

F

false function 12-37
FileDependencies property 14-22
files
  sharing 6-12
    using an LSF scheduler 6-25
findJob function 12-39
findResource function 12-41
findTask function 12-45
FinishedFcn property 14-24

FinishTime property 14-26
for loop
    distributed 12-47
Function property 14-28
functions
    cancel 12-2
    cell 12-4
    clear 12-5
    createJob 12-6
    createParallelJob 12-8
    createTask 12-11
    darray 12-14
    dcolon 12-18
    dcolonpartition 12-19
    dctconfig 12-20
    dctRunOnAll 12-22
    defaultParallelConfig 12-23
    demote 12-25
    destroy 12-26
    dfeval 12-27
    dfevalasync 12-31
    distribdim 12-33
    distribute 12-34
    eye 12-35
    false 12-37
    findJob 12-39
    findResource 12-41
    findTask 12-45
    for
        distributed 12-47
        drange 12-47
    gather 12-49
    gcat 12-51
    get 12-52
    getAllOutputArguments 12-54
    getCurrentJob 12-56
    getCurrentJobmanager 12-57
    getCurrentTask 12-58
    getCurrentWorker 12-59
    getDebugLog 12-60
    getFileDependencyDir 12-62
    gop 12-63
    gplus 12-65
    help 12-66
    Inf 12-67
    inspect 12-69
    isdarray 12-71
    isreplicated 12-72
    jobStartup 12-73
    labBarrier 12-74
    labBroadcast 12-75
    labindex 12-77
    labProbe 12-78
    labReceive 12-79
    labSend 12-80
    labSendReceive 12-81
    length 12-84
    local 12-85
    localspan 12-86
    matlabpool 12-87
    methods 12-90
    mpiLibConf 12-92
    mpiprofile 12-93
    mpiSettings 12-98
    NaN 12-100
    numlabs 12-102
    ones 12-103
    parfor 12-105
    partition 12-108
    pause 12-109
    pload 12-110 12-116
    pmode 12-112
    promote 12-115
    rand 12-118
    randn 12-120
    redistribute 12-122
    resume 12-123
    set 12-124
    setupForParallelExecution 12-127
    size 12-129

sparse 12-130
speye 12-132
sprand 12-134
sprandn 12-136
submit 12-138
taskFinish 12-139
taskStartup 12-140
true 12-141
waitForState 12-143
zeros 12-145

G

gather function 12-49
gcat function 12-51
generic scheduler
  distributed jobs 6-30
  parallel jobs 7-7
genericscheduler object 10-4
get function 12-52
getAllOutputArguments function 12-54
getCurrentJob function 12-56
getCurrentJobmanager function 12-57
getCurrentTask function 12-58
getCurrentWorker function 12-59
getDebugLogp function 12-60
getFileDependencyDir function 12-62
gop function 12-63
gplus function 12-65

H

HasSharedFilesystem property 14-29
help
  command-line 1-11
help function 12-66
HostAddress property 14-30
HostName property 14-31

I

ID property 14-32
IdleWorkers property 14-34
Inf function 12-67
InputArguments property 14-35
inspect function 12-69
isdarray function 12-71
isreplicated function 12-72

J

job
  creating
    example 6-9
  creating on generic scheduler
    example 6-41
  creating on LSF or CCS scheduler
    example 6-21
  life cycle 2-4
  local scheduler 6-3
  submitting to generic scheduler queue 6-43
  submitting to local scheduler 6-5
  submitting to LSF or CCS scheduler
    queue 6-23
  submitting to queue 6-11
job manager
  finding
    example 6-3 6-7
job object 10-6
JobData property 14-36
jobmanager object 10-9
JobManager property 14-37
Jobs property 14-38
jobStartup function 12-73

L

labBarrier function 12-74
labBroadcast function 12-75
labindex function 12-77

labProbe function 12-78
labReceive function 12-79
labSend function 12-80
labSendReceive function 12-81
length function 12-84
local function 12-85
localscheduler object 10-11
localspan function 12-86
LSF scheduler 6-18
lsfscheduler object 10-13

M

MasterName property 14-39 14-60
MatlabCommandToRun property 14-40
matlabpool
    getting started 3-3
matlabpool function 12-87
MaximumNumberOfWorkers property 14-41
methods function 12-90
MinimumNumberOfWorkers property 14-42
mpiexec object 10-15
MpiexecFileName property 14-43
mpiLibConf function 12-92
mpiprofile function 12-93
mpiSettings function 12-98

N

Name property 14-44
NaN function 12-100
NumberOfBusyWorkers property 14-46
NumberOfIdleWorkers property 14-47
NumberOfOutputArguments property 14-48
numlabs function 12-102

O

objects 1-8
    ccsscheduler 10-2
    genericscheduler 10-4
    job 10-6
    jobmanager 10-9
    localscheduler 10-11
    lsfscheduler 10-13
    mpiexec 10-15
    paralleljob 10-17
    saving or sending 2-15
    simplejob 10-20
    simpleparalleljob 10-22
    simpletask 10-25
    task 10-27
    worker 10-29
ones function 12-103
OutputArguments property 14-49

P

parallel for-loops. *See* parfor-loops
parallel jobs 7-2
    supported schedulers 7-4
paralleljob object 10-17
ParallelSubmissionWrapperScript
    property 14-51
ParallelSubmitFcn property 14-52
Parent property 14-53
parfor function 12-105
parfor-loops 3-1
    break 3-9
    broadcast variables 3-17
    classification of variables 3-12
    compared to for-loops 3-5
    error handling 3-7
    for-drange 3-11
    global variables 3-9
    improving performance 3-26
    limitations 3-8
    local vs. cluster workers 3-10
    loop variable 3-13
    MATLAB path 3-7
    nested functions 3-9

- nested loops 3-9
- nondistributable functions 3-9
- persistent variables 3-9
- programming considerations 3-7
- reduction assignments 3-18
- reduction assignments, associativity 3-21
- reduction assignments, commutativity 3-22
- reduction assignments, overloading 3-23
- reduction variables 3-17
- release compatibility 3-11
- return 3-9
- sliced variables 3-14
- temporary variables 3-24
- transparency 3-8

partition function 12-108
PathDependencies property 14-54
pause function 12-109
platforms
- supported 1-7

pload function 12-110 12-116
pmode function 12-112
PreviousJob property 14-55
PreviousTask property 14-56
programming
- basic session 6-7
- guidelines 2-2
- local scheduler 6-2
- tips 2-15

promote function 12-115
properties
- BusyWorkers 14-2
- CaptureCommandWindowOutput 14-3
- ClusterMatlabRoot 14-5
- ClusterName 14-6
- ClusterOsType 14-7
- ClusterSize 14-8
- CommandWindowOutput 14-9
- Configuration 14-11
- CreateTime 14-13
- CurrentJob 14-14
- CurrentTask 14-15
- DataLocation 14-16
- EnvironmentSetMethod 14-18
- Error 14-19
- ErrorIdentifier 14-20
- ErrorMessage 14-21
- FileDependencies 14-22
- FinishedFcn 14-24
- FinishTime 14-26
- Function 14-28
- HasSharedFilesystem 14-29
- HostAddress 14-30
- HostName 14-31
- ID 14-32
- IdleWorkers 14-34
- InputArguments 14-35
- JobData 14-36
- JobManager 14-37
- Jobs 14-38
- MasterName 14-39 14-60
- MatlabCommandToRun 14-40
- MaximumNumberOfWorkers 14-41
- MinimumNumberOfWorkers 14-42
- MpiexecFileName 14-43
- Name 14-44
- NumberOfBusyWorkers 14-46
- NumberOfIdleWorkers 14-47
- NumberOfOutputArguments 14-48
- OutputArguments 14-49
- ParallelSubmissionWrapperScript 14-51
- ParallelSubmitFcn 14-52
- Parent 14-53
- PathDependencies 14-54
- PreviousJob 14-55
- PreviousTask 14-56
- QueuedFcn 14-57
- RestartWorker 14-58
- RunningFcn 14-59
- StartTime 14-61
- State 14-63

SubmitArguments 14-66
SubmitFcn 14-68
SubmitTime 14-69
Tag 14-70
Tasks 14-71
Timeout 14-73
Type 14-75
UserData 14-76
UserName 14-78
Worker 14-79
WorkerMachineOsType 14-80

Q

QueuedFcn property 14-57

R

rand function 12-118
randn function 12-120
redistribute function 12-122
RestartWorker property 14-58
results
local scheduler 6-5
retrieving 6-11
retrieving from job on generic scheduler 6-43
retrieving from job on LSF scheduler 6-24
resume function 12-123
RunningFcn property 14-59

S

saving
objects 2-15
scheduler
CCS 6-18
finding, example 6-20
generic interface
distributed jobs 6-30
parallel jobs 7-7
LSF 6-18
finding, example 6-19
set function 12-124
setupForParallelExecution function 12-127
simplejob object 10-20
simpleparalleljob object 10-22
simpletask object 10-25
size function 12-129
sparse function 12-130
speye function 12-132
sprand function 12-134
sprandn function 12-136
StartTime property 14-61
State property 14-63
submit function 12-138
SubmitArguments property 14-66
SubmitFcn property 14-68
SubmitTime property 14-69

T

Tag property 14-70
task
creating
example 6-10
creating on generic scheduler
example 6-42
creating on LSF scheduler
example 6-23
local scheduler 6-5
task object 10-27
taskFinish function 12-139
Tasks property 14-71
taskStartup function 12-140
Timeout property 14-73
troubleshooting
programs 2-29
true function 12-141
Type property 14-75

U

user configurations 2-6
UserData property 14-76
UserName property 14-78

W

waitForState function 12-143
worker object 10-29
Worker property 14-79
WorkerMachineOsType property 14-80

Z

zeros function 12-145