Distributed Computing Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 2

The MathWorks

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	Mail

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

Distributed Computing Toolbox User's Guide

Revision History

November 2004	Online only	New for Version 1.0 (Release 14SP1+)
March 2005	Online only	Revised for Version 1.0.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.0.2 (Release 14SP3)
November 2005	Online only	Revised for Version 2.0 (Release 14SP3+)
March 2006	Online only	Revised for Version 2.0.1 (Release 2006a)

Contents

Getting Started

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

What Are the Distributed Computing Products?

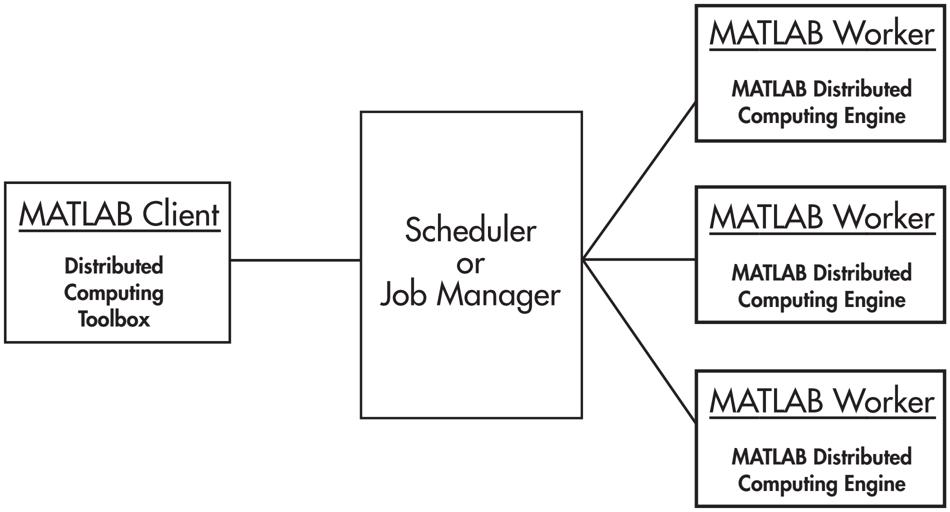
The Distributed Computing Toolbox and the MATLAB Distributed Computing Engine enable you to coordinate and execute independent MATLAB operations simultaneously on a cluster of computers, speeding up execution of large MATLAB jobs.

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 the Distributed Computing Toolbox to perform the definition of jobs and tasks. The 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 LSF.

See the "Glossary" for definitions of the distributed computing terms used in this manual.


MATLAB Client
Distributed Computing Toolbox
Scheduler or Job Manager
MATLAB Worker
MATLAB Distributed Computing Engine
MATLAB Worker
MATLAB Distributed Computing Engine
MATLAB Worker
MATLAB Distributed Computing Engine

Basic Distributed Computing Configuration

Determining Product Installation and Versions

To determine if the 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 the 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

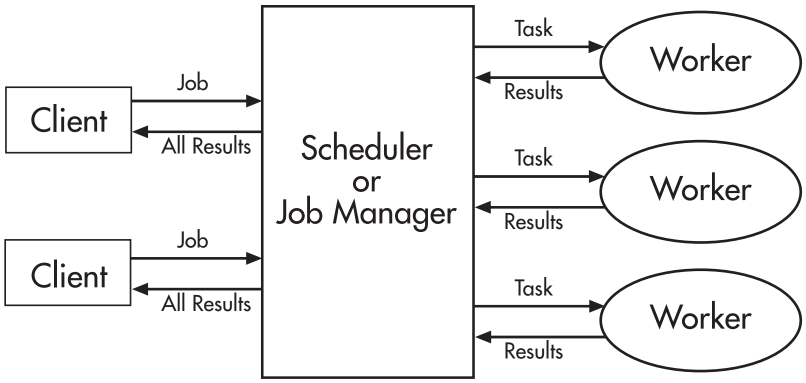
Job Managers, Workers, and Clients

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.

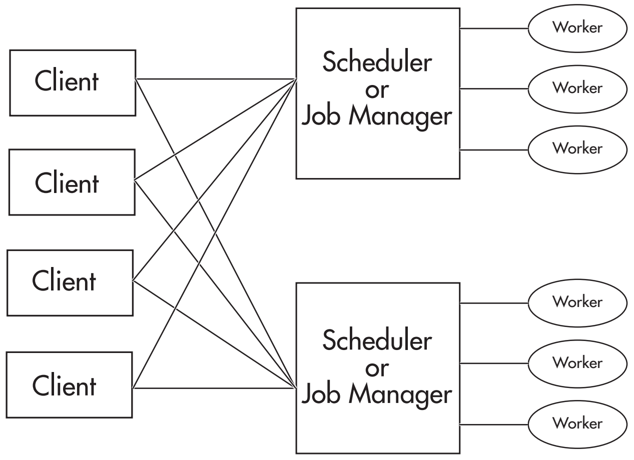
A MATLAB Distributed Computing Engine setup usually includes many workers that can all execute tasks simultaneously, speeding up execution of large MATLAB jobs. It is generally not important which worker executes a specific task. The workers evaluate tasks one at a time, returning the results to the 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

Third-Party Schedulers

As an alternative to using the MathWorks job manager, you can use a third-party scheduler. This could be a Platform Computing LSF scheduler, an mpiexec scheduler, or a generic scheduler.

Choosing between a 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

The 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.

The 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 the Distributed Computing Toolbox

Overview

A typical Distributed Computing Toolbox client session includes the following steps. Details of each step appear in "Creating and Running Jobs" on page 2-9. A basic example follows in the next section.

1. Find a Job Manager (or scheduler) — Your network may have one or more job managers available (but usually only one scheduler). The function you use to find a job manager or scheduler creates an object in your current MATLAB session to represent the job manager or scheduler that will run your job.

2. Create a Job — You create a job to hold a collection of tasks. The job exists on the job manager (or scheduler's data location), but a job object in the local MATLAB session represents that job.

3. Create Tasks — You create tasks to add to the job. Each task of a job can be represented by a task object in your local MATLAB session.

4. Submit a Job to the Job Queue for Execution — When your job has all its tasks defined, you submit it to the queue in the job manager or scheduler. The job manager or scheduler distributes your job's tasks to the worker sessions for evaluation. When all of the workers are completed with the job's tasks, the job moves to the finished state.

5. Retrieve the Job's Results — The resulting data from the evaluation of the job is available as a property value of each task object.

6. Destroy the Job — When a job is complete and you have its results, you might want to permanently remove the job from the job manager. Once a job is destroyed, its data is gone forever.

Example: Programming a Basic Job with a Job Manager

This example illustrates the basic steps in creating and running a job that contains a few simple tasks. Each task performs a `sum` on an input array.

1 Find a job manager. Use `findResource` to locate a job manager and create the job manager object `jm`, which represents the job manager in the cluster whose name is `MyJobManager` running on the host `JobMgrHost`.

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

2 Create a job. Create job `j` on the job manager.

```
j = createJob(jm);
```

3 Create tasks. Create three tasks on the job `j`. Each task evaluates the `sum` of the array that is passed as an input argument.

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

4 Submit the job to the queue. The job manager moves the job into the queue to be executed when workers are available.

```
submit(j);
```

5 Retrieve results. Wait for the job to complete, then get the results from all the job's tasks.

```
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 job manager.

```
destroy(j)
```

Example: Evaluating a Basic Function

The `dfeval` function allows you to evaluate a function in a cluster of workers without having to 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. Your cluster must use a MathWorks job manager for `dfeval` to work.

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

This example runs the job as three tasks in the same way the previous example does.

For more information about `dfeval` and in what circumstances you can use it, see "Evaluating Functions in a Cluster" on page 2-5.

Example: Programming a Basic Job with an LSF Scheduler

This example illustrates the basic steps in creating and running a job by using a third-party scheduler instead of a MathWorks job manager. Each task performs a `sum` on an input array.

1. **Find a scheduler.** Use `findResource` to locate a scheduler and create the scheduler object `sched`, which represents your cluster's LSF scheduler.

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

2. **Create a job.** Create job `j` on the scheduler.

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

3. **Create tasks.** Create three tasks on the job `j`. Each task evaluates the `sum` of the array that is passed as an input argument.

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

4. **Submit the job to the queue.** The scheduler moves the job into the queue to be executed when nodes are available.

   ```
   submit(j);
   ```

5. **Retrieve results.** Wait for the job to complete, then get the results from all the job's tasks.

   ```
   waitForState(j)
   ```

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

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

   ```
   destroy(j)
   ```

Getting Help

Command-Line Help

You can get command-line help on the object functions in the 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('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 The property or function name must be entered with lowercase letters, even though function names are case sensitive in other situations.

Programming Distributed and Parallel Applications

This chapter provides information you need for programming with the Distributed Computing Toolbox to define and run jobs. The sections are as follows.

Program Development Guidelines

When writing code for the 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 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.

2 **Run code distributed to only one node,** where that node is likely the local computer running a MATLAB worker in addition to your MATLAB client. Create a job and task to verify that the function is working in a distributed computing model.

3 **Distribute the code to two nodes.** Expand your job to include two tasks, preferably executed on two different workers on different computers.

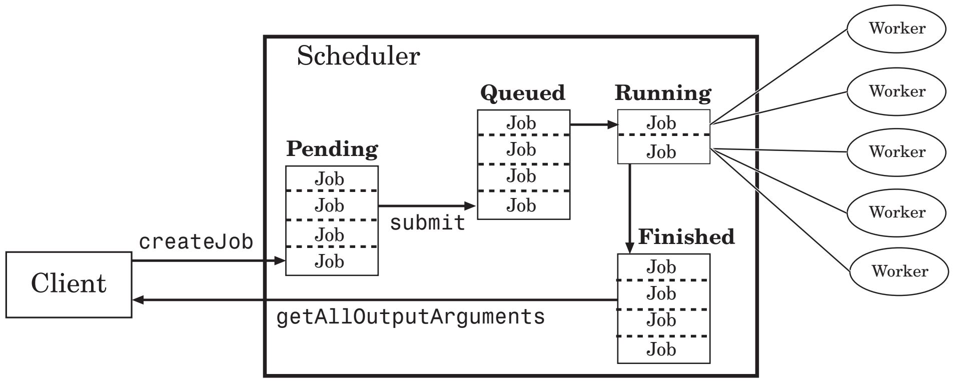
4 **Distribute the code to N nodes.** Scale up your job to include as many tasks as you need.

Note The client session of MATLAB must be running the Java Virtual Machine (JVM) to use the 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 the Distributed Computing Toolbox. The job's first state is `pending`. This is when you define the job by adding tasks to it.
Queued	When you execute the `submit` function on a job, the scheduler places the job in the queue, and the job's state is `queued`. The scheduler executes jobs in the queue in the sequence in which they are submitted, all jobs moving up the queue as the jobs before them are 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`.

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.

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. The 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" on page 2-5
- "Evaluating Functions Asynchronously" on page 2-7

Evaluating Functions Synchronously

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

Scope of dfeval

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 the 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
- 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 2-44, 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 job manager). 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.

Example: Using 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

```
[A, B] = dfeval(@myfun, {a b c d}, {e f g h}, {w x y z});
```

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, `A` and `B`. These cell arrays will have four elements each, for the four tasks. The first element of `A` will have the first output argument from the first task, the first element of `B` 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(a,e,w)`	`A{1},B{1}`
`myfun(b,f,x)`	`A{2},B{2}`
`myfun(c,g,y)`	`A{3},B{3}`
`myfun(d,h,z)`	`A{4},B{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.

```
[A{1}, B{1}] = myfun(a,e,w);
[A{2}, B{2}] = myfun(b,f,x);
[A{3}, B{3}] = myfun(c,g,y);
[A{4}, B{4}] = myfun(d,h,z);
```

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

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 off to the job manager and get access to the command line while the job is being run *asynchronously*, 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(@myfun, 2, {a b c d}, {e f g h}, {w x y z});
```

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')
data = getAllOutputArguments(Job1)
```

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 data will have two elements. So, `data{1,1}` contains the first output argument from the first task, `data{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 Job Manager” on page 2-9
- “Using an LSF Scheduler” on page 2-19
- “Using a Generic Scheduler” on page 2-27

Using a Job Manager

Creating and Running Jobs

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 the Distributed Computing Toolbox using a MathWorks job manager:

- “Find a Job Manager” on page 2-10
- “Create a Job” on page 2-11
- “Create Tasks” on page 2-12
- “Submit a Job to the Job Queue” on page 2-12
- “Retrieve the Job’s Results” on page 2-13

Note that the objects that the client session uses to interact with the job manager are only references to data that is actually contained in the job manager process, not in the client session. After jobs and tasks are created, you can shut down 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'}
                   Jobs: [0x1 double]
                  State: 'running'
          Configuration: ''
    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'}
                    Jobs: [1x1 distcomp.job]
                   State: 'running'
           Configuration: ''
     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 2-13.

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

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

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

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

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 2-14
- "Passing Data Between Sessions" on page 2-14
- "Passing M-Code for Startup and Finish" on page 2-15

Directly Accessing Files. If the workers all have access to the same drives on the network, they can access needed files that reside on these shared resources. This is the preferred method for sharing data, as it minimizes network traffic.

You must define each worker session's path so that it looks for files in the right places. You can define the path

- By using the job's `PathDependencies` property. This is the preferred method for setting the path, because it is specific to the job.
- By putting the `path` command in any of the appropriate startup files for the worker:
  - `$MATLAB\toolbox\local\startup.m` file
  - `$MATLAB\toolbox\distcomp\user\jobStartup.m` file
  - `$MATLAB\toolbox\distcomp\user\taskStartup.m` file.

  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 the 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.

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 2-51.

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

```
$MATLAB/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 2-16
- "Recovering Objects" on page 2-16
- "Permanently Removing Objects" on page 2-18

What Happens When the Client Session Ends? When you close the client session of the Distributed Computing Toolbox, all of the objects in the workspace are cleared. However, the objects in the 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 the Distributed Computing Toolbox can access any of the objects in the 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"
- "Starting a Job Manager from a Clean State"

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('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 an LSF Scheduler

If your network already uses a Load Sharing Facility (LSF), you can use the Distributed Computing Toolbox to create jobs to be distributed by your existing scheduler. The following sections provide instructions for using your LSF scheduler:

- “Creating and Running Jobs with an LSF Scheduler” on page 2-19
- “Sharing Code” on page 2-23
- “Managing Objects” on page 2-25

Creating and Running Jobs with an LSF Scheduler

This section details the steps of a typical programming session with the Distributed Computing Toolbox for jobs distributed to workers by an LSF scheduler.

This section assumes you have LSF installed and running on your network. For more information about LSF, see `http://www.platform.com/Products/`.

The following sections illustrate how to program the Distributed Computing Toolbox to use an LSF scheduler:

- “Find a Scheduler” on page 2-19
- “Create a Job” on page 2-20
- “Create Tasks” on page 2-22
- “Submit a Job to the Job Queue” on page 2-22
- “Retrieve the Job’s Results” on page 2-23

Find a 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 name 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.

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)
           DataLocation: '\\apps\data\project_55'
    HasSharedFilesystem: 1
                   Jobs: [0x1 double]
      ClusterMatlabRoot: '\\apps\matlab\'
            ClusterName: 'CENTER_MATRIX_CLUSTER'
             MasterName: 'masterhost.clusternet.ourdomain.com'
        SubmitArguments: ''
```

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)
              Type: 'job'
              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: []
```

Note that this job using an LSF scheduler 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)
           DataLocation: '\\apps\data\project_55'
    HasSharedFilesystem: 1
                   Jobs: [1x1 distcomp.simplejob]
       ClusterMatlabRoot: '\\apps\matlab\'
             ClusterName: 'CENTRAL_CLUSTER'
              MasterName: 'masterhost.clusternet.ourdomain.com'
         SubmitArguments: ''
```

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 2-13.

Note 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
```

Submit a Job to the Job Queue. To run your job and have its tasks evaluated, you submit the job to the LSF 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. 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`. 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 while the job's files might not yet have completed their transfer.

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 2-24
- "Passing Data Between Sessions" on page 2-24
- "Passing M-Code for Startup and Finish" on page 2-25

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:
  - `$MATLAB\toolbox\local\startup.m`
  - `$MATLAB\toolbox\distcomp\user\jobStartup.m`
  - `$MATLAB\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.
- `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.

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

```
$MATLAB/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 LSF 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 2-26
- “Recovering Objects” on page 2-26
- “Destroying Jobs” on page 2-27

What Happens When the Client Session Ends? When you close the client session of the Distributed Computing Toolbox, all of the objects in the workspace are cleared. However, job objects in the scheduler remain in place. 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 objects in the scheduler or their data.

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 objects. You can retrieve the job results later in another client session.

Recovering Objects. A client session of the 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 those objects contained in that scheduler. All the jobs contained in the scheduler are accessible in its `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 permanently removes from the scheduler those jobs whose data is in `/apps/data/project_88`. The `clear` function removes the object references from the local MATLAB client workspace.

Using a Generic Scheduler

You can use a generic scheduler to run jobs and distribute tasks to workers. A generic scheduler provides the means to interact with other third-party schedulers, or to create your own scripts for distributing tasks to other nodes on the cluster for evaluation.

The scheduler is a separate application that receives information from your MATLAB client session. With that information, the scheduler starts remote MATLAB worker sessions to evaluate individual tasks of the job. Whereas a job manager keeps MATLAB workers running between tasks, a third-party scheduler runs a MATLAB worker for only as long as it takes the worker to evaluate one task.

However, from the perspective of the MATLAB client, using a generic scheduler is similar to using a job manager or LSF scheduler. You create a scheduler object, jobs, and tasks. Some of the properties of the objects are different, but the theory of running your job is the same. You submit your job to the queue for execution and then retrieve the results. But unlike the MathWorks job manager, which manages the job data for you, a third-party or generic scheduler requires that you manage where job data is stored and assure that all the workers can access it.

The following sections describe how the Distributed Computing Toolbox interacts with your scheduler to run jobs.

- “Programming the Generic Scheduler Object in MATLAB” on page 2-28
- “Using the Submit Function” on page 2-29
- “Using the Decode Function” on page 2-32
- “Running a Job” on page 2-33

Programming the Generic Scheduler Object in MATLAB

This section illustrates how you write a program for running a job with a generic scheduler.

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'` will create a generic scheduler object.)

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

Set Scheduler Object Properties. 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 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.

If `DataLocation` is not set, the default location for job data is the current working directory of the MATLAB client, which might not be accessible to the worker nodes.

Set the `ClusterMatlabRoot` property to specify where the MATLAB installation is that the workers are to run for their tasks, if MATLAB is not on the worker's system path.

```
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.

```
get(sched)
          DataLocation: '\\apps\data\project_101'
   HasSharedFilesystem: 1
                  Jobs: [0x1 double]
     ClusterMatlabRoot: '\\apps\matlab\'
    MatlabCommandToRun: 'matlab -dmlworker -nodisplay -r
                         distcomp_evaluate_filetask'
                  Type: 'generic'
             SubmitFcn: []
```

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

```
set(sched,'SubmitFcn',@submitFunc)
```

Using the Submit Function

This section describes the user-defined submit function you use for a generic scheduler. This function, specified by the `SubmitFcn` property, runs in the MATLAB client when you submit a job to the generic scheduler's queue for execution. Generally, you need only one user-defined submit function for your cluster, and all applications running on that cluster can use the same function.

The user-defined submit function has three 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 the decode function
- To call the scheduler command to execute a job

Identify the Decode Function. Your submit function that runs on the MATLAB client has a complementary decode function that runs on the MATLAB worker. That decode function is identified in your submit function. For the worker to run the decode function, the function filename must be passed in the environment variable `MDCE_DECODE_FUNCTION`. This environment variable must exist on the worker computer before the scheduler starts the MATLAB worker session on that computer. The function's location must be on the path of the MATLAB worker.

Pass Location Data to the Worker. When the user-defined submit function is called, it receives at least three arguments: the scheduler object, the job object, and a properties object. (You can use additional arguments if necessary.) Through your submit function, you must make available to the workers some of the information from the properties object. This could be done through shared files, environment variables, or any other means convenient for your configuration.

To see what information is passed into your submit function, type

```
get(distcomp.setprop)

      StorageConstructor: ''
         StorageLocation: ''
             JobLocation: ''
           TaskLocations: {0x1 cell}
           NumberOfTasks: 0
```

Though the properties appear without any values, you can see what the property names are. In your submit function, you must make the values of these properties available for the decode function that will run from the MATLAB worker session on another node.

Call the Scheduler Command. With properties of the scheduler and job objects, you define the command for your scheduler to run a job. The following example illustrates a command composed in part from properties of the scheduler object and job being run.

Example Submit Function. Following is an example of a user-defined submit function that uses environment variables to pass information into the MATLAB worker sessions. This example runs worker sessions on the same machine. Commonly, you would have your function communicate with a third-party scheduler to distribute the tasks, or you might have your function run worker sessions on remote nodes with something like `ssh`.

For the complementary decode function, see "Example Decode Function" on page 2-33.

```
function submitFunc(scheduler, job, props, varargin)
% This helper function is used by generic schedulers. It prepares
% the environment for a MATLAB worker, and starts whichever
% MATLAB is first on the path.
% See also workerDecodeFunc.
%
% Assign the relevant values to environment variables, starting
% with identifying the decode function to be run by the worker:
decodeFcn = 'workerDecodeFunc';
dct_putenv('MDCE_DECODE_FUNCTION', decodeFcn);
%
% Set the other job-related environment variables:
dct_putenv('MDCE_STORAGE_LOCATION', props.StorageLocation);
dct_putenv('MDCE_STORAGE_CONSTRUCTOR',props.StorageConstructor);
dct_putenv('MDCE_JOB_LOCATION', props.JobLocation);
%
% Set the task-related variable:
prevDir = cd(fileparts(which(decodeFcn)));
for i = 1:props.NumberOfTasks
     dct_putenv('MDCE_TASK_LOCATION', props.TaskLocations{i});
  % Run a MATLAB worker for each task in the job:
  if ispc
     system(['start ' scheduler.MatlabCommandToRun]);
  else
     logLocation = [scheduler.DataLocation filesep ...
                       props.TaskLocations{i} '.log'];
```

```
        system([scheduler.MatlabCommandToRun ' > ' ...
                                        logLocation ' &']);
      end
    end
    cd(prevDir);
```

Using the Decode Function

This section describes the decode function run by the MATLAB worker that receives a task from your generic scheduler. The purpose of the decode function is to receive information from the scheduler about job and task data locations.

The decode function must be on the path of the MATLAB worker. It runs in the MATLAB worker as soon as that session begins. Generally, you need only one decode function for your cluster, and all applications running on that cluster can use the same function.

Because the scheduler object's submit function identifies the decode function, the decode and submit functions complement each other. That is, the decode function complies with the data transfer means defined in the submit function.

The decode function has an output argument that is an object whose properties reflect the information necessary for the MATLAB worker to evaluate its assigned task. To see the properties of this object, type

```
get(distcomp.runprop)

      StorageConstructor: ''
         StorageLocation: ''
             JobLocation: ''
            TaskLocation: ''
     DependencyDirectory: 'C:\Temp\tmp12345'
     HasSharedFilesystem: 1
```

The decode function must set the property values with data received from the complementary submit function, by whatever means data was made available. (The `DependencyDirectory` property is optional. This allows you to define where the `FileDependencies` are unzipped on the worker, as opposed to using the default location.)

Example Decode Function. Following is an example decode function that complements the user-defined submit function in the previous example. Because that submit function made the properties available as environment variables, this corresponding decode function reads the values of those environment variables and returns them to the MATLAB worker session.

For the complementary submit function, see "Example Submit Function" on page 2-31.

```
function workerDecodeFunc(runprop)
% This function is referenced by SubmitFunc. If a generic
% scheduler has been created with SubmitFcn = 'submitFunc', this
% function will be called on the MATLAB workers started by that
% generic scheduler.
% THIS FUNCTION MUST BE ON THE PATH OF THOSE MATLAB WORKERS;
% typically, this is accomplished by changing to the directory
% where this function is, before starting those MATLAB workers.
%
% Read environment variables into local variables. The names of
% the environment variables were determined by submitFunc.
storageConstructor = getenv('MDCE_STORAGE_CONSTRUCTOR');
storageLocation = getenv('MDCE_STORAGE_LOCATION');
jobLocation = getenv('MDCE_JOB_LOCATION');
taskLocation = getenv('MDCE_TASK_LOCATION');
%
% Set runprop properties from the local variables:
set(runprop, ...
    'StorageConstructor', storageConstructor, ...
    'StorageLocation', storageLocation, ...
    'JobLocation', jobLocation, ....
    'TaskLocation', taskLocation);
```

Running a Job

This section illustrates the running of a job on your generic scheduler from the MATLAB client session. With the scheduler running and the user-defined submit and decode functions defined, running a job is now similar to running with a job manager or any other type of scheduler.

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)
                Type: 'job'
                Name: 'Job1'
                  ID: 1
            UserName: 'neo'
                 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.genericscheduler]
            UserData: []
```

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

Note that 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)
           DataLocation: '\\apps\data\project_101'
    HasSharedFilesystem: 1
                   Jobs: [1x1 distcomp.simplejob]
      ClusterMatlabRoot: '\\apps\matlab\'
     MatlabCommandToRun: 'matlab -dmlworker -nodisplay -r
                          distcomp_evaluate_filetask'
                   Type: 'generic'
              SubmitFcn: @submitFunc
```

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
```

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`. So this function causes MATLAB to pause until the `State` property of `j` is `'finished'` or `'failed'`.

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
```

Programming Parallel Jobs

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 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.

Distributed Job	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.
Supported by any type of scheduler.	Supported only by mpiexec schedulers and the MathWorks job manager.

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.

The following sections describe how to program parallel jobs:

- "Using a Job Manager" on page 2-38
- "Using an mpiexec Scheduler" on page 2-40
- "Further Notes on Parallel Jobs" on page 2-43

Using a Job Manager

You can run a parallel job using the job manager or an mpiexec scheduler. This section illustrates a parallel job programmed for the job manager.

Coding the Task Function

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. More specifically, four labs run a parallel job with a 4-by-4 magic square. The first lab broadcasts the matrix to all the other labs, each of which calculates the sum of one column of the matrix. All of these column sums are combined 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 identied by labindex for this lab
    column_sum = sum(A(:,labindex))

    % Calculate total sum by combining column sum from all labs
    total_sum = gop(@plus, 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. In this case, 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 you choose will depend 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.

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

You create the job with the `createParallelJob` function.

```
pjob = createParallelJob(jm);
```

To run the job on four labs, set the job's properties that limit the number of workers.

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

The function file `colsum.m` 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'})
```

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 `gop` 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 an mpiexec Scheduler

Parallel jobs can run on an mpiexec scheduler. You might already have access to an mpiexec scheduler on your cluster, or your administrator can set one up using the mpiexec software included with the MATLAB Distributed Computing Engine.

This example estimates the value of pi by dividing up the integration of the function $4/(1 + x^2)$ over the range 0 to 1. The task function can accommodate any number of labs, but in this example the client code specifies the number of labs as 4.

Coding the Task Function

In this example, the task function requires no input arguments. As a lab runs the task, it is aware of how many labs are running the job, and divides the total integration interval into a number of subintervals. So, if you use four labs, each lab integrates one-fourth of the entire integral. The total number of labs is identified by the value `numlabs`, while each lab performing a task for this job has a unique `labindex`, starting with `1`, `2`, etc. In this example, `numlabs` is `4`, and the values of `labindex` are `1`, `2`, `3`, and `4`.

When all the labs have finished calculating their own portions of the integral, labs `2`, `3`, and `4` use the `labSend` function to transfer their results to lab `1`, which collects this data with the `labReceive` function.

```
function piApprox = quadpi
% QUADPI   Approximate pi via parallel numerical quadrature.

%   Copyright 2005 The MathWorks, Inc.

% Approximate pi by the numerical integral
% of F = 4/(1 + x^2) from 0 to 1.
F = @(x)4./(1 + x.^2);
% Each lab calculates the integral of F over a
% subinterval [a, b] of [0, 1].
a = (labindex - 1)/numlabs;
b = labindex/numlabs;
% Use a built-in MATLAB quadrature method to approximate
% the integral.
myIntegral = quadl(F,a,b);

% The labs have now all calculated their portions of the
% integral of F, and will all send their results to lab 1, which
% will add them together to form the entire integral over [0, 1].
if (labindex == 1)
    % Receive the integral contribution from all the other labs.
    piApprox = myIntegral;
    for otherLab = 2:numlabs
        piApprox = piApprox + labReceive(otherLab);
    end
else
    % Send the integral contribution to lab 1.
```

```
        piApprox = [];
        labSend(myIntegral, 1);
    end
```

Coding in the Client

To create a job for the mpiexec scheduler, first use `findResource` to find the scheduler and create a scheduler object. Then set the scheduler object properties with values appropriate for your network configuration.

```
sched = findResource('scheduler', 'type', 'mpiexec');
set(sched,'SubmitArguments', ['-phrase MATLAB -noprompt ' ...
     '-machinefile MyListOfNodes -pwdfile MyLoginFile']);
```

Create a parallel job.

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

Set the number of workers for the job to run on. This example uses exactly four workers, but you can use any number of workers that are simultaneously available to you.

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

Define the task of the parallel job with `createTask`. You create only one task, but the toolbox duplicates it to run simultaneously on all the workers (labs).

The file `quadpi.m` must be available to all the workers. It could be passed in the job's `FileDependencies` property, but in this example it is in a directory on the path of each of the workers, so the client code shown here does not have to mention the file or its location. Note that the task returns only one argument (the value of its portion of the integration), and that it has no input arguments (the empty cell array).

```
createTask(pjob, @quadpi, 1, {});
```

Use `submit` to run the job.

```
submit(pjob);
```

Make the MATLAB client wait for the job to finish before collecting the results.

```
waitForState(pjob, 'finished');
```

The result consists of the sum of the values from all of the labs, as collected by lab 1. The example compares its result to the built-in value of pi.

```
data = getAllOutputArguments(pjob);
piApprox = data{1};
fprintf('pi is approximately %.15f.\n', piApprox);
fprintf('Error is %g.\n', abs(pi - piApprox));
```

Further Notes on Parallel Jobs

Though you create only one task for a parallel job, the system creates additional identical tasks for each worker. So if a parallel job runs on four workers (labs), when the job is run, the `Tasks` property of the job will contain four task objects.

The first task in the job's `Tasks` property corresponds to the task run by the lab whose `labindex` is `1`. If there is some unique code for one of the labs, or if you need to be able to identify the results from a particular lab, you should use `labindex == 1` for that purpose.

For other labs, there is no correspondence to the sequence of objects in the `Tasks` property and the `labindex` value of the labs. However, the order of those tasks in the `Tasks` property does correspond to the sequence of results in the `OutputArguments` property and with those returned by the `getAllOutputArguments` function.

Programming with User Configurations

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

- `createJob`
- `createParallelJob`
- `createTask`
- `dfeval`
- `dfevalasync`
- `findResource`
- `set`

The following sections describe how to define and apply user configurations:

- "Defining Configurations" on page 2-44
- "Applying Configurations in Client Code" on page 2-45

Defining Configurations

The Distributed Computing Toolbox includes a file called `$MATLAB/toolbox/distcomp/user/distcompUserConfig.m`. To use configurations, you should copy this file to a directory that is higher on your MATLAB path than `$MATLAB/toolbox/distcomp/user`, and edit your copy so that it accurately reflects your scheduler and how you want to run your jobs.

The file contains configurations for each type of scheduler supported by the toolbox. Each configuration takes the name of the subfunction in which it is defined. For each configuration, there are listed several parameters or properties that you can set, arranged by object type. You can also add your own configuration to the file by following the instructions included in the file.

Example — Setting Properties in the User Configuration File

Suppose you want to set several properties for a job being run by a job manager. In the `distcompUserConfig.m` file, you edit the configuration called `jobmanager`.

Find the section of the file identified by the line

```
function conf = jobmanager()
```

In that section is a block of code that reads

```
 % Job properties
 conf.job.PathDependencies = {};
 conf.job.FileDependencies = {};
% The following job properties are specific to the job manager
 conf.job.RestartWorker = false;
 conf.job.MaximumNumberOfWorkers = inf;
 conf.job.MinimumNumberOfWorkers = 1;
 conf.job.Timeout = inf;
```

To set the maximum and minimum number of workers and the timeout for a job, edit the last three line in this section. For example,

```
conf.job.MaximumNumberOfWorkers = 4;
conf.job.MinimumNumberOfWorkers = 4;
conf.job.Timeout = 180;
```

When this configuration is applied to a job object, the job will run only on 4 workers, and have a timeout of 3 minutes.

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.

Finding Schedulers

When calling the `findResource` function, you can use configurations to identify a particular scheduler. For example,

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

This command finds the scheduler defined by the settings of the `jobmanager` configuration in the `distcompUserConfig.m` file. The advantage of configurations is that you can alter your schedule choices without changing your MATLAB application code. To accommodate different schedulers, the file includes configurations called `jobmanager`, `lsf`, `mpiexec`, and `generic`. You can also add your own configurations to the file.

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', 'lsf');
```

Properties applied to the `lsfsched` object are defined in the section of the configuration file that begins with the lines

```
function conf = lsf()
%LSF Return a sample configuration for an LSF cluster.
```

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','jobmanager')
job2 = createJob(jm)
set(job2,'Configuration','jobmanager')
```

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

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

When you apply a configuration to an object, all the properties defined in that configuration section of the `distcompUserConfig.m` file 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 change any of the job's properties that had been set by that configuration, the job's configuration property is cleared.

Writing Scheduler-Independent Jobs

Because the properties of scheduler, job, and task objects can be defined in a configuration file, 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');
```

In the configuration file, 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 or different kind of scheduler can now be made in the configuration, without any modification needed in the application.

Programming Tips and Notes

This section provides programming tips that might enhance your program performance.

Saving or Sending Objects

Do not use the `save` or `load` functions 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 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 2-16.

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 the Distributed Computing Toolbox on a Macintosh.

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.

Troubleshooting and Debugging

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

`$MATLAB/toolbox/distcomp/bin` (for UNIX or Macintosh)
`$MATLAB\toolbox\distcomp\bin\win32` (for Windows)

Job Managers

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

This file can be found at

`$MATLAB/toolbox/distcomp/bin/mdce_def.sh` (for UNIX or Macintosh)
`$MATLAB\toolbox\distcomp\bin\win32\mdce_def.bat` (for Windows)

The parameter in this file controlling the Java memory limit is `JOB_MANAGER_MAXIMUM_MEMORY`. You should set this limit to 4 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 2-14.

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 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.
- An mpiexec scheduler saves output messages in its debug log.
- 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 from 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 MATLAB property inspector to search for these tasks.

```
inspect(yourjob);
```

Alternatively, 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`.

Connection Problems Between 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 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.

Job Manager Cannot See 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.

Function Reference

This chapter describes the Distributed Computing Toolbox M-file functions that you use directly to evaluate MATLAB code in a cluster of computers.

Functions — By Category

General Toolbox Functions	Distributed Computing Toolbox functions not specific to a particular object type
Job Manager Functions	Functions that operate on a job manager object
Scheduler Functions	Functions that operate on a scheduler object that is not a job manager
Job Functions	Functions that operate on a job object
Task Functions	Functions that operate on a task object
Toolbox Functions Used in Parallel Jobs	Functions that are meaninful only within a parallel job
Toolbox Functions Used in MATLAB Workers	Functions that are meaningful only within a MATLAB worker session

General Toolbox Functions

`clear`	Remove objects from MATLAB workspace
`dctconfig`	Configure settings for Distributed Computing Toolbox client session
`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
`methods`	List functions of object class
`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
`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
`getDebugLog`	Read output messages from parallel job run by mpiexec scheduler
`mpiLibConf`	Location of MPI implementation
`mpiSettings`	Configure options for MPI communication

Job Functions

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

Toolbox Functions Used in Parallel Jobs

`gop`	Global operation across all labs
`labBarrier`	Block execution until all labs have reached 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 specified lab
`numlabs`	Total number of labs operating in parallel on current job

Toolbox Functions Used in MATLAB Workers

`getCurrentJob`	Job object whose task is currently being evaluated
`getCurrentJobmanager`	Get job manager object that distributed current task
`getCurrentTask`	Task object currently being evaluated in this worker session
`getCurrentWorker`	Worker object currently running this session

Functions — Alphabetical List

This section contains detailed descriptions of the Distributed Computing Toolbox functions. Each function reference page contains some or all of the following information:

- The function name
- The function purpose
- The function syntax

  Valid input argument and output argument combinations are shown. In some cases, an ellipsis (. . .) is used for the input arguments. This means that all preceding input argument combinations are valid for the specified output argument(s).
- A description of each argument
- A description of each function syntax
- Additional remarks about usage
- An example of usage
- Related functions and properties

Purpose

Cancel task or job

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.

Example

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: 'dce:task:cancelled'
       Timeout: Inf
    CreateTime: 'Fri Oct 22 11:38:39 EDT 2004'
     StartTime: 'Fri Oct 22 11:38:46 EDT 2004'
    FinishTime: 'Fri Oct 22 11:38:46 EDT 2004'
        Worker: []
        Parent: [1x1 distcomp.job]
      UserData: []
    RunningFcn: []
   FinishedFcn: []
```

See Also `destroy`, `submit`

Purpose

Remove objects from MATLAB workspace

Syntax

`clear obj`

Arguments

`obj`	An object or an array of objects.

Description

`clear obj` removes `obj` from the MATLAB workspace.

Remarks

If `obj` references an object in the job manager, it is cleared from the workspace, but it remains in the job manager. You can restore `obj` to the workspace with the `findResource`, `findJob`, or `findTask` function; or with the `Jobs` or `Tasks` property.

Example

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`

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 job's `DataLocation` property.

`obj = createJob(..., '`**`configuration`**`', 'ConfigurationName', ...)` creates a job object with the property values specified in the configuration `ConfigurationName`. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44.

Example

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.

```
disp(out{1}{1});
```

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.

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.

`pjob = createParallelJob(..., `**`'configuration'`**`, 'ConfigurationName',...)` creates a parallel job object with the property values specified in the configuration `ConfigurationName`. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44.

Example

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

```
obj = createTask(j, functionhandle, numoutputargs, inputargs)
obj = createTask(..., 'p1',v1,'p2',v2, ...)
obj = createTask(..., 'configuration', 'ConfigurationName', ...)
```

Arguments

`j`	The job that the task object is created in.
`functionhandle`	A handle to the function that is called when the task is evaluated.
`numoutputargs`	The number of output arguments to be returned from execution of the task function.
`inputargs`	A row cell array specifying the input arguments to be passed to the function `functionhandle`. Each element in the cell array will be passed as a separate input argument.
`p1`, *`p2`*	Task object properties configured at object creation.
`v1`, `v2`	Initial values for corresponding task object properties.

Description

`obj = createTask(j, functionhandle, numoutputargs, inputargs)` creates a new task object in job `j`, and returns a reference, `obj`, to the added task object.

`obj = 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.

`obj = createTask(..., '`**`configuration`**`', 'ConfigurationName', ...)` creates a task job object with the property values specified in the configuration `ConfigurationName`. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44.

Example

Create a job object.

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

Add a task object to be evaluated that 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});
```

See Also

`createJob`

dctconfig

Purpose

Configure settings for Distributed Computing Toolbox client session

Syntax

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

Arguments

p1	Property to configure. Supported properties are `'port'` and `'hostname'`.
`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 the Distributed Computing Toolbox. This is useful in environments where the choice of ports is limited. By default, the client session uses a random port to communicate with the other sessions of the 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 the 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. You can find the default value by calling `dctconfig` without any input arguments before any values have been set.

`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.

Example

View the current settings for hostname and port.

```
config = dctconfig
config =
        port: 0
    hostname: 'machine32'
```

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

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

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 to be 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`

Purpose

Remove job or task object from its 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.

Example

Destroy a job and its tasks.

```
jm = findResource('scheduler','type','jobmanager', ...
               'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm, 'Name', 'myjob');
t = createTask(j, @rand, 1, {10});
destroy(j);
clear t
clear j
```

Note that task `t` is also destroyed as part of job `j`.

See Also

`createJob`, `createTask`

dfeval

Purpose Evaluate function using cluster

Syntax

```
[y1,...,ym] = dfeval(F, x1,...,xn)
[y1,...,ym] = dfeval(F, x1,...,xn, '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 the 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. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44. Note that configurations enable you to use `dfeval` with any type of scheduler.

Example

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);
```

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 it returns immediately with a single output argument containing the job object that it has created and sent to the cluster.

`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. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44. Note that configurations enable you to use `dfeval` with any type of scheduler.

Example

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

Purpose

Find job objects stored in scheduler

Syntax

```
out = findJob(jm)
[pending queued running finished] = findJob(jm)
out = findJob(jm, 'p1', v1, 'p2', v2,...)
```

Arguments

`jm`	Scheduler object in which to find the job.
`pending`	Array of jobs in scheduler `jm` whose `State` is `pending`.
`queued`	Array of jobs in scheduler `jm` whose `State` is `queued`.
`running`	Array of jobs in scheduler `jm` whose `State` is `running`.
`finished`	Array of jobs in scheduler `jm` whose `State` is `finished`.
`out`	Array of jobs found in scheduler `jm`.
`p1`, *`p2`*	Job object properties to match.
`v1`, `v2`	Values for corresponding object properties.

Description

`out = findJob(jm)` returns an array, `out`, of all job objects stored in the scheduler `jm`. Jobs in the array will be ordered by `State` in the following order: `'pending'`, `'queued'`, `'running'`, `'finished'`; within the `'queued'` state, jobs are listed in the order in which they are queued.

`[pending queued running finished] = findJob(jm)` returns arrays of all job objects stored in the scheduler `jm`, by state. Jobs in the array `queued` will be in the order in which they are queued, with the job at `queued(1)` being the next to execute.

`out = findJob(jm, '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.

Jobs in the `queued` state are returned in the same order as they appear in the job queue service.

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'`, `'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, defined in `distcompUserConfig.m`.

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'`, `'LSF'`, `'mpiexec'`, or any string starting with `'generic'`. 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. For more information about which ports these processes use, see "Setting TCP Ports" in the MATLAB Distributed Computing Engine System Administrator's Guide.

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`. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44.

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`.

Example

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')
```

See Also

`findJob`, `findTask`

findTask

Purpose Task objects belonging to job object

Syntax

```
tasks = findTask(obj)
[pending running finished] = findTask(obj)
tasks = findTask(obj, 'p1', v1, 'p2', v2, ...)
```

Arguments

`obj`	Job object.
`tasks`	Returned task objects.
`pending`	Array of tasks in job `obj` whose `State` is `pending`.
`running`	Array of tasks in job `obj` whose `State` is `running`.
`finished`	Array of tasks in job `obj` whose `State` is `finished`.
`p1`, *`p2`*	Task object properties to match.
`v1`, `v2`	Values for corresponding object properties.

Description

`tasks = findTask(obj)` gets a 1-by-N array of task objects belonging to a job object `obj`.

`[pending running finished] = findTask(obj)` returns arrays of all task objects stored in the job object `obj`, sorted by state.

`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.

Example

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`

get

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');
```

Example

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.

Example 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`

getCurrentJobmanager

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`

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`

getCurrentWorker

Purpose

Worker object currently running this session

Syntax

`worker = getCurrentWorker`

Arguments

`worker`	The worker object that is currently evaluating the task that contains this function.

Description

`worker = getCurrentWorker` returns the worker object representing the session that is currently evaluating the task that calls this function.

Remarks

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

Example

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`

getDebugLog

Purpose

Read output messages from parallel job run by mpiexec scheduler

Syntax

```
str = getDebugLog (mpiexec, pjob)
```

Arguments

`str`	Variable to which messages are returned as a string expression.
`mpiexec`	Scheduler object referring to mpiexec scheduler, created by `findResource`.
`pjob`	Object identifying parallel job whose messages you want.

Description

`getDebugLog (mpiexec, job)` returns any output written to the standard output or standard error stream by the job identified by `pjob`, being run by the scheduler identified by `mpiexec`.

See Also

`findResource`, `createParallelJob`

gop

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))
```

Example 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`

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`.

Example

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 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`.

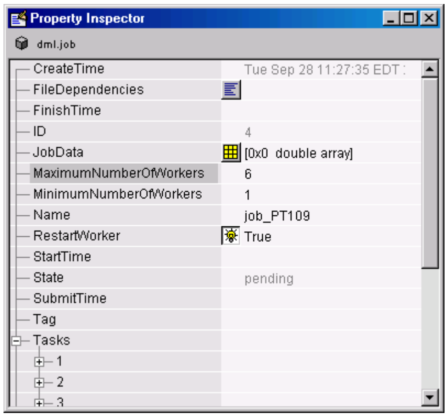
Remarks You can also open the Property Inspector via the Workspace browser by double-clicking an object.

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.

Example Open the Property Inspector for the job object `j1`.

```
inspect(j1)
```



Property Inspector

dml.job

CreateTime	Tue Sep 28 11:27:35 EDT :
FileDependencies	
FinishTime	
ID	4
JobData	[0x0 double array]
MaximumNumberOfWorkers	6
MinimumNumberOfWorkers	1
Name	job_PT109
RestartWorker	True
StartTime	
State	pending
SubmitTime	
Tag	
Tasks	
1	
2	
3	

See Also `get`, `set`

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 have reached 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.

Example 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`

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.

Example

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`

labindex

Purpose Index of this lab

Syntax

```
id = labindex
```

Description `id = labindex` returns the index of the lab currently executing the function. `labindex` is assigned to each lab when a job begins execution, and applies only for the duration of that job. The value of `labindex` spans from `1` to `n`, where `n` is the number of labs running the current job, defined by `numlabs`.

See Also `numlabs`

Purpose

Test to see if messages are ready to be received from other lab

Syntax

```
is_data_available = labProbe
is_data_available = labProbe(source)
is_data_available = labProbe('any', tag)
is_data_available = labProbe(source, tag)
[is_data_available, source, tag] = labProbe
```

Arguments

`source`	`labindex` of a particular lab from which to test for 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 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`

labReceive

Purpose

Receive data from another lab

Syntax

```
data = labReceive
data = labReceive(source)
data = labReceive('any', tag)
data = labReceive(source, tag)
[data, source, tag] = labReceive
```

Arguments

`source`	`labindex` of a particular lab from which to receive data.
`tag`	Tag defined by the sending lab's `labSend` function to identify particular data.
`'any'`	String to indicate that data can come from any lab.
`data`	Data sent by 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`

Purpose

Send data to another specified 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 nonnegative integer.

Remarks

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

See Also

`labBarrier`, `labindex`, `labProbe`, `labReceive`, `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))`.

Example

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

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

See Also

`size`

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.

Example

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  findJob    pause      resume

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

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

See Also

`help`

mpiLibConf

Purpose Location of MPI implementation

Syntax `[primaryLib, extras] = mpiLibConf`

Arguments

`primaryLib`

`extras`

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 `$MATLAB/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`

Example View the current MPI implementation library.

```
mpiLibConf
    mpich2.dll
```

Purpose

Configure options for MPI communication

Syntax

```
mpiSettings('DeadlockDetection', 'on')
mpiSettings('MessageLogging', 'on')
mpiSettings('MessageLoggingDestination', 'CommandWindow')
mpiSettings('MessageLoggingDestination', 'stdout')
mpiSettings('MessageLoggingDestination', 'File', <fname>)
```

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',<fname>)` 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.`

Example

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

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`

pause

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`

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

If the job object is not the first job in the queue, the position of `job` and the previous job object are exchanged.

See Also

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

resume

Purpose Resume processing queue in job manager

Syntax

```
resume(jm)
```

Arguments

`jm`	Job manager object whose queue is resumed.

Description `resume(jm)` resumes processing of the job manager's queue so that jobs waiting in the queued state will be run. This call will do nothing if the job manager is not paused.

See Also `pause`, `waitForState`

Purpose

Configure or display object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,'configuration', 'ConfigurationName',...)
set(obj,PN,PV)
set(obj,S)
```

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 configuration to use, defined in `distcompUserConfig.m`.

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`. Configurations are defined in the file `distcompUserConfig.m`. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44.

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`

size

Purpose

Size of object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,...,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,..., mn`	The lengths of the first `n` dimensions of `obj`.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

`length`

Purpose

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`.

Example

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`

taskFinish

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`

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 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.

Example

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`

Property Reference

This chapter describes the Distributed Computing Toolbox object properties in detail.

Properties — By Category

Job Manager Properties	Properties of job manager objects
Scheduler Properties	Properties of scheduler objects
Job Properties	Properties of job objects
Task Properties	Properties of task objects
Worker Properties	Properties of worker objects

Job Manager Properties

`BusyWorkers`	Workers currently running tasks
`Configuration`	Specify configuration to apply to object or toolbox function
`HostAddress`	IP address of host running job manager
`HostName`	Name of host running job manager
`IdleWorkers`	Idle workers available to run tasks
`Jobs`	Jobs contained in job manager
`Name`	Name of job manager
`NumberOfBusyWorkers`	Number of workers currently running tasks
`NumberOfIdleWorkers`	Number of idle workers available to run tasks
`State`	Current state of job manager

Scheduler Properties

`ClusterMatlabRoot`	Specify MATLAB root for cluster
`ClusterName`	Name of LSF cluster
`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 are to share `DataLocation`
`Jobs`	Jobs contained in directory identified by `DataLocation` property value
`MasterName`	Name of LSF master node
`MatlabCommandToRun`	MATLAB command that generic scheduler runs to start lab
`MpiexecFileName`	Specify pathname of executable mpiexec command
`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 generic scheduler
`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 job was created
`FileDependencies`	Directories and files that worker can access
`FinishedFcn`	Specify callback to execute after job runs
`FinishTime`	When 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`	Specify name for job object
`Parent`	Parent job manager object of job
`PathDependencies`	Specify directories to add to MATLAB worker path
`QueuedFcn`	Specify M-file function to execute when job is added to queue
`RestartWorker`	Specify whether to restart MATLAB workers before evaluating job tasks
`RunningFcn`	Specify M-file function to execute when job starts running
`StartTime`	When job started running
`State`	Current state of job object
`SubmitTime`	When job was submitted to job queue
`Tag`	Specify label to associate with job object
`Tasks`	Tasks contained in job object
`Timeout`	Specify time limit to complete job
`Type`	Type of object
`UserData`	Specify data to associate with job 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 was created
`ErrorIdentifier`	Task error identifier
`ErrorMessage`	Message from task error
`FinishedFcn`	Specify callback to execute after task runs
`FinishTime`	When task 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 job object of task
`RunningFcn`	Specify M-file function to execute when task starts running
`State`	Current state of task object
`StartTime`	When task started running
`Timeout`	Specify time limit to complete task
`Type`	Type of object
`UserData`	Specify data to associate with task object
`Worker`	Worker session that performed task

Worker Properties

`CurrentJob`	Job whose task this worker is currently evaluating
`CurrentTask`	Task that worker is currently running
`HostAddress`	IP address of host running worker session

`HostName`	Name of host running worker session
`Name`	Name of worker object
`PreviousJob`	Job whose task this worker previously ran
`PreviousTask`	Task that this worker previously ran
`State`	Current state of worker object

Properties — Alphabetical List

This section contains detailed descriptions of the Distributed Computing Toolbox object properties. Each property reference page contains some or all of the following information:

- The property name
- A description of the property
- The property characteristics, including
  - Usage — the object(s) the property is associated with
  - Read-only — the condition under which the property is read-only

    A property can be read-only always, never, or depending on the state of the object. You can configure a property value using the `set` command or dot notation. You can return the current property value using the `get` command or dot notation.
  - Data type — the property data type

    This is the data type you use when specifying a property value
- Valid property values including the default value

  When property values are given by a predefined list, the default value is usually indicated by `{}` (curly braces).
- An example using the property
- Related properties and functions

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.

Example

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

`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.

Example

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. For generic schedulers, `ClusterMatlabRoot` is prefixed to `MatlabCommandToRun`.

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`

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`

CommandWindowOutput

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.

Example

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. The configuration is defined in the `distcompUserConfig.m` file. For details about writing and applying configurations, see "Programming with User Configurations" on page 2-44.

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 in the file `distcompUserConfig.m`. 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.

Example

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.

Example Create a job, then get its Create`Time`.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm);
get(j,'CreateTime')
ans =
Mon Jun 28 10:13:47 EDT 2004
```

See Also

Functions

`createJob`, `createTask`

Properties

`FinishTime`, `StartTime`, `SubmitTime`

Purpose

Job whose task this worker session is currently evaluating

Description

`CurrentJob` indicates the job whose task the worker is evaluating at the present time.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Job object

Values

`CurrentJob` is an empty vector while the worker is not evaluating a task.

See Also

Properties

`CurrentTask`, `PreviousJob`, `PreviousTask`, `Worker`

CurrentTask

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. The default value is the directory in which MATLAB was started.

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.

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`

EnvironmentSetMethod

Purpose Specify means of setting environment variables for mpiexec scheduler

Description The mpiexec scheduler needs to supply environment variables to the MATLAB processes (labs) that it launches. There are two means by which it can do this, determined by the `EnvironmentSetMethod` property.

Characteristics

Usage	mpiexec scheduler object
Read-only	Never
Data type	String

Values A value of `'-env'` instructs the mpiexec scheduler to insert into the mpiexec command line additional directives of the form `-env VARNAME value`.

A value of `'setenv'` instructs the mpiexec scheduler to set the environment variables in the environment that launches mpiexec.

Purpose Task error identifier

Description `ErrorIdentifier` contains the identifier output from execution of the `lasterror` command if an error occurs during the task evaluation.

Characteristics

Usage	Task object
Read-only	Always
Data type	String

Values `ErrorIdentifier` is empty before an attempt to run a task. `ErrorIdentifier` 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.

See Also **Properties**

`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.

Characteristics

Usage	Task object
Read-only	Always
Data type	String

Values

`ErrorMessage` is empty before an attempt to run a task. `ErrorMessage` remains empty if the evaluation of a task object's function does not produce an error.

Example

Retrieve error message from 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

`ErrorIdentifier`, `Function`

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.

Example

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

jobStartup, taskFinish, taskStartup

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.

Example

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 clientTask`Completed.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`

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.

Example

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`

HasSharedFilesystem

Purpose

Specify whether nodes share DataLocation

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

Example

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

Example

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`

ID

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.

Example

Examine the ID property of different objects.

```
jm = findResource('scheduler','type','jobmanager', ...
           'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm)
createTask(j, @rand, 1, {2,4});
createTask(j, @rand, 1, {2,4});
tasks = get(j, 'Tasks');
get(tasks, 'ID')
ans =
    [1]
    [2]
```

The `ID` values are the only unique properties distinguishing these two tasks.

See Also

Functions

`createJob`, `createTask`

Properties

`Jobs`, `Tasks`

Purpose

Idle workers available to run tasks

Description

The `IdleWorkers` property value indicates which workers are currently available to the job manager for the performance of job tasks.

Characteristics

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

Values

As workers complete tasks and assume new ones, the lists of workers in `BusyWorkers` and `IdleWorkers` can change rapidly. If you examine these two properties at different times, you might see the same worker on both lists if that worker has changed its status between those times.

If a worker stops unexpectedly, the job manager's knowledge of that as a busy or idle worker does not get updated until the job manager runs the next job and tries to send a task to that worker.

Example

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`, `MaximumNumberOfWorkers`, `MinimumNumberOfWorkers`, `NumberOfBusyWorkers`, `NumberOfIdleWorkers`

InputArguments

Purpose

Input arguments to task object

Description

`InputArguments` is a 1-by-N cell array in which each element is an expected input argument to the task function. You specify the input arguments when you create a task with the `createTask` function.

Characteristics

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

Values

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

Example

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.

Example

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`

Jobs

Purpose

Jobs contained in job manager service or in scheduler's DataLocation

Description

The `Jobs` property contains an array of all the job objects in a job manager, whether the jobs are pending, queued, running, or finished. Job objects will be categorized by their `State` property and job objects in the `'queued'` state will be displayed in the order in which they are queued, with the next job to execute at the top (first).

Characteristics

Usage	Job manager or scheduler object
Read-only	Always
Data type	Array of job objects

Example

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 should send to a worker to start MATLAB for a task evaluation. To assure that the correct MATLAB is run, `MatlablCommandToRun` is prefixed by `ClusterMatlabRoot`.

Characteristics

Usage	Generic scheduler object
Read-only	Always
Data type	String

Values `MatlabCommandToRun` is set by the toolbox when the scheduler object is created.

See Also

Properties

`ClusterMatlabRoot`, `SubmitFcn`

Purpose

Specify maximum number of 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.

Example

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`, `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 cancelled.

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.

Example

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 6 workers are available to perform task evaluations.

See Also

Properties

`BusyWorkers`, `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 find the service you want when creating an object with the `findResource` function.

You configure `Name` as a descriptive name for a job object at any time except when the job is queued or running.

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`.

Example

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

```
jm = findResource('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.

Example 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`, `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.

Example

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`, `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
Data type	Double

Values A matrix is considered one argument.

Example 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.

Example

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`

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.

Examples Set the MATLAB worker path in a mixed-platform environment to use functions in both the central repository (`/central/funcs`) and the department archive (`/dept1/funcs`).

```
sch = findResource('scheduler','name','LSF')
job1 = createJob(sch)
p = {'/central/funcs','/dept1/funcs', ...
     '\\OurDomain\central\funcs','\\OurDomain\dept1\funcs'}
set(job1, 'PathDependencies', p)
```

See Also

Properties

`ClusterMatlabRoot`, `FileDependencies`

PreviousJob

Purpose Job whose task this worker previously ran

Description `PreviousJob` indicates the job whose task the worker most recently evaluated.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Job object

Values `PreviousJob` is an empty vector until the worker finishes evaluating its first task.

See Also

Properties

`CurrentJob`, `CurrentTask`, `PreviousTask`, `Worker`

PreviousTask

Purpose Task that this worker previously ran

Description `PreviousTask` indicates the task that the worker most recently evaluated.

Characteristics

Usage	Worker object
Read-only	Always
Data type	Task object

Values `PreviousTask` is an empty vector until the worker finishes evaluating its first task.

See Also

Properties

`CurrentJob`, `CurrentTask`, `PreviousJob`, `Worker`

Purpose

Specify 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.

Example

Create a job and set its `QueuedFcn` property, using a function handle to an anonymous function that sends information to the display.

```
jm = findResource('scheduler','type','jobmanager', ...
           'name','MyJobManager','LookupURL','JobMgrHost');
j = createJob(jm, 'Name', 'Job_52a');
set(j, 'QueuedFcn', ...
  @(job,eventdata) disp([job.Name ' now queued for execution.']))
.
.
.
submit(j)
Job_52a now queued for execution.
```

See Also

Functions

`submit`

Properties

`FinishedFcn`, `RunningFcn`

Purpose

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.

Example

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`

RunningFcn

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.

Example 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`

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.

Example 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`

State

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.

Example

Create a job manager object representing a job manager service, and create a job object; then examine each object's `State` property.

```
jm = findResource('scheduler','type','jobmanager', ...
          'name','MyJobManager','LookupURL','JobMgrHost');
get(jm, 'State')
ans =
    running
j = createJob(jm);
get(j, 'State')
ans =
  pending
```

See Also

Functions

`createJob`, `createTask`, `findResource`, `pause`, `resume`, `submit`

SubmitArguments

Purpose

Specify additional arguments to use when submitting job to 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 “Using the Submit Function” on page 2-29.

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 Submit Function” on page 2-29.

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.

Example

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

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.

Example

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

Example

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

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

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

See Also

Functions

`createTask`, `destroy`, `findTask`

Properties

`Jobs`

Timeout

Purpose

Specify time limit to complete task or job

Description

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

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

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

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

Characteristics

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

Values

The default value for Timeout is large enough so that in practice, tasks and jobs will never time out. You should set the value of Timeout to the number of seconds you want to allow for completion of tasks and jobs.

Example

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

Purpose

Type of object

Description

`Type` indicates the type of object.

Characteristics

Usage	Scheduler object, scheduler job object, or scheduler task object
Read-only	Always
Data type	String

Values

`Type` is a string set to `'task'`, `'job'`, or the name of the generic scheduler.

UserData

Purpose

Specify data to associate with job or task 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. 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, 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	Job object or task object
Read-only	Never
Data type	Any type

Values

The default value is an empty vector.

Example

Suppose you create the job object `job1`.

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

You can associate data with `job1` by storing it in `UserData`.

```
coeff.a = 1.0;
coeff.b = -1.25;
job1.UserData = coeff
get(job1,'UserData')
ans =
    a: 1
    b: -1.2500
```

Purpose User who created job

Description The `UserName` property value is a string indicating the login name of the user who created the job.

Characteristics

Usage	Job object
Read-only	Always
Data type	String

Example Examine a job to see who created it.

```
get(job1, 'UserName')
ans =
jsmith
```

Worker

Purpose

Worker session that performed task

Description

The `Worker` property value is an object representing the worker session that evaluated the task.

Characteristics

Usage	Task object
Read-only	Always
Data type	Worker object

Values

Before a task is evaluated, its `Worker` property value is an empty vector.

Example

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 Macintosh with Solaris 2.

See Also

Properties

`HostAddress`, `HostName`

Glossary

CHECKPOINT-BASE	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.
computer	A system with one or more processors.
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.
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 find-grained.
distributed computing	Computing with distributed applications, running the application on several nodes simultaneously.
distributed computing demos	Demonstration programs that use the Distributed Computing Toolbox, as opposed to sequential demos.
DNS	Domain Name System. A system that translates Internet domain names into IP addresses.
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.
MATLAB 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 lower-case 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.
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.
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.
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