Distributed Computing Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 1

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	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

Printing History:	November 2004	Online only	New for Version 1.0 (Release 14SP1+)
	March 2005	Online only	Revised for Version 1.0.1 (Release 14SP2)

Contents

1 Getting Started

2 Network Administration

3 Programming a Distributed Application

4 Function Reference

5 Property Reference

Index

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.

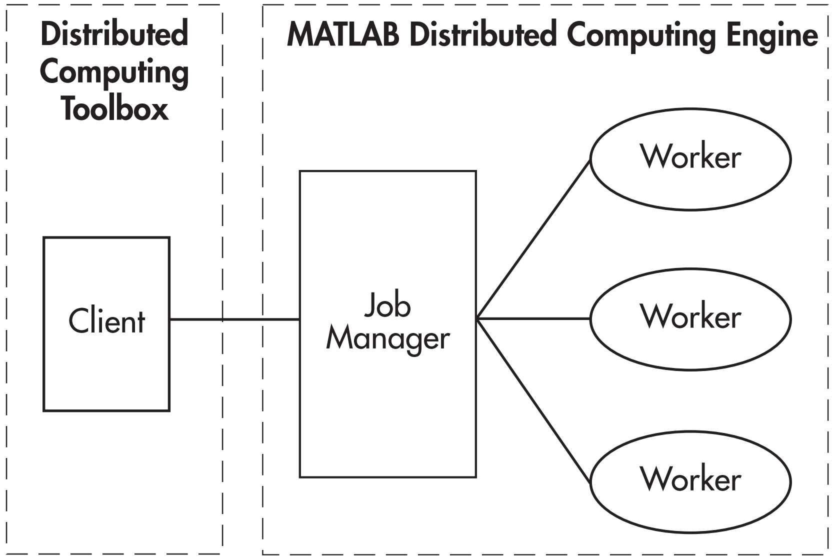
What Are the Distributed Computing Products?

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*.



Basic Distributed Computing Configuration

The following table summarizes the distributed computing terms introduced so far. The next section more fully explains these terms.

MATLAB Distributed Computing Terms

Name	Description
Job	The complete large-scale operation to perform in MATLAB, composed of a set of tasks.
Task	One segment of a job to be evaluated by a worker.
Client	The MATLAB session that defines a job using the Distributed Computing Toolbox.
Job manager	The part of the MATLAB Distributed Computing Engine that coordinates job execution, distributing tasks to individual workers for evaluation. This is represented in the client session by a job manager object.
Worker	The session of MATLAB in the MATLAB Distributed Computing Engine that evaluates tasks by executing the tasks' functions. This is represented in the client session by a worker object.

Toolbox and Engine Components

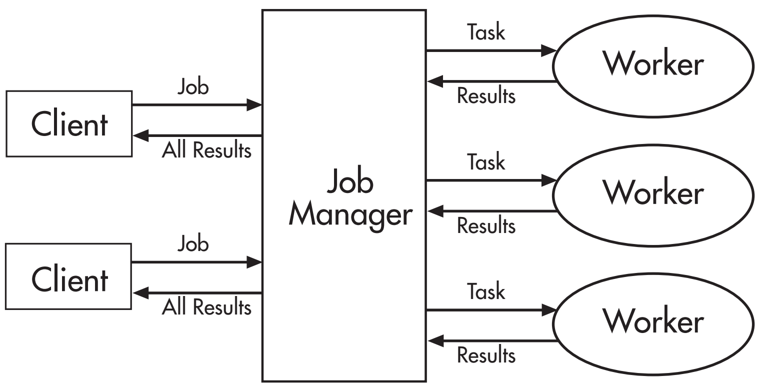
Job Managers, Workers, and Clients

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.

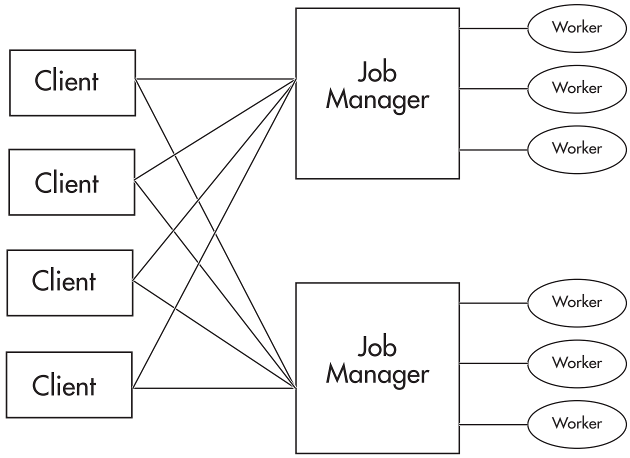
A MATLAB Distributed Computing Engine setup usually includes many workers that can all execute tasks simultaneously, speeding up execution of large MATLAB jobs. It is generally not important which worker executes a specific task. The workers evaluate tasks one at a time, returning the results to the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

Components on Mixed Platforms

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.

In a mixed platform environment, be sure to follow the proper installation instructions for the local machine on which you are installing the software.

The MATLAB Distributed Computing Engine Daemon

Every machine that hosts a worker or job manager session must also run the MATLAB Distributed Computing Engine (MDCE) Service. The MDCE daemon makes it possible for these processes on different machines to communicate with each other.

The MDCE daemon also recovers worker and job manager sessions when their host machines crash. If a worker or job manager machine crashes, when MDCE 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.

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. 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, 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 3-9. A basic example follows in the next section.

1. Find a Job Manager — Your network may have one or more job managers available. The function you use to find a job manager creates an object in your current MATLAB session to represent the job manager 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, but a job object in the local MATLAB session represents that job.

3. Create Tasks — While your job is in the pending state, you can 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 is completely defined with all its tasks, you submit it to the queue in the job manager. The job manager distributes your job's tasks to its workers for evaluation. When all of the workers are completed with the job's tasks, the job manager moves the job 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.

Example: Programming a Basic Job

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`.

   ```
   jm = findResource('jobmanager','name','MyJobManager');
   ```

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]
```

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.

```
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 3-5.

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.

2

Network Administration

This chapter provides information you need for network administration of the Distributed Computing Toolbox and the MATLAB Distributed Computing Engine. The sections are as follows.

Preparing for Distributed Computing

Before You Start

Before attempting an installation of the Distributed Computing Toolbox and MATLAB Distributed Computing Engine, read the "Getting Started" chapter to familiarize yourself with the concepts and vocabulary of the products.

Planning Your Network Layout

Generally, there is not much difficulty in deciding which machines will run worker processes and which will run client processes. Worker sessions usually run on the cluster of machines dedicated to that purpose. The client session of MATLAB usually runs where MATLAB programs are run, often on a user's desktop.

The job manager process should run on a stable machine, with adequate resources to manage the number of tasks and amount of data expected in your distributed computing applications.

The following table shows what products and processes are needed for each of these roles in the distributed computing configuration.

Session	Product	Processes
Client	Distributed Computing Toolbox	MATLAB with toolbox
Worker	MATLAB Distributed Computing Engine	MDCE service; worker
Job manager	MATLAB Distributed Computing Engine	MDCE service; job manager

The MATLAB Distributed Computing Engine (MDCE) service or daemon is included in the engine software. It is separate from the worker and job manager processes, and it must be running on all machines that run worker or job manager sessions.

You can install both toolbox and engine software on the same machine, so that one machine can run both client and engine sessions.

Network Requirements

The Distributed Computing Toolbox and the MATLAB Distributed Computing Engine require these network configurations:

- A network configuration that supports Jini.

  Jini technology facilitates communication between the machines and processes that comprise a distributed computing configuration.

  The MATLAB Distributed Computing Engine provides Jini as part of the job manager scripts, so you do not need to download or install it separately. Jini starts up automatically with the job manager service if it is not already running.

  For information about Jini network technology, go to the Web site `http://www.jini.org/`.

- Distributed computing processes rely on a DNS service being present on the network in order to locate one another.
- To allow communications between them, services of the MATLAB Distributed Computing Engine must be within multicast range of each other on the UDP port numbered 4160.
- Distributed computing processes make use of several TCP ports. If you need to control which ports these services use, see "Setting TCP Ports" on page 2-17.
- On UNIX systems, the command

  ```
  hostname -i
  ```

  must return the address of a network interface card (NIC) instead of the loopback address, so that distributed computing processes can recognize and communicate with each other. Distributed computing processes will work correctly on machines with multiple NICs.
- The job manager's checkpoint directories can grow to occupy a lot of disk space. Be sure to locate them where they can be accommodated. To control where the checkpoint directories are stored, see "Locating Checkpoint Directories" on page 2-19.

UNIX and Macintosh System Administration

This section describes the steps you take to configure and run the MATLAB Distributed Computing Engine on a cluster of UNIX and/or Macintosh machines:

- “Configuring the MDCE Daemon” on page 2-4
- “Starting Job Managers” on page 2-5
- “Starting Workers” on page 2-6
- “Stopping Job Managers and Workers” on page 2-7
- “Stopping and Uninstalling the MDCE Daemon” on page 2-8

Before You Start

Finding Your Installation

Throughout this section, MATLABROOT refers to the location of your installed MATLAB Distributed Computing Engine. Where you see this term used in the instructions that follow, substitute the path to your location or a link that points to it.

Getting Information from Logs

The MDCE services record their activities and messages in log files. If you encounter any problems or error messages during any steps of administering these services, consult the log files in the `/var/log/mdce` directory for more information. (See “Locating Log Files” on page 2-19 for information on changing the location of these log files.)

Configuring the MDCE Daemon

The MDCE daemon must be running on all machines being used for job managers or workers. This daemon facilitates communications between processes, and manages the MATLAB Distributed Computing Engine services. One of the major tasks of the MDCE daemon is to recover job manager and worker sessions after a system crash, so that jobs and tasks are not lost as a result of such accidents.

You need to configure the MDCE daemon on each machine that will be running a job manager or worker session.

Note You must have root privileges to install the MDCE daemon.

---

1 On UNIX systems that support `chkconfig` and that will be running job managers or worker sessions, enter the following commands. These commands register the MDCE daemon as a known service and configure it to start automatically at system boot time.

```
cd /etc/init.d/
ln -s MATLABROOT/toolbox/distcomp/bin/mdce mdce

chkconfig --add mdce
chkconfig --level 345 mdce on
```

Note To make use of `chkconfig` in a Red Hat Linux system, you might prefer to link to `MATLABROOT/toolbox/distcomp/bin/util/rh_mdce` rather than to `MATLABROOT/toolbox/distcomp/bin/mdce`, as is it customized for Red Hat Linux.

---

2 Start the MDCE daemon by typing the command

```
/etc/init.d/mdce start
```

or

Reboot your machine. Rebooting your machine starts the MDCE daemon.

Once installed, the MDCE daemon starts running each time the machine is rebooted. The MDCE daemon continues to run until explicitly stopped or uninstalled, regardless of whether a job manager or worker session is running.

Starting Job Managers

On the computer that will run the job manager, enter the following commands, using any text you want for the name `MyJobManager`.

```
cd MATLABROOT/toolbox/distcomp/bin
startjobmanager.sh -name MyJobManager
```

Note If you have more than one job manager on your cluster, each must have a unique name.

Where to Find More Information. The `startjobmanager` script has options that allow you to delete the job manager's history or alter the startup default parameters. For descriptions of these options, see "Overriding the Script Defaults" on page 2-14. For a command-line listing of all options, type

```
startjobmanager.sh -help
```

Starting Workers

On each computer used as a worker, enter the following commands, using the text for `MyJobManager` that identifies the name of the job manager you want this worker registered with.

```
cd MATLABROOT/toolbox/distcomp/bin
startworker.sh -jobmanager MyJobManager
```

To run a job manager session and a worker session on the same machine, run the scripts for each as usual.

To run more than one worker session on the same machine, give each worker a unique name with the `-name` option.

```
startworker.sh -jobmanager MyJobManager -name worker1
startworker.sh -jobmanager MyJobManager -name worker2
```

Where to Find More Information. The `startworker` script has options that allow you to delete the worker's history or alter the startup default parameters. For descriptions of these options, see "Overriding the Script Defaults" on page 2-14. For a command-line listing of all options, type

```
startworker.sh -help
```

Note If the number of threads created by the engine services on a UNIX machine exceeds the limitation set by the `maxproc` value, the services will fail and generate an out-of-memory error. You can check your `maxproc` value on UNIX with the `limit` command. (Different versions of UNIX might have different names for this property instead of `maxproc`, such as `descriptors` on Solaris.)

Stopping Job Managers and Workers

After all Distributed Computing Toolbox sessions are finished, you might want to shut down the engine network services so that they are not consuming network resources.

1 On the machine running the job manager, enter the commands

```
cd MATLABROOT/toolbox/distcomp/bin
stopjobmanager.sh -name MyJobManager
```

If you have more than one job manager running on this machine, you can stop each of them individually by name.

For a list of all options to the script, type

```
stopjobmanager.sh -help
```

2 On each machine running a worker session, enter the commands

```
cd MATLABROOT/toolbox/distcomp/bin
stopworker.sh
```

If you have more than one worker session running on this machine, you can stop each of them individually by name.

```
stopworker.sh -name worker1
stopworker.sh -name worker2
```

For a list of all options to the script, type

```
stopworker.sh -help
```

Stopping and Uninstalling the MDCE Daemon

Normally, you configure the MDCE daemon to start at system boot time and continue running until the machine is shut down. However, if you plan to uninstall the MATLAB Distributed Computing Engine from a machine, you might want to uninstall the MDCE daemon also, as it will not be needed any more.

Note You must have root privileges to stop or uninstall the MDCE daemon.

---

1. Use the following command to stop the MDCE daemon.

   ```
   /etc/init.d/mdce stop
   ```

2. Remove the installed link to prevent the daemon from starting up again at system reboot.

   ```
   cd /etc/init.d/
   rm mdce
   ```

Windows System Administration

This section describes the steps you take to configure and run the MATLAB Distributed Computing Engine on a cluster of Windows machines:

- "Configuring the MDCE Service" on page 2-9
- "Starting Job Managers" on page 2-11
- "Starting Workers" on page 2-11
- "Stopping Job Managers and Workers" on page 2-12
- "Stopping and Uninstalling the MDCE Service" on page 2-13

Before You Start

Finding Your Installation

Throughout this section, MATLABROOT refers to the location of your installed MATLAB Distributed Computing Engine. Where you see this term used in the instructions that follow, substitute the path to your location.

Getting Information from Logs

The MDCE services record their activities and messages in log files. If you encounter any problems or error messages during any steps of administering these services, consult the log files in the `<TEMP>\MDCE\Log` folder (typically, `C:\TEMP\MDCE\Log`) for more information. (See "Locating Log Files" on page 2-19 for information on changing the location of these log files.)

Configuring the MDCE Service

The MDCE service must be running on all machines being used for job managers or workers. This service facilitates communications between processes, and manages the MATLAB Distributed Computing Engine services. One of the major tasks of the MDCE service is to recover job manager and worker sessions after a system crash, so that jobs and tasks are not lost as a result of such accidents.

You need to install the MDCE service only once on each machine.

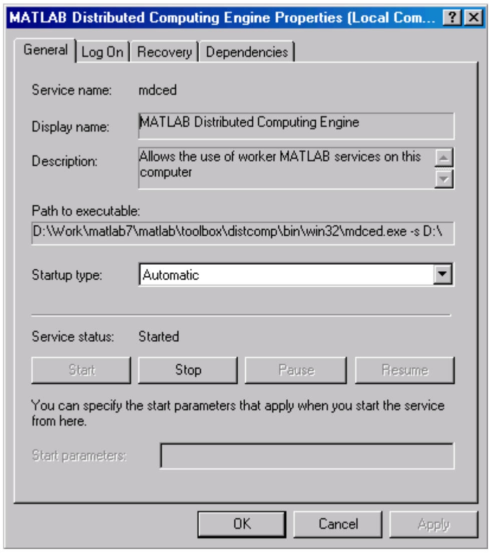
1 On all Windows PCs that will be running job managers or worker sessions, open a Command Prompt window and enter the following commands.

```
cd MATLABROOT\toolbox\distcomp\bin\win32
mdce install
```

This step installs the `mdce` service.

2 Verify the installation by going to **Start** -> **Control Panel** and double-clicking on `Administrative Tools`, then double-clicking on `Services`. In the list of services is `MATLAB Distributed Computing Engine Service`. Double-click on this entry to make the following dialog appear.

Note that the **Startup type** is `Automatic`. In this mode, the MDCE service starts up when the machine is rebooted.



3 To start the MDCE service without rebooting, click **Start** in the MATLAB Distributed Computing Engine Properties dialog box

or

Type in the Command Prompt window

```
cd MATLABROOT\toolbox\distcomp\bin\win32
mdce start
```

Once installed, the MDCE service starts running each time the machine is rebooted. The MDCE service continues to run until explicitly stopped or uninstalled, regardless of whether a job manager or worker session is running.

Starting Job Managers

On the Windows PC that will run the MATLAB Distributed Computing Engine job manager, open a Command Prompt window and enter the following commands, using any text you want for `MyJobManager`.

```
cd MATLABROOT\toolbox\distcomp\bin\win32
startjobmanager -name MyJobManager
```

Note If you have more than one job manager on your cluster, each must have a unique name.

Where to Find More Information. The `startjobmanager` script has options that allow you to delete the job manager's history or alter the startup default parameters. For descriptions of these options, see "Overriding the Script Defaults" on page 2-14. For a command-line listing of all options, type

```
startjobmanager -help
```

Starting Workers

On each Windows PC used as a worker, open a Command Prompt window and enter the following commands, using the text for `MyJobManager` that identifies the name of the job manager you want this worker registered with.

```
cd MATLABROOT\toolbox\distcomp\bin\win32
startworker -jobmanager MyJobManager
```

To run a job manager session and a worker session on the same machine, run the scripts for each as usual.

To run more than one worker session on the same machine, give each worker a unique name with the `-name` option.

```
startworker -jobmanager MyJobManager -name worker1
startworker -jobmanager MyJobManager -name worker2
```

Where to Find More Information. The `startworker` script has options that allow you to delete the worker's history or alter the startup default parameters. For descriptions of these options, see "Overriding the Script Defaults" on page 2-14. For a command-line listing of all options, type

```
startworker -help
```

Stopping Job Managers and Workers

After all Distributed Computing Toolbox sessions are finished, you might want to shut down the engine network services so that they are not consuming network resources.

1 On the Windows PC running the MATLAB Distributed Computing Engine job manager, open a Command Prompt window and enter the following commands.

   ```
   cd MATLABROOT\toolbox\distcomp\bin\win32
   stopjobmanager -name MyJobManager
   ```

   If you have more than one job manager running on this machine, you can stop each of them individually by name.

   For a list of all options to the script, type

   ```
   stopjobmanager -help
   ```

2 On each Windows PC running a worker session, enter the commands

   ```
   cd MATLABROOT\toolbox\distcomp\bin\win32
   stopworker
   ```

   If you have more than one worker session running on this machine, you can stop each of them individually by name.

   ```
   stopworker -name worker1
   stopworker -name worker2
   ```

   For a list of all options to the script, type

   ```
   stopworker -help
   ```

Stopping and Uninstalling the MDCE Service

Normally, you configure the MDCE service to start at system boot time and continue running until the machine is shut down. If you need to stop the MCDE service while leaving the machine on, open a Command Prompt window and enter the following commands.

```
cd MATLABROOT\toolbox\distcomp\bin\win32
mdce stop
```

If you plan to uninstall the MATLAB Distributed Computing Engine from a machine, you might want to uninstall the MDCE service also, as it will no longer be needed.

You do not need to stop the service before uninstalling it.

To uninstall the MDCE service, open a Command Prompt window and enter the following commands.

```
cd MATLABROOT\toolbox\distcomp\bin\win32
mdce uninstall
```

Customizing Engine Services

The scripts of the MATLAB Distributed Computing Engine run using several default parameters. You can customize the scripts, as described in the following sections:

- "Overriding the Script Defaults" on page 2-14
- "Defining the Script Defaults" on page 2-15

Overriding the Script Defaults

Specifying the Defaults File

The default parameters used by the MDCE service, job managers, and workers are defined in the file `MATLABROOT/toolbox/distcomp/bin/mdce_def.sh` (UNIX) or `MATLABROOT\toolbox\distcomp\bin\win32\mdce_def.bat` (Windows). Before starting the MDCE service, a job manager, or worker, you can edit this file to set the default parameters with values you require.

Alternatively, you can make a copy of this file, modify it, and specify that this copy be used for the default parameters.

On UNIX or Macintosh,

```
startjobmanager.sh -mdcedef my_mdce_def.sh
startworker.sh -mdcedef my_worker_def.sh
```

On Windows,

```
startjobmanager -mdcedef my_mdce_def.bat
startworker -mdcedef my_worker_def.bat
```

For more information, see "Defining the Script Defaults" on page 2-15.

Starting in a Clean State

When a job manager or worker starts up, it normally resumes its session from the past. This way, a job queue won't be destroyed or lost if the job manager machine crashes or if the job manager is inadvertently shut down. If you want to start up a job manager or worker from a clean state, with all history deleted, use the `-clean` flag on the `start` command.

On UNIX or Macintosh,

```
startjobmanager.sh -clean -name MyJobManager
startworker.sh -clean -jobmanager MyJobManager
```

On Windows,

```
startjobmanager -clean -name MyJobManager
startworker -clean -jobmanager MyJobManager
```

Defining the Script Defaults

The scripts for the engine services require values for several parameters. These parameters set the process name, the user name, log file location, ports, etc. Some of these can be set using flags on the command lines, but the full set of user-configurable parameters can be accessed in the `mdce_def` file.

Note The startup script flags take precedence over the settings in the `mdce_def` file.

The default parameters used by the engine service scripts are defined in the file `MATLABROOT\toolbox\distcomp\bin\win32\mdce_def.bat` (Windows), or `MATLABROOT/toolbox/distcomp/bin/mdce_def.sh` (UNIX/Macintosh). To set the default parameters, you edit this file before starting a service.

Alternatively, you can make a copy of this file, modify it, and specify that this new copy be used for the service default parameters using the `-mdcedef` flag.

Note that if you specify a new `mdce_def` file instead of the default file for one of the services, the new file is not automatically used by the other services. If you want to use the same alternative file for all your services, you must specify it for each service you call.

For example, on Windows systems, you use the parameter file `my_mdce_def.bat` by typing

```
mdce -mdcedef my_mdce_def.bat
startjobmanager -mdcedef my_mdce_def.bat
startworker -mdcedef my_mdce_def.bat
stopworker -mdcedef my_mdce_def.bat
stopjobmanager -mdcedef my_mdce_def.bat
```

On UNIX or Macintosh systems, you use the parameter file my_mdce_def.sh by typing

```
mdce -mdcedef my_mdce_def.sh
startjobmanager.sh -mdcedef my_mdce_def.sh
startworker.sh -mdcedef my_mdce_def.sh
stopworker.sh -mdcedef my_mdce_def.sh
stopjobmanager.sh -mdcedef my_mdce_def.sh
```

Note If a job manager is started with a name that was previously used on the same host, it will use the settings already established for that previous session. To use updated settings in the mdce_def file, use the -clean flag when starting the job manager again.

Setting TCP Ports

By default, job manager and worker sessions run on anonymous ports, though the job manager lookup service will use port 4160 if it is available. You can specify the port that a job manager or worker service runs on by specifying the following port settings in the mdce_def file.

Parameter	Description
JOB_MANAGER_UNICAST_PORT	This setting specifies the port for discovering the job manager's lookup service. The port must be known if you are going to use the 'LookupURL' option of the findResource function. You can set it by modifying it in the mdce_def file, or you can look up its value in the log file LOGBASE/mdce-service.log (if no port is displayed with the host in the entry for the service, it is using port 4160).
MIN_MDCE_PORT MAX_MDCE_PORT	These settings define a range of ports for the job manager and worker sessions to use. The MIN_MDCE_PORT value is used by the job manager's lookup service. If you run more than one job manger, those started after the first one should use different mdce_def files to define a unique range of ports for each job manager.
PHOENIX_PORT_1 PHOENIX_PORT_2	The MDCE service (daemon) uses these ports. Modify them if the default values are not available on your systems.

Note If you want to run more than one job manager on the same machine, they must all have unique names and unique port numbers. You can either specify these parameters using flags with the startup commands, or use different mdce_def files for each.

Setting the User

By default, the job manager and worker services run as the user who starts them. You can run the services as a different user with the following settings in the `mdce_def` file.

Parameter	Description
`MDCEUSER`	Set this parameter to run the MDCE services as a user different from the user who starts the service. On a UNIX system, set the value before starting the service; on a Windows system, set it before installing the service.
`MDCEPASS`	On a Windows system, set this parameter to specify the password for the user identified in the `MDCEUSER` parameter; otherwise, the system will prompt you for the password when the service is installed.

Note On UNIX systems, `MDCEUSER` requires that the current machine has the `sudo` utility installed, and that the current user be allowed to use `sudo` to execute commands as the user identified by `MDCEUSER`. For further information, refer to your system documentation on the `sudo` and `sudoers` utilities (for example, `man sudo` and `man sudoers`).

Note On Windows systems, when executing the `mdce start` script, the user defined by `MDCEUSER` must be listed among those who can log on as a service. To see the list of valid users, click the Windows **Start** -> **Settings** -> **Control Panel**. Double-click `Administrative Tools`, then `Local Security Policy`. In the tree, select `User Rights Assignment`, then in the right panel, double-click `Log on as a service`. This dialog must list the user defined for `MDCEUSER` in your `mdce_def.bat` file. If not, you can add the user to this dialog according to the instructions in the `mdce_def.bat` file, or when running `mdce start`, you can use another `mdce_def.bat` file that specifies a listed user.

Accessing Service Record Files

The services of the MATLAB Distributed Computing Engine generate various record files in the normal course of their operations. The MDCE service, job manager, and worker sessions all generate such files. The types of information stored by the services are described in the following sections:

- "Locating Log Files" on page 2-19
- "Locating Checkpoint Directories" on page 2-19

Locating Log Files

Log files for each service contain entries for the services' operations. These might be of particular interest to the network administrator in cases when problems arise.

Platform	File Location
Windows	On Windows systems, the default location of the log files is `<TEMP>\MDCE\Log`, where `<TEMP>` is the value of the system `TEMP` variable. For example, if `TEMP` is set to `C:\TEMP`, then the log files are placed in `C:\TEMP\MDCE\Log`. You can set alternative locations for the log files by modifying the `LOGBASE` setting in the `mdce_def.bat` file for any of the engine scripts.
UNIX and Macintosh	On UNIX and Macintosh systems, the default location of the log files is `/var/log/mdce/`. You can set alternative locations for the log files by modifying the `LOGBASE` setting in the `mdce_def.sh` file for any of the engine scripts.

Locating Checkpoint Directories

Checkpoint directories contain information related to persistence data, which the engine services use to create continuity from one instance of a session to another. For example, if you stop and restart a job manager, the new session will continue the old session, using all the same data.

A primary feature offered by the checkpoint directories is in crash recovery. This allows engine services to automatically resume their sessions after a system goes down and comes back up, without losing any data. (If a job manager crashes, its workers can take up to 2 minutes to reregister with it.)

Platform	File Location
Windows	On Windows systems, the default location of the checkpoint directories is <TEMP>\MDCE\Checkpoint, where <TEMP> is the value of the system TEMP variable. For example, if TEMP is set to C:\TEMP, then the checkpoint directories are placed in C:\TEMP\MDCE\Checkpoint. You can set alternative locations for the checkpoint directories by modifying the CHECKPOINTBASE setting in the mdce_def.bat file for any of the engine scripts.
UNIX and Macintosh	On UNIX and Macintosh systems, the checkpoint directories are placed by default in /var/lib/mdce/. You can set alternative locations for the checkpoint directories by modifying the CHECKPOINTBASE setting in the mdce_def.sh file for any of the engine scripts.

Controlling MDCE Sessions from a Script

Many clusters use batch processing systems such as a Portable Batch System (PBS) or Load Sharing Facility (LSF) in which a central job manager or scheduler manages the allocation of network resources to users. MDCE sessions can run within such an environment. A user submits a batch job to request some nodes in the cluster. The batch job starts the MDCE service and an MDCE job manager or worker on each of these nodes. The following sections show generic scripts for starting and stopping the MDCE sessions from the batch job:

Starting MDCE Sessions

To start MDCE sessions in the cluster from a batch job, use a script like the following. This example is generic:

```
# Select one node to be the job manager
JOB_MANAGER_NODE = 18

foreach <cluster nodes>
   <execute on node>:
       mdce start
       if <node> == JOB_MANAGER_NODE
           startjobmanager -name MyJobManager
       else
           startworker -jobmanager MyJobManager
       end
end
```

The Distributed Computing Toolbox client session runs on a machine that is not one of the cluster nodes, but it can access the job manager using `findResource` with a multicast call (not using the `LookupURL` option) or with a unicast call (using the `LookupURL` option).

Stopping MDCE Sessions

To stop MDCE sessions in the cluster from a batch job, use a script like the following generic example:

```
JOB_MANAGER_NODE = 18
foreach <cluster nodes>
   <execute on node>:
       if <node> == JOB_MANAGER_NODE
           stopjobmanager -name MyJobManager
       else
           stopworker -jobmanager MyJobManager
       end
       mdce stop
end
```

Running Sessions for a Specified Time

To prevent another cluster job from executing on a node while the node is running an MDCE session, use a script like the following. After a specified time, the batch job stops the MDCE sessions and the cluster job completes.

```
# Select one node to be the job manager
JOB_MANAGER_NODE = 18

foreach <cluster nodes>
   <execute on node>:
       mdce start
       if <node> == JOB_MANAGER_NODE
           startjobmanager -name MyJobManager
       else
           startworker -jobmanager MyJobManager
       end

       <wait allotted time>

       if <node> == JOB_MANAGER_NODE
           stopjobmanager -name MyJobManager
       else
           stopworker -jobmanager MyJobManager
       end
       mdce stop
end
```

Programming a Distributed Application

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.

2 **Run code distributed to only one node,** where that node is likely the local machine. 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.

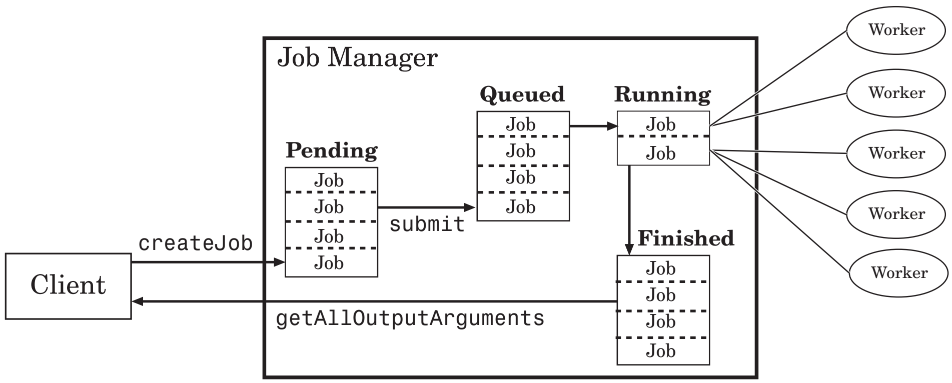
4 **Distribute the code to N nodes.** Scale up your job to include as many tasks as you need.

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 job manager 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 job manager places the job in the queue, and the job's state is `queued`. The job manager 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 job manager 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 job manager begins executing the next job. In this way, there can be more than one running job 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, even if you clear all the objects from the client session. The job manager 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.

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 3-5
- "Evaluating Functions Asynchronously" on page 3-7

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

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.
- You do not have direct access to the job manger, job, or task objects, i.e., there are no objects in your MATLAB workspace to manipulate (though you can get them from `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 in parallel 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 run the example of the previous section asynchronously, 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 right away. 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.

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:

- "Find a Job Manager" on page 3-9
- "Create a Job" on page 3-10
- "Create Tasks" on page 3-11
- "Submit a Job to the Job Queue" on page 3-12
- "Retrieve the Job's Results" on page 3-12

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, restart it, and your job will still be 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

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.

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

If you do not specify any property values to search on, `findresource` returns all available job managers.

```
all_managers = findResource('jobmanager')
```

You can 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)
```

To find a specific job manager, use parameter-value pairs for matching.

```
jm = findResource('jobmanager', 'Name', 'MyJobManager')
get(jm)
                  Name: 'MyJobManager'
              Hostname: 'bonanza'
           HostAddress: '123.123.123.123'
                  Jobs: [0x1 double]
                 State: 'running'
   NumberOfBusyWorkers: 0
           BusyWorkers: [0x1 double]
   NumberOfIdleWorkers: 2
           IdleWorkers: [2x1 distcomp.worker]
```

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}
                 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'
      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 Data" on page 3-14.

Create Tasks

After you have created your job, and while it is still in the pending state, 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. 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` properties 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.

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

```
results = getAllOutputArguments(job1);
```

Display the results from each task.

```
for i = 1:5
  disp(results{i})
end
    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 Data

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 data are explained in the following sections:

- “Directly Accessing Files” on page 3-14
- “Passing Data Between Sessions” on page 3-15
- “Passing M-Code for Startup and Finish” on page 3-15

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.

Defining the Path

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

- When MATLAB starts up on a worker by putting the `path` command in the worker machine’s `MATLABROOT\toolbox\local\startup.m` file
- For each new job on a worker by putting the command in the worker’s `MATLABROOT\toolbox\distcomp\user\jobStartup.m` file
- For each new task on a worker by putting the command in the worker’s `MATLABROOT\toolbox\distcomp\user\taskStartup.m` file.

Setting the User Name

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 a `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” on page 2-15.

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.
- `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 worker MATLAB session.

The 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.

Passing M-Code for Startup and Finish

As a session of MATLAB, a worker session executes its `startup.m` file each time it starts. You can place the `startup.m` file in any directory on the worker's MATLAB path, such as `toolbox/distcomp/user`.

Three additional M-files can initialize and clean up a worker session as it begins or completes evaluations of tasks for a job:

- `jobStartup.m` automatically executes on a worker when the worker runs its first task of a job.
- `taskStartup.m` automatically executes on a worker each time the worker begins evaluation of a task.
- `taskFinish.m` automatically executes on a worker each time the worker completes evaluation of a task.

Empty versions of these files are provided in the directory

```
MATLABROOT/toolbox/distcomp/user
```

You can edit these files to include whatever M-code you want the worker to execute at the indicated times.

Alternatively, you can create your own versions of these M-files and pass them to the job as part of the `FileDependencies` property.

The worker gives precedence to the versions provided in the `FileDependencies` property. If any of these files is not included in that property, 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 3-17
- "Recovering Objects" on page 3-17
- "Permanently Removing Objects" on page 3-18

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 these object were created by the current client session or another client session.

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

```
jm = findResource('jobmanager','Name','Job_Mgr_123')
```

You can find all available job managers by omitting any specific property information.

```
jm_set = findResource('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 = 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 client MATLAB 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.

```
jm = findResource('jobmanager','name','MyJobManager')
finished_jobs = findJob(jm,'State','finished')
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 you start a job manager, by default it starts so that it resumes its former session with all jobs intact. Alternatively, you can start a job manager 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" on page 2-14.

Programming Tips

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

Saving 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.

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.

Data Size Limit on Object Properties

The size of data that can be sent in any one setting of object properties is approximately 50 MB, due to the size of the heap allocated to the Java Virtual Machine (JVM) in a MATLAB session.

Interrupting a Job

Because jobs and tasks are run outside the client session, you cannot use **Ctrl+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`.

4

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 — Categorical List

This section contains descriptions of all the Distributed Computing Toolbox commands and functions.

General Functions	Distributed Computing Toolbox functions not specific to a particular object type
Job Manager Functions	Functions that operate on a job manager object
Job Functions	Functions that operate on a job object
Task Functions	Functions that operate on a task object

General Functions

`clear`	Remove objects from MATLAB workspace
`dctconfig`	Configure settings for Distributed Computing Toolbox client session
`dfeval`	Evaluate function using a cluster of computers
`dfevalasync`	Evaluate function asynchronously using a cluster of computer
`findResource`	Find available distributed computing resources
`get`	Return object properties
`getCurrentJob`	Get job object whose task is currently being evaluated by this worker session
`getCurrentJobmanager`	Get job manager object that sent current task to this worker session
`getCurrentTask`	Get task object currently being evaluated in this worker session
`getCurrentWorker`	Get worker object currently running this session
`help`	Display help for toolbox functions in Command Window
`inspect`	Open Property Inspector

`jobStartup`	Job startup M-file for user-defined options
`length`	Return length of object array
`methods`	List functions of object class
`set`	Configure or display object properties
`size`	Return size of object array
`taskFinish`	Task finish M-file for user-defined options
`taskStartup`	Task startup M-file for user-defined options

Job Manager Functions

`createJob`	Create job object
`findJob`	Find job objects stored in job queue
`pause`	Pause job manager queue
`resume`	Resume processing queue in job manager

Job Functions

`cancel`	Cancel a pending, queued, or running job
`createTask`	Create new task in job
`demote`	Demote job in job queue
`destroy`	Remove job object from a job manager and memory
`findTask`	Get task objects belonging to job object
`getAllOutputArguments`	Retrieve output arguments from all tasks evaluated in job object
`promote`	Promote job in job queue
`submit`	Queue job in job queue service
`waitForState`	Wait for job object to change state

Task Functions

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

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

cancel

Purpose

Cancel a pending or running task, or cancel a pending, queued, or running 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 tasks `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 job manager

Syntax

```
obj = createJob(jobmanager)
obj = createJob(..., 'p1', v1, 'p2', v2, ...)
```

Arguments

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

Description

`obj = createJob(jobmanager)` creates a job object at the specified remote location.

`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.

Example

Construct a job object.

```
jm = findResource('jobmanager');
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 `createTask`, `findJob`, `submit`

Purpose

Create new task in job

Syntax

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

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.

Example

Create a job object.

```
jm = findResource('jobmanager');
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`

Purpose

Configure settings for Distributed Computing Toolbox client session

Syntax

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

Arguments

p1	Property to configure.
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.

The only property supported in this release 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 an anonymous 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.

`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

Set the current client session port number to `21000`.

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

demote

Purpose

Demote job in job queue

Syntax

```
demote(obj)
```

Arguments

`obj`	Job object demoted in the job queue.

Description

`demote(obj)` demotes the job object `obj` that is queued in a job queue.

If `obj` is not the last job in the queue, `demote` exchanges the position of `obj` 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 from 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('jobmanager');
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 a cluster of computers

Syntax

```
[y1,...,ym] = dfeval(F, x1,...,xn)
[y1,...,ym] = dfeval(F, x1,...,xn, 'P1', V1, 'P2', V2,...)
```

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 job manager, task, or worker objects are not permitted.)
- `'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.

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 a cluster of computers

Syntax

```
Job = dfevalasync(F, numArgOut, x1,...,xn, 'P1', V1, 'P2', V2,...)
```

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 has been created and sent to the cluster.

Remarks When the job is finished, you can obtain the results associated with the job by executing the command

```
data = getAllOutputArguments(Job)
```

`data` is an `M`-by-`numArgOut` cell array, where `M` is the number of tasks.

See Also `dfeval`, `feval`

findJob

Purpose

Find job objects stored in job manager

Syntax

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

Arguments

`jm`	Job manager object in which to find the job.
`pending`	Array of jobs in job manager `jm` whose `State` is `pending`.
`queued`	Array of jobs in job manager `jm` whose `State` is `queued`.
`running`	Array of jobs in job manager `jm` whose `State` is `running`.
`finished`	Array of jobs in job manager `jm` whose `State` is `finished`.
`out`	Array of jobs found in job manager `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 job manager `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 job manager `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

findResource

Purpose Find available MATLAB Distributed Computing Engine resources

Syntax

```
out = findResource('type')
out = findResource('type','LookupURL','host:port', ...)
out = findResource('type', 'p1', v1, 'p2', v2,...)
```

Arguments

`'type'`	Type of resource to find: `'jobmanager'` or `'worker'`.
`out`	Object or array of objects returned.
`'LookupURL'`	Literal string to indicate usage of a remote lookup service.
`'host:port'`	Host (IP address or host name) and port of remote lookup service to use.
p1, *p2*	Object properties to match.
`v1`, `v2`	Values for corresponding object properties.

Description

`out = findResource('type')` returns an array, `out`, containing objects representing all available distributed computing resources of the given *type*. Acceptable types include `'jobmanager'` and `'worker'`.

`out = findResource('type','LookupURL','host:port')` uses the lookup service of the job manager running at a specific location. The lookup service is part of a job manager. By default, `findResource` uses all the lookup services that are available to the local machine via multicast. If you specify `'LookupURL'` with a host and port, `findResource` uses the lookup service of the job manager running at that location. This URL is not necessarily the host running the job manager or worker session that this call to `findResource` returns, it is only where the lookup is performed from. This unicast call is useful when you want to find resources that might not be available via multicast or in a network that doesn't support multicast. For more information about which ports these services use, see "Setting TCP Ports" on page 2-17.

`out = findResource('type','p1', v1, 'p2', v2,...)` returns an array, `out`, of resources of the given *type* 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 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 `MyJobManager`, then `findResource` will *not* find that object if searching for a `Name` property value of `myjobmanager`.

Remarks

Note that it is permissible to use parameter-value string pairs, structures, and parameter-value cell array pairs in the same call to `findResource`.

Example

Find particular job managers.

```
jm1 = findResource('jobmanager', 'Name', 'ClusterQueue1');
jm2 = findResource('jobmanager', 'Name', 'ClusterQueue2');
```

Find all job managers. In this example, there are four.

```
all_job_managers = findResource('jobmanager')
all_job_managers =
    distcomp.jobmanager: 1-by-4
```

Find all job managers accessible from the lookup service on a particular host.

```
jms = findResource('jobmanager','LookupURL','123.123.1.1:6789');
```

Find a particular job manager accessible from the lookup service on a particular host. In this example, `subnet2.host_alpha` port `6789` is where the lookup is performed, but the job manager named `SN2Jmgr` might be running on another machine.

```
jm = findResource('jobmanager', ...
'LookupURL', 'subnet2.host_alpha:6789', 'Name', 'SN2JMgr');
```

See Also

`findJob`, `findTask`

findTask

Purpose

Get 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('jobmanager');
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 Return 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 Retrieve 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

Because `getAllOutputArguments` results in a call to a remote service, it 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('jobmanager');
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 Get job object whose task is currently being evaluated by this worker session

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`

Purpose Get job manager object that sent current task to this worker session

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.

See Also `getCurrentJob`, `getCurrentTask`, `getCurrentWorker`

getCurrentTask

Purpose Get 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`

Purpose Get 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, 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('jobmanager','Name','MyJobManager')
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`

help

Purpose Display 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`.

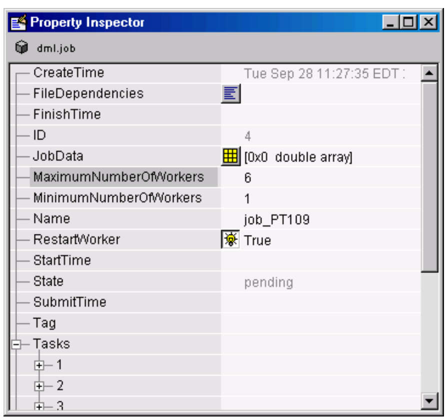
Remarks You can also open the Property Inspector via the Workspace browser by double-clicking an object.

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)
```



See Also `get`, `set`

jobStartup

Purpose

Job startup M-file for user-defined options

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`

Purpose Return 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`

methods

Purpose List functions of object class

Syntax

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

Arguments

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

Description

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

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

Example

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

```
jm = findResource('jobmanager','name','MyJobManager');
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`

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 be run. Jobs that are already running will continue to run. This call will do nothing if the job manager is already paused.

See Also `resume`, `waitForState`

promote

Purpose Promote job in job queue

Syntax

```
promote(obj)
```

Arguments

`obj`	Job object promoted in the queue.

Description `promote(obj)` promotes the job object `obj`, that is queued in a job queue.

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

See Also `createJob`, `demote`, `findJob`, `submit`

Purpose Resume processing queue in job manager

Syntax `resume(jm)`

Arguments

`jm`	Job manager object whose queue is resumed.

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

See Also `pause`, `waitForState`

Purpose

Configure or display object properties

Syntax

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

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.

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.

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 Return 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 job queue service

Syntax

```
submit(obj)
```

Arguments

`obj`	Job object to be queued.

Description `submit(obj)` queues the job object, `obj`, in the job manager resource. The resource where a job queue resides was determined when the job was created.

Remarks When a job contained in a job manager 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 by the job queue service.

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 a job manager service named `jobmanager1`.

```
jm1 = findResource('jobmanager', 'Name', 'jobmanager1');
```

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 Task finish M-file for user-defined options

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

Task startup M-file for user-defined options

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`

waitForState

Purpose

Wait for object to change state

Syntax

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

Arguments

`obj`	Job or task object whose change in state to wait for.
`'state'`	Value of the object's `State` property to wait for.
`timeout`	Maximum time to wait, in seconds.

Description

`waitForState(obj)` blocks execution in the client session until the job or task identified by the object `obj` reaches the `'finished'` state. 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 — Categorical List

This section contains descriptions of all toolbox properties.

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

Job Manager Properties

`BusyWorkers`	Indicate workers currently running tasks
`HostAddress`	Indicate IP address of host machine running job manager
`HostName`	Indicate name of host machine running job manager
`IdleWorkers`	Indicate which workers are idle and available to run tasks
`Jobs`	Indicate jobs contained in job manager
`Name`	Indicate name of job manager
`NumberOfBusyWorkers`	Indicate number of workers currently running tasks
`NumberOfIdleWorkers`	Indicate number of workers available to run tasks
`State`	Indicate current state of job manager

Job Properties

CreateTime	Indicate when job was created
FileDependencies	Indicate directories and files that worker can access
FinishedFcn	Specify callback to execute when job finishes running
FinishTime	Indicate when job finished
ID	Indicate object identifier
JobData	Indicate data made available to all workers for job's tasks
MaximumNumberOfWorkers	Specify maximum number of workers to perform tasks of a job
MinimumNumberOfWorkers	Specify minimum number of workers to perform tasks of a job
Name	Specify name for job object
Parent	Indicate parent job manager object of job
QueuedFcn	Specify M-file function to execute when job is added to queue
RestartWorker	Specify whether to restart MATLAB on worker before it evaluates task
RunningFcn	Specify M-file function to execute when job or task starts running
StartTime	Indicate when job started running
State	Indicate current state of job object
SubmitTime	Indicate when job was submitted to job queue
Tag	Specify label to associate with job object
Tasks	Indicate tasks contained in job object
Timeout	Specify time limit for completion of job

UserData	Specify data to associate with job object
UserName	Indicate user who created job

Task Properties

CaptureCommandWindowOutput	Specify whether to return command window output
CommandWindowOutput	Indicate text produced by execution of task object's function
CreateTime	Indicate when task was created
ErrorIdentifier	Indicate task error identifier
ErrorMessage	Indicate output message from task error
FinishedFcn	Specify callback to execute when task finishes running
FinishTime	Indicate when task finished
Function	Indicate function called when evaluating task
ID	Indicate object identifier
InputArguments	Indicate input arguments to task object
NumberOfOutputArguments	Indicate number of arguments returned by task function
OutputArguments	Data returned from the execution of task
Parent	Indicate parent job object of task
RunningFcn	Specify M-file function to execute when job or task starts running
State	Indicate current state of task object
StartTime	Indicate when task started running
Timeout	Specify time limit for completion of task
UserData	Specify data to associate with task object
Worker	Indicate worker session that performed task

Worker Properties

CurrentJob	Indicate job whose task the worker is currently running
CurrentTask	Indicate task that worker is currently running
HostAddress	Indicate IP address of host machine running worker session
HostName	Indicate name of host machine running worker session
Name	Indicate name of worker object
PreviousJob	Indicate job whose task the worker previously ran
PreviousTask	Indicate task that worker previously ran

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

Purpose Indicate 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 job manager.

```
jm = findResource('jobmanager', 'Name', 'MyJobManager');
workers_running_tasks = get(jm, 'BusyWorkers')
```

See Also

Properties

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

CaptureCommandWindowOutput

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('jobmanager', 'Name', 'MyJobManager');
j = createJob(jm);
createTask(j, @myfun, 1, {x});
createTask(j, @myfun, 1, {x});
.
.
.
alltasks = get(j, 'Tasks');
set(alltasks, 'CaptureCommandWindowOutput', true)
```

See Also

Properties

`Function`, `CommandWindowOutput`

Purpose

Indicate 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('jobmanager', 'Name', 'MyJobManager');
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`

CreateTime

Purpose Indicate 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 `CreateTime`.

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

Indicate job whose task the worker is currently running

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 Indicate 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`

ErrorIdentifier

Purpose Indicate 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 Indicate output 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('jobmanager', 'Name', 'MyJobManager')
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

Indicate 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 worker MATLAB 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 when task or job finishes 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 `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 Set the `FinishedFcn` property for a job and its task, using a function handle to an anonymous function that sends information to the display.

```
jm = findResource('jobmanager', 'Name',' MyJobManager');
j = createJob(jm, 'Name', 'Job_52a')

set(j, 'FinishedFcn', ...
    @(job,eventdata) disp([job.Name ' ' job.State]));
createTask(j, @rand, 1, {2,4}, ...
    'FinishedFcn', @(task,eventdata) disp('Task completed'));

submit(j)
Task completed
Task completed
Task completed
Job_52a finished
```

See Also

Properties

`QueuedFcn`, `RunningFcn`

FinishTime

Purpose

Indicate 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('jobmanager', 'Name',' MyJobManager');
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`

Purpose

Indicate 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`

HostAddress

Purpose

Indicate IP address of host machine running job manager or worker session

Description

HostAddress indicates the numerical IP address of the host machine 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	String

Example

Create a job manager object and examine its HostAddress property.

```
jm = findResource('jobmanager', 'Name', 'MyJobManager')
get(jm, 'HostAddress')
ans =
123.123.123.123
```

See Also

Functions

findResource

Properties

HostName

Purpose Indicate name of host machine 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('jobmanager', 'Name', 'MyJobManager')
get(jm, 'HostName')
ans =
JobMgrHost
```

See Also

Functions

findResource

Properties

HostAddress

ID

Purpose

Indicate 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('jobmanager', 'Name', 'MyJobManager')
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

Indicate which workers are idle and 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('jobmanager', 'Name', 'MyJobManager');
get(jm, 'NumberOfIdleWorkers')
```

See Also

Properties

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

InputArguments

Purpose

Indicate 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('jobmanager', 'Name', 'MyJobManager');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
get(t, 'InputArguments')
ans =
    [2]    [4]
```

See Also

Functions

`createTask`

Properties

`Function`, `OutputArguments`

Purpose

Indicate 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 doesn't have to be passed to the worker for each task in a job that the worker evaluates.

Note that to access the data in a job's JobData property, the worker session evaluating the task needs to have access to the job. Therefore, the job object must be passed to the task object as an input argument. See 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)

creatTask(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`.

See Also

Functions

`createJob`, `createTask`

Jobs

Purpose

Indicate jobs contained in job manager service

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
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('jobmanager', 'Name', 'MyJobManager');
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

Specify maximum number of workers to perform tasks of a job

Description

With `MaximumNumberOfWorkers` you specify the most number of workers to be used to perform the evaluation of the job's tasks. This property limits the portion of the cluster used for the job.

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('jobmanager', 'Name', 'MyJobManager');
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 tasks of a job

Description

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

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('jobmanager', 'Name', 'MyJobManager');
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 name for job object, or indicate name of job manager 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" on page 2-14. 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 manger 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
```

Name

See Also

Functions

`findResource`, `createJob`

Purpose Indicate 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('jobmanager', 'Name', 'MyJobManager');
get(jm, 'NumberOfBusyWorkers')
```

See Also

Properties

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

NumberOfIdleWorkers

Purpose

Indicate number of workers that are idle and 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('jobmanager', 'Name', 'MyJobManager');
get(jm, 'NumberOfIdleWorkers')
```

See Also

Properties

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

NumberOfOutputArguments

Purpose Indicate 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('jobmanager', 'Name', 'MyJobManager');
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`

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('jobmanager', 'Name', 'MyJobManager');
j = createJob(jm);
t = createTask(j, @rand, 1, {2, 4});
submit(j)
```

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

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

Retrieve the results from all the tasks of a job.

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

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

See Also

Functions

`createTask`, `getAllOutputArguments`

Properties

`Function`, `InputArguments`, `NumberOfOutputArguments`

Purpose Indicate parent object of job or task

Description A job's `Parent` property indicates the parent job manager object that contains the job. A task's `Parent` property indicates the parent job object that contains the task.

Characteristics

Usage	Job object or task object
Read-only	Always
Data type	Job manager object or job object

See Also

Properties

`Jobs`, `Tasks`

PreviousJob

Purpose Indicate job whose task the worker previously ran

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

Characteristics

Usage	Worker object
Read-only	Always
Data type	Job object

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

See Also

Properties

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

Purpose Indicate task that worker previously ran

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

Characteristics

Usage	Worker object
Read-only	Always
Data type	Task object

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

See Also

Properties

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

QueuedFcn

Purpose

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

Description

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

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

Characteristics

Usage	Job object
Read-only	Never
Data type	Callback

Values

`QueuedFcn` can be set to any valid MATLAB callback value.

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('jobmanager', 'Name', 'MyJobManager');
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 tasks in job

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, 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('jobmanager', 'Name', 'MyJobManager');
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('jobmanager', 'Name', 'MyJobManager');
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

Indicate when job or task started running

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('jobmanager', 'Name',' MyJobManager');
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

Indicate current state of task object, job object, 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.
- `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('jobmanager', 'Name', 'MyJobManager');
get(jm, 'State')
ans =
    running
j = createJob(jm);
get(j, 'State')
ans =
  pending
```

See Also

Functions

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

SubmitTime

Purpose Indicate when job was submitted to job 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('jobmanager', 'Name',' MyJobManager');
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`

Tasks

Purpose Indicate 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('jobmanager', 'Name', 'MyJobManager');
j = createJob(jm);
createTask(j, ...)
.
.
.
createTask(j, ...)
alltasks = get(j, 'Tasks')
alltasks =
     distcomp.task: 10-by-1
set(alltasks, 'Timeout', 20);
```

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

See Also

Functions

`createTask`, `destroy`, `findTask`

Properties

`Jobs`

Purpose

Specify time limit for completion of 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 practical limit of time you want to allow for evaluation of tasks and jobs.

Example

Set a job's `Timeout` value to 1 minute.

```
jm = findResource('jobmanager', 'Name', 'MyJobManager');
j = createJob(jm);
set(j, 'Timeout', 60)
```

See Also

Functions

`submit`

Properties

`ErrorMessage`, `State`

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 locally, not in a remote session. If you close the client session where `UserData` is set for an object, then later access the same object from another client by getting it from 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 Indicate 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

Indicate 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`

Index

P

Q

R