<table>
<thead>
<tr>
<th>Date</th>
<th>Revision Type</th>
<th>Version Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2004</td>
<td>Online only</td>
<td>New for Version 1.0 (Release 14SP1+)</td>
</tr>
<tr>
<td>March 2005</td>
<td>Online only</td>
<td>Revised for Version 1.0.1 (Release 14SP2)</td>
</tr>
<tr>
<td>September 2005</td>
<td>Online only</td>
<td>Revised for Version 1.0.2 (Release 14SP3)</td>
</tr>
<tr>
<td>November 2005</td>
<td>Online only</td>
<td>Revised for Version 2.0 (Release 14SP3+)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Revised for Version 2.0.1 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for Version 3.0 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for Version 3.1 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for Version 3.2 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for Version 3.3 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for Version 4.0 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for Version 4.1 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for Version 4.2 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for Version 4.3 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for Version 5.0 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for Version 5.1 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for Version 5.2 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for Version 6.0 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for Version 6.1 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for Version 6.2 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for Version 6.3 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for Version 6.4 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for Version 6.5 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for Version 6.6 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for Version 6.7 (Release 2015b)</td>
</tr>
<tr>
<td>March 2016</td>
<td>Online only</td>
<td>Revised for Version 6.8 (Release 2016a)</td>
</tr>
<tr>
<td>September 2016</td>
<td>Online only</td>
<td>Revised for Version 6.9 (Release 2016b)</td>
</tr>
<tr>
<td>March 2017</td>
<td>Online only</td>
<td>Revised for Version 6.10 (Release 2017a)</td>
</tr>
<tr>
<td>September 2017</td>
<td>Online only</td>
<td>Revised for Version 6.11 (Release 2017b)</td>
</tr>
<tr>
<td>March 2018</td>
<td>Online only</td>
<td>Revised for Version 6.12 (Release 2018a)</td>
</tr>
<tr>
<td>September 2018</td>
<td>Online only</td>
<td>Revised for Version 6.13 (Release 2018b)</td>
</tr>
</tbody>
</table>
Getting Started

1 Parallel Computing Toolbox Product Description ........................................ 1-2
   Key Features ............................................................................. 1-2

Parallel Computing Support in MathWorks Products ................................ 1-3

Create and Use Distributed Arrays ...................................................... 1-4
   Creating Distributed Arrays ...................................................... 1-4
   Creating Codistributed Arrays .................................................... 1-5

Determine Product Installation and Versions ......................................... 1-7

Interactively Run a Loop in Parallel Using parfor .................................. 1-8

Run Batch Parallel Jobs .................................................................... 1-11
   Run a Batch Job ...................................................................... 1-11
   Run a Batch Job with a Parallel Pool ........................................ 1-12

Run Script as Batch Job from the Current Folder Browser .................... 1-15

Distribute Arrays and Run SPMD ....................................................... 1-16
   Distributed Arrays .................................................................. 1-16
   Single Program Multiple Data (spmd) ....................................... 1-16
   Composites ............................................................................ 1-16

What Is Parallel Computing? .................................................................. 1-18

Choose a Parallel Computing Solution ............................................... 1-20

Run MATLAB Functions with Automatic Parallel Support ................... 1-24
   Find Automatic Parallel Support ............................................. 1-24

Run Non-Blocking Code in Parallel Using parfeval .............................. 1-26
Parallel for-Loops (parfor)

Decide When to Use parfor ................................ 2-2
  parfor-Loops in MATLAB ................................ 2-2
  Deciding When to Use parfor ............................ 2-2
  Example of parfor With Low Parallel Overhead ....... 2-3
  Example of parfor With High Parallel Overhead ...... 2-6

Convert for-Loops Into parfor-Loops ....................... 2-8

Ensure That parfor-Loop Iterations are Independent ...... 2-12

Nested parfor-Loops and for-Loops ........................ 2-15
  Parallelizing Nested Loops .............................. 2-15
  Convert Nested for-Loops to parfor ..................... 2-16
  Nested Loops: Requirements and Limitations .......... 2-19
  Nested Functions ...................................... 2-21
  Nested spmd Statements ................................ 2-22
  Break and Return Statements ............................ 2-22
  P-Code Scripts ....................................... 2-22

Scale Up parfor-Loops to Cluster and Cloud ............... 2-23

Use parfor-Loops for Reduction Assignments ............... 2-28

Use Objects and Handles in parfor-Loops .................. 2-30
  Using Objects in parfor-Loops ......................... 2-30
  Handle Classes ....................................... 2-30
  Sliced Variables Referencing Function Handles ....... 2-31

Troubleshoot Variables in parfor-Loops ..................... 2-32
  Ensure That parfor-Loop Variables Are Consecutive Increasing Integers ..................................... 2-32
  Avoid Overflows in parfor-Loops ....................... 2-32
  Solve Variable Classification Issues in parfor-Loops .... 2-33
  Structure Arrays in parfor-Loops ....................... 2-36
  Converting the Body of a parfor-Loop into a Function .... 2-37
Single Program Multiple Data (spmd)

Run Single Programs on Multiple Data Sets ........................................... 3-2
  Introduction ........................................................................... 3-2
  When to Use spmd ......................................................... 3-2
  Define an spmd Statement .................................................. 3-3
  Display Output ............................................................... 3-5
  MATLAB Path .................................................................... 3-5
  Error Handling ................................................................. 3-5
  spmd Limitations .............................................................. 3-6

Access Worker Variables with Composites ........................................... 3-8
  Introduction to Composites ............................................... 3-8
  Create Composites in spmd Statements .................................. 3-8
  Variable Persistence and Sequences of spmd ......................... 3-10
  Create Composites Outside spmd Statements ......................... 3-11

Distributing Arrays to Parallel Workers ................................................. 3-12
  Using Distributed Arrays to Partition Data Across Workers ........ 3-12
  Load Distributed Arrays in Parallel Using datastore ................. 3-12
  Alternative Methods for Creating Distributed and Codistributed
    Arrays ........................................................................... 3-15

Interactive Parallel Computation with pmode

pmode Versus spmd ................................................................. 4-2

Run Communicating Jobs Interactively Using pmode ......................... 4-3

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

Running pmode Interactive Jobs on a Cluster ..................................... 4-15

Plotting Distributed Data Using pmode ........................................... 4-16

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

How Parallel Computing Products Run a Job .......................... 6-2
  Overview ........................................ 6-2
  Toolbox and Server Components .................................. 6-3
  Life Cycle of a Job ..................................... 6-7

Program a Job on a Local Cluster .................................... 6-10

Specify Your Parallel Preferences ..................................... 6-12

Discover Clusters and Use Cluster Profiles ......................... 6-15
  Create and Manage Cluster Profiles ................................ 6-16
  Discover Clusters .................................... 6-17
  Create Cloud Cluster ..................................... 6-19
  Add and Modify Cluster Profiles .................................. 6-20
  Import and Export Cluster Profiles ................................. 6-24
  Edit Number of Workers and Cluster Settings .................... 6-25
  Use Your Cluster from MATLAB .................................... 6-26

Apply Callbacks to MJS Jobs and Tasks ................................. 6-28

Job Monitor .................................................. 6-32
  Typical Use Cases ........................................ 6-32
  Manage Jobs Using the Job Monitor ................................ 6-33
  Identify Task Errors Using the Job Monitor ....................... 6-33

Programming Tips .................................................. 6-35
  Program Development Guidelines .................................. 6-35
  Current Working Directory of a MATLAB Worker .................. 6-36
  Writing to Files from Workers ................................... 6-37
  Saving or Sending Objects ...................................... 6-37
  Using clear functions ........................................ 6-37
  Running Tasks That Call Simulink Software ....................... 6-38
  Using the pause Function ..................................... 6-38
  Transmitting Large Amounts of Data ................................ 6-38
  Interrupting a Job ......................................... 6-38
  Speeding Up a Job .......................................... 6-38

Control Random Number Streams ...................................... 6-40
  Different Workers .......................................... 6-40
Client and Workers ........................................... 6-41
Client and GPU .............................................. 6-42
Worker CPU and Worker GPU .............................. 6-44

Profiling Parallel Code ........................................ 6-45
  Introduction ................................................ 6-45
  Collecting Parallel Profile Data ......................... 6-45
  Viewing Parallel Profile Data ............................. 6-46

HPC Challenge Benchmarks .................................... 6-54

Troubleshooting and Debugging .............................. 6-55
  Attached Files Size Limitations ......................... 6-55
  File Access and Permissions ............................. 6-55
  No Results or Failed Job ................................. 6-57
  Connection Problems Between the Client and MJS ...... 6-57
  SFTP Error: Received Message Too Long ................ 6-58

Big Data Workflow Using Tall Arrays and Datastores ...... 6-60
  Running Tall Arrays in Parallel ......................... 6-61
  Use mapreduce to Control Where Your Code Runs ....... 6-62

Use Tall Arrays on a Parallel Pool .......................... 6-63

Use Tall Arrays on a Spark Enabled Hadoop Cluster ...... 6-67
  Creating and Using Tall Tables .......................... 6-67

Run mapreduce on a Parallel Pool .......................... 6-71
  Start Parallel Pool ..................................... 6-71
  Compare Parallel mapreduce .............................. 6-71

Run mapreduce on a Hadoop Cluster .......................... 6-75
  Cluster Preparation ..................................... 6-75
  Output Format and Order ................................ 6-75
  Calculate Mean Delay ................................... 6-75

Partition a Datastore in Parallel .......................... 6-78
Further Notes on Communicating Jobs ............................................. 8-10
  Number of Tasks in a Communicating Job ..................................... 8-10
  Avoid Deadlock and Other Dependency Errors ................................ 8-10

GPU Computing

9

GPU Capabilities and Performance ........................................... 9-2
  Capabilities ............................................................................. 9-2
  Performance Benchmarking ....................................................... 9-2

Establish Arrays on a GPU ..................................................... 9-3
  Create GPU Arrays from Existing Data ...................................... 9-3
  Create GPU Arrays Directly ..................................................... 9-4
  Examine gpuArray Characteristics ......................................... 9-6

Run MATLAB Functions on a GPU ........................................ 9-8
  MATLAB Functions with gpuArray Arguments .............................. 9-8
  Check or Select a GPU ............................................................ 9-8
  Use MATLAB Functions with a GPU ........................................ 9-9
  Sharpen an Image Using the GPU ............................................. 9-11
  Compute the Mandelbrot Set using GPU-Enabled Functions ............ 9-11
  Work with Sparse Arrays on a GPU ......................................... 9-13
  Work with Complex Numbers on a GPU .................................... 9-15
  Special Conditions for gpuArray Inputs .................................... 9-16
  Acknowledgments ..................................................................... 9-16

Run Element-wise MATLAB Code on GPU ............................... 9-18
  MATLAB Code vs. gpuArray Objects ....................................... 9-18
  Run Your MATLAB Functions on a GPU ................................. 9-18
  Example: Run Your MATLAB Code ........................................ 9-19
  Supported MATLAB Code ....................................................... 9-19

Identify and Select a GPU Device ............................................ 9-23

Run CUDA or PTX Code on GPU ............................................. 9-25
  Overview ................................................................................. 9-25
  Create a CUDAKernel Object .................................................. 9-26
  Run a CUDAKernel .................................................................. 9-31
Complete Kernel Workflow ................................. 9-33

**Run MEX-Functions Containing CUDA Code** .................. 9-36
  Write a MEX-File Containing CUDA Code ...................... 9-36
  Run the Resulting MEX-Functions ............................ 9-37
  Comparison to a CUDA Kernel ............................... 9-37
  Access Complex Data ...................................... 9-38
  Compile a GPU MEX-File ................................. 9-38

**Measure and Improve GPU Performance** ....................... 9-40
  Getting Started with GPU Benchmarking ..................... 9-40
  Improve Performance Using Single Precision Calculations 9-40
  Basic Workflow for Improving Performance .................. 9-41
  Advanced Tools for Improving Performance .................. 9-42
  Best Practices for Improving Performance .................. 9-43
  Measure Performance on the GPU ............................ 9-44
  Vectorize for Improved GPU Performance .................... 9-45
  Troubleshooting GPUs .................................... 9-48

**GPU Support by Release** .................................. 9-49
  Increase the CUDA Cache Size ............................. 9-50

**Objects — Alphabetical List**

**Functions — Alphabetical List**

**Glossary**
Getting Started

• “Parallel Computing Toolbox Product Description” on page 1-2
• “Parallel Computing Support in MathWorks Products” on page 1-3
• “Create and Use Distributed Arrays” on page 1-4
• “Determine Product Installation and Versions” on page 1-7
• “Interactively Run a Loop in Parallel Using parfor” on page 1-8
• “Run Batch Parallel Jobs” on page 1-11
• “Run Script as Batch Job from the Current Folder Browser” on page 1-15
• “Distribute Arrays and Run SPMD” on page 1-16
• “What Is Parallel Computing?” on page 1-18
• “Choose a Parallel Computing Solution” on page 1-20
• “Run MATLAB Functions with Automatic Parallel Support” on page 1-24
• “Run Non-Blocking Code in Parallel Using parfeval” on page 1-26
• “Evaluate Functions in the Background Using parfeval” on page 1-27
Parallel Computing Toolbox Product Description

Perform parallel computations on multicore computers, GPUs, and computer clusters

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters. High-level constructs—parallel for-loops, special array types, and parallelized numerical algorithms—let you parallelize MATLAB® applications without CUDA or MPI programming. You can use the toolbox with Simulink® to run multiple simulations of a model in parallel.

The toolbox lets you use the full processing power of multicore desktops by executing applications on workers (MATLAB computational engines) that run locally. Without changing the code, you can run the same applications on a computer cluster or a grid computing service (using MATLAB Distributed Computing Server™). You can run parallel applications interactively or in batch.

Key Features

- Parallel for-loops (parfor) for running task-parallel algorithms on multiple processors
- Support for CUDA enabled NVIDIA GPUs
- Full use of multicore processors on the desktop via workers that run locally
- Computer cluster and grid support (with MATLAB Distributed Computing Server)
- Interactive and batch execution of parallel applications
- Distributed arrays and spmd (single-program-multiple-data) for large dataset handling and data-parallel algorithms
Parallel Computing Support in MathWorks Products

Parallel Computing Toolbox provides you with tools for a local cluster of workers on your client machine. MATLAB Distributed Computing Server software allows you to run as many MATLAB workers on a remote cluster of computers as your licensing allows.

Most MathWorks products enable you to run applications in parallel. For example, Simulink models can run simultaneously in parallel, as described in “Run Multiple Simulations” (Simulink). MATLAB Compiler™ and MATLAB Compiler SDK™ software let you build and deploy parallel applications; for example, see the “Parallel Computing” section of MATLAB Compiler “Standalone Applications” (MATLAB Compiler).

Several MathWorks products now offer built-in support for the parallel computing products, without requiring extra coding. For the current list of these products and their parallel functionality, see:

Create and Use Distributed Arrays

If your data is currently in the memory of your local machine, you can use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool. Distributed arrays use the combined memory of multiple workers in a parallel pool to store the elements of an array. For alternative ways of partitioning data, see “Distributing Arrays to Parallel Workers” on page 3-12. You can use distributed arrays to scale up your big data computation. Consider distributed arrays when you have access to a cluster, as you can combine the memory of multiple machines in your cluster.

A distributed array is a single variable, split over multiple workers in your parallel pool. You can work with this variable as one single entity, without having to worry about its distributed nature. Explore the functionalities available for distributed arrays in the Parallel Computing Toolbox: “Run MATLAB Functions with Distributed Arrays” on page 5-24.

When you create a distributed array, you cannot control the details of the distribution. On the other hand, codistributed arrays allow you to control all aspects of distribution, including dimensions and partitions. In the following, you learn how to create both distributed and codistributed arrays.

Creating Distributed Arrays

You can create a distributed array in different ways:

- Use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool.
- You can directly construct a distributed array on the workers. You do not need to first create the array in the client, so that client workspace memory requirements are reduced. The functions available include `eye(___,'distributed')`, `rand(___,'distributed')`, etc. For a full list, see the distributed object reference page.
• Create a codistributed array inside an spmd statement, see “Single Program Multiple Data (spmd)” on page 1-16. Then access it as a distributed array outside the spmd statement. This lets you use distribution schemes other than the default.

In this example, you create an array in the client workspace, then turn it into a distributed array:

```
parpool('local',4)  % Create pool
A = magic(4);       % Create magic 4-by-4 matrix
B = distributed(A); % Distribute to the workers
B                   % View results in client.
whos                % B is a distributed array here.
delete(gcp)         % Stop pool
```

You have created B as a distributed array, split over the workers in your parallel pool. This is shown in the figure.

![Array Split Over Workers](image)

### Creating Codistributed Arrays

Unlike distributed arrays, codistributed arrays allow you to control all aspects of distribution, including dimensions and partitions. You can create a codistributed array in different ways:

- “Partitioning a Larger Array” on page 5-7 — Start with a large array that is replicated on all workers, and partition it so that the pieces are distributed across the workers. This is most useful when you have sufficient memory to store the initial replicated array.
- “Building from Smaller Arrays” on page 5-8 — Start with smaller replicated arrays stored on each worker, and combine them so that each array becomes a segment of a
larger codistributed array. This method reduces memory requirements as it lets you build a codistributed array from smaller pieces.

- “Using MATLAB Constructor Functions” on page 5-9 — Use any of the MATLAB constructor functions like `rand` or `zeros` with a codistributor object argument. These functions offer a quick means of constructing a codistributed array of any size in just one step.

In this example, you create a codistributed array inside an `spmd` statement, using a nondefault distribution scheme. First, define 1-D distribution along the third dimension, with 4 parts on worker 1, and 12 parts on worker 2. Then create a 3-by-3-by-16 array of zeros.

```matlab
parpool('local',2) % Create pool
spmd
codist = codistributor1d(3,[4,12]);
Z = zeros(3,3,16,codist);
Z = Z + labindex;
end
Z           % View results in client.
whos        % Z is a distributed array here.
delete(gcp) % Stop pool
```

For more details on codistributed arrays, see “Working with Codistributed Arrays” on page 5-5.

### See Also

### Related Examples

- “Distributing Arrays to Parallel Workers” on page 3-12
- “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
- “Single Program Multiple Data (spmd)” on page 1-16
Determine Product Installation and Versions

To determine if Parallel Computing Toolbox software is installed on your system, type this command at the MATLAB prompt.

ver

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

If you want to run your applications on a cluster, see your system administrator to verify that the version of Parallel Computing Toolbox you are using is the same as the version of MATLAB Distributed Computing Server installed on your cluster.
Interactively Run a Loop in Parallel Using `parfor`

In this example, you start with a slow `for`-loop, and you speed up the calculation using a `parfor`-loop instead. `parfor` splits the execution of `for`-loop iterations over the workers in a parallel pool.

This example calculates the spectral radius of a matrix and converts a `for`-loop into a `parfor`-loop. Find out how to measure the resulting speedup.

1. In the MATLAB Editor, enter the following `for`-loop. Add `tic` and `toc` to measure the time elapsed.

   ```matlab
tic
   n = 200;
   A = 500;
   a = zeros(n);
   for i = 1:n
       a(i) = max(abs(eig(rand(A))));
   end
   toc
   ```

2. Run the script, and note the elapsed time.

   ```matlab
   Elapsed time is 31.935373 seconds.
   ```

3. In the script, replace the `for`-loop with a `parfor`-loop.

   ```matlab
tic
   n = 200;
   A = 500;
   ```
a = zeros(n);  
parfor i = 1:n  
a(i) = max(abs(eig(rand(A))));  
end  
toc  

4 Run the new script, and run it again. Note that the first run is slower than the second run, because the parallel pool takes some time to start and make the code available to the workers. Note the elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers...
Elapsed time is 10.760068 seconds.

The parfor run on four workers is about three times faster than the corresponding for-loop run. The speed-up is smaller than the ideal speed-up of a factor of four on four workers. This is due to parallel overhead, including the time required to transfer data from the client to the workers and back. This example shows a good speed-up with relatively small parallel overhead, and benefits from conversion into a parfor-loop. Not all for-loop iterations can be turned into faster parfor-loops. To learn more, see “Decide When to Use parfor” on page 2-2.

One key requirement for using parfor-loops is that the individual iterations must be independent. Independent problems suitable for parfor processing include Monte Carlo simulations and parameter sweeps. For next steps, see “Convert for-Loops Into parfor-Loops” on page 2-8.

In this example, you managed to speed up the calculation by converting the for-loop into a parfor-loop on four workers. You might reduce the elapsed time further by increasing the number of workers in your parallel pool, see “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23.

You can modify your cluster profiles to control how many workers run your loops, and whether the workers are local or on a cluster. For more information on profiles, see “Discover Clusters and Use Cluster Profiles” on page 6-15.

Modify your parallel preferences to control whether a parallel pool is created automatically, and how long it remains available before timing out. For more information on preferences, see “Specify Your Parallel Preferences” on page 6-12.
You can run Simulink models in parallel with the `parsim` command instead of using `parfor`-loops. For more information and examples of using Simulink in parallel, see “Run Multiple Simulations” (Simulink).

**See Also**

`parfor` | `parpool` | `tic` | `toc`

**More About**

- “Decide When to Use `parfor`” on page 2-2
- “Convert for-Loops Into `parfor`-Loops” on page 2-8
- “Scale Up `parfor`-Loops to Cluster and Cloud” on page 2-23
Run Batch Parallel Jobs

Run a Batch Job

To offload work from your MATLAB session to run in the background in another session, you can use the batch command inside a script.

1. To create the script, type:
   ```
   edit mywave
   ```

2. In the MATLAB Editor, create a `for`-loop:
   ```
   for i = 1:1024
       A(i) = sin(i*2*pi/1024);
   end
   ```

3. Save the file and close the Editor.

4. Use the `batch` command in the MATLAB Command Window to run your script on a separate MATLAB worker:
   ```
   job = batch('mywave')
   ```

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

6. The `load` command transfers variables created on the worker to the client workspace, where you can view the results:
   ```
   load(job,'A')
   plot(A)
   ```

7. When the job is complete, permanently delete its data and remove its reference from the workspace:
   ```
   delete(job)
   clear job
   ```
batch runs your code on a local worker or a cluster worker, but does not require a parallel pool.

You can use batch to run either scripts or functions. For more details, see the batch reference page.

**Run a Batch Job with a Parallel Pool**

You can combine the abilities to offload a job and run a loop in a parallel pool. This example combines the two to create a simple batch `parfor`-loop.

1. To create a script, type:
   
   ```
   edit mywave
   ```

2. In the MATLAB Editor, create a `parfor`-loop:
   
   ```
   parfor i = 1:1024
   A(i) = sin(i*2*pi/1024);
   end
   ```

3. Save the file and close the Editor.

4. Run the script in MATLAB with the `batch` command. Indicate that the script should use a parallel pool for the loop:

   ```
   job = batch('mywave','Pool',3)
   ```

   This command specifies that three workers (in addition to the one running the batch script) are to evaluate the loop iterations. Therefore, this example uses a total of four local workers, including the one worker running the batch script. Altogether, there are five MATLAB sessions involved, as shown in the following diagram.
To view the results:

```matlab
wait(job)
load(job,'A')
plot(A)
```

The results look the same as before, however, there are two important differences in execution:

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

When the job is complete, permanently delete its data and remove its reference from the workspace:

```matlab
delete(job)
clear job
```
See Also

batch
Run Script as Batch Job from the Current Folder Browser

From the Current Folder browser, you can run a MATLAB script as a batch job by browsing to the file’s folder, right-clicking the file, and selecting Run Script as Batch Job. The batch job runs on the cluster identified by the default cluster profile. The following figure shows the menu option to run the script file script1.m:

Running a script as a batch from the browser uses only one worker from the cluster. So even if the script contains a parfor loop or spmd block, it does not open an additional pool of workers on the cluster. These code blocks execute on the single worker used for the batch job. If your batch script requires opening an additional pool of workers, you can run it from the command line, as described in “Run a Batch Job with a Parallel Pool” on page 1-12.

When you run a batch job from the browser, this also opens the Job Monitor. The Job Monitor is a tool that lets you track your job in the scheduler queue. For more information about the Job Monitor and its capabilities, see “Job Monitor” on page 6-32.
Distribute Arrays and Run SPMD

Distributed Arrays

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

Use the distributed function to distribute an array among the workers:

```matlab
M = magic(4) % a 4-by-4 magic square in the client workspace
MM = distributed(M)
```

Now MM is a distributed array, equivalent to M, and you can manipulate or access its elements in the same way as any other array.

```matlab
M2 = 2*MM; % M2 is also distributed, calculation performed on workers
x = M2(1,1) % x on the client is set to first element of M2
```

Single Program Multiple Data (spmd)

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

```matlab
spmd     % By default creates pool and uses all workers
    R = rand(4);
end
```

This code creates an individual 4-by-4 matrix, R, of random numbers on each worker in the pool.

Composites

Following an spmd statement, in the client context, the values from the block are accessible, even though the data is actually stored on the workers. On the client, these variables are called Composite objects. Each element of a composite is a symbol referencing the value (data) on a worker in the pool. Note that because a variable might not be defined on every worker, a Composite might have undefined elements.
Continuing with the example from above, on the client, the Composite R has one element for each worker:

\[ X = R\{3\}; \quad \text{% Set } X \text{ to the value of } R \text{ from worker 3.} \]

The line above retrieves the data from worker 3 to assign the value of \( X \). The following code sends data to worker 3:

\[ X = X + 2; \]
\[ R\{3\} = X; \quad \text{% Send the value of } X \text{ from the client to worker 3.} \]

If the parallel pool remains open between \texttt{spmd} statements and the same workers are used, the data on each worker persists from one \texttt{spmd} statement to another.

\texttt{spmd}
\begin{verbatim}
R = R + labindex  \% Use values of R from previous spmd.
end
\end{verbatim}

A typical use for \texttt{spmd} is to run the same code on a number of workers, each of which accesses a different set of data. For example:

\texttt{spmd}
\begin{verbatim}
INP = load(['somedatafile' num2str(labindex) '.mat']);
RES = somefun(INP)
end
\end{verbatim}

Then the values of \( RES \) on the workers are accessible from the client as \( RES\{1\} \) from worker 1, \( RES\{2\} \) from worker 2, etc.

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

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

Although data persists on the workers from one \texttt{spmd} block to another as long as the parallel pool remains open, data does not persist from one instance of a parallel pool to another. That is, if the pool is deleted and a new one created, all data from the first pool is lost.

For more information about using distributed arrays, \texttt{spmd}, and Composites, see “Distributed Arrays”.

1-17
What Is Parallel Computing?

Parallel computing allows you to carry out many calculations simultaneously. Large problems can often be split into smaller ones, which are then solved at the same time.

The main reasons to consider parallel computing are to

- Save time by distributing tasks and executing these simultaneously
- Solve big data problems by distributing data
- Take advantage of your desktop computer resources and scale up to clusters and cloud computing

With Parallel Computing Toolbox, you can

- Accelerate your code using interactive parallel computing tools, such as `parfor` and `parfeval`
- Scale up your computation using interactive Big Data processing tools, such as `distributed`, `tall`, `datastore`, and `mapreduce`
- Use `gpuArray` to speed up your calculation on the GPU of your computer
- Use `batch` to offload your calculation to computer clusters or cloud computing facilities

Here are some useful Parallel Computing concepts:

- **CPU**: Central Processing Unit, comprising multiple cores or processors
- **GPU**: Graphics Processing Unit, now widely used for general purpose (GP) GPU computing
- **Node**: standalone computer, containing one or more CPUs / GPUs. Nodes are networked to form a cluster or supercomputer
- **Thread**: smallest set of instructions that can be managed independently by a scheduler. On a GPU, multiprocessor or multicore system, multiple threads can be executed simultaneously (multi-threading)
- **Batch**: off-load execution of a functional script to run in the background
- **Scalability**: increase in parallel speedup with the addition of more resources

What tools do MATLAB and Parallel Computing Toolbox offer?
• MATLAB workers: MATLAB computational engines for parallel computing, associated with the cores in a multicore machine
• Parallel pool: a parallel pool of MATLAB workers can be created using parpool
• Speed up: Accelerate your code by running on multiple MATLAB workers, using parfor and parfeval
• Scale up: Partition your big data across multiple MATLAB workers, using distributed arrays and mapreduce
• Asynchronous processing: Use parfeval to execute a computing task without waiting for it to complete

See Also

Related Examples
• “Choose a Parallel Computing Solution” on page 1-20
• “Identify and Select a GPU Device” on page 9-23
• “Decide When to Use parfor” on page 2-2
• “Run Single Programs on Multiple Data Sets” on page 3-2
• “Evaluate Functions in the Background Using parfeval” on page 1-27
• “Distributing Arrays to Parallel Workers” on page 3-12
• “Run Batch Parallel Jobs” on page 1-11
Choose a Parallel Computing Solution

Process your data faster or scale up your big data computation using the capabilities of MATLAB, Parallel Computing Toolbox and MATLAB Distributed Computing Server.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solutions</th>
<th>Required Products</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you want to process your data faster?</td>
<td>Profile your code.</td>
<td>MATLAB</td>
<td>“Profile to Improve Performance” (MATLAB)</td>
</tr>
<tr>
<td></td>
<td>Vectorize your code.</td>
<td>MATLAB</td>
<td>“Vectorization” (MATLAB)</td>
</tr>
<tr>
<td></td>
<td>Use automatic parallel computing support in MathWorks products.</td>
<td>MATLAB, Parallel Computing Toolbox</td>
<td>“Run MATLAB Functions with Automatic Parallel Support” on page 1-24</td>
</tr>
<tr>
<td>If you have a GPU, try gpuArray.</td>
<td></td>
<td>MATLAB, Parallel Computing Toolbox</td>
<td>“Run MATLAB Functions on a GPU” on page 9-8</td>
</tr>
<tr>
<td></td>
<td>Use parfor.</td>
<td>MATLAB, Parallel Computing Toolbox</td>
<td>“Interactively Run a Loop in Parallel Using parfor” on page 1-8</td>
</tr>
<tr>
<td>Are you looking for other ways to speed up your processing?</td>
<td>Try parfeval.</td>
<td>MATLAB, Parallel Computing Toolbox</td>
<td>“Evaluate Functions in the Background Using parfeval” on page 1-27</td>
</tr>
<tr>
<td></td>
<td>Try spmd.</td>
<td>MATLAB, Parallel Computing Toolbox</td>
<td>“Run Single Programs on Multiple Data Sets” on page 3-2</td>
</tr>
<tr>
<td>Problem</td>
<td>Solutions</td>
<td>Required Products</td>
<td>More Information</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Do you want to scale up your big data calculation?</td>
<td>To work with out-of-memory data with any number of rows, use tall arrays. This workflow is well suited to data analytics and machine learning.</td>
<td>MATLAB</td>
<td>“Big Data Workflow Using Tall Arrays and Datastores” on page 6-60</td>
</tr>
<tr>
<td></td>
<td>Use tall arrays in parallel on your local machine.</td>
<td>MATLAB Parallel Computing Toolbox</td>
<td>“Use Tall Arrays on a Parallel Pool” on page 6-63</td>
</tr>
<tr>
<td></td>
<td>Use tall arrays in parallel on your cluster.</td>
<td>MATLAB Parallel Computing Toolbox MATLAB Distributed Computing Server</td>
<td>“Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67</td>
</tr>
<tr>
<td></td>
<td>If your data is large in multiple dimensions, use distributed instead. This workflow is well suited to linear algebra problems.</td>
<td>MATLAB Parallel Computing Toolbox MATLAB Distributed Computing Server</td>
<td>“Run MATLAB Functions with Distributed Arrays” on page 5-24</td>
</tr>
<tr>
<td></td>
<td>Do you want to offload to a cluster?</td>
<td>Use batch to run your code on clusters and clouds.</td>
<td>MATLAB Distributed Computing Server</td>
</tr>
</tbody>
</table>
1 - Getting Started

FASTER PROCESSING?

Do you want to process your data faster or do you have big data?

Profile it
Vectorize it
Use built-in support

BIG DATA?

Does your data fit in the memory of your local machine or cluster?

YES

NO

Do you have a GPU?

YES
gpuArray

distributed arrays

tall arrays

Are you looking for other ways to speed up your processing?

YES

parfor
parfeval
spmd

Do you want to offload to cluster or cloud?

YES

batch

Finish
See Also

Related Examples

- “Profile to Improve Performance” (MATLAB)
- “Vectorization” (MATLAB)
- Built-in Parallel Computing Support
- “Identify and Select a GPU Device” on page 9-23
- “Interactively Run a Loop in Parallel Using parfor” on page 1-8
- “Evaluate Functions in the Background Using parfeval” on page 1-27
- “Run Single Programs on Multiple Data Sets” on page 3-2
- “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
- “Use Tall Arrays on a Parallel Pool” on page 6-63
- “Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
- “Distributing Arrays to Parallel Workers” on page 3-12
- “Run Batch Parallel Jobs” on page 1-11
Run MATLAB Functions with Automatic Parallel Support

A growing number of functions, System object™, and features in several MATLAB and Simulink products help you take advantage of parallel computing resources without requiring any extra coding. You can enable this support by simply setting a flag or preference. To take advantage of this functionality on your desktop, you need Parallel Computing Toolbox. Run calculations in parallel using local workers to speed up large calculations. To scale the parallel computing support to larger resources such as computer clusters, you also need MATLAB Distributed Computing Server.

- Some functions run automatically in parallel by default. For example, parfor, parsim, and tall.
- Many other functions run automatically in parallel if you set an option to use parallel.

When you run a function with parallel enabled, MATLAB automatically opens a parallel pool of workers. MATLAB runs the computation across the available workers.

Automatic parallel support starts a parallel pool of workers using the default cluster profile. If you have not touched your parallel preferences, the default profile is local. Control parallel behaviour with the parallel preferences, including scaling up to a cluster, automatic pool creation, and preferred number of workers.

Find Automatic Parallel Support

- On function pages, find information under Extended Capabilities.
- To browse supported functions by product, click the Functions tab, select a product, and select the check box Automatic Parallel Support.

If a function you are interested in does not include automatic parallel support, there are alternatives:

- If you have a GPU, many MATLAB functions can run automatically on a GPU. See “Run MATLAB Functions on a GPU” on page 9-8.
- Any MATLAB code inside a for loop can be made into a parallel for loop, provided the iterations are independent. See parfor.
- If you are you looking for other ways to speed up your processing or to scale up your big data calculation, see “Choose a Parallel Computing Solution” on page 1-20.
See Also

Related Examples
• “Specify Your Parallel Preferences” on page 6-12
• “Run Code on Parallel Pools” on page 2-67
• “Scale up from Desktop to Cluster”

More About
• “Run MATLAB Functions on a GPU” on page 9-8
• “Parallel for-Loops (parfor)”
• “Choose a Parallel Computing Solution” on page 1-20
Run Non-Blocking Code in Parallel Using parfeval

You can execute a function on one or all parallel pool workers, without waiting for it to complete, using parfeval or parfevalOnAll. This can be useful if you want to be able to plot intermediate results. In addition, parfeval allows you to break out of a loop early, if you have established that your results are good enough. This may be convenient in e.g. optimization procedures. Note that this is different from using parfor, where you have to wait for the loop to complete.
Evaluate Functions in the Background Using parfeval

This example shows how you can use parfeval to evaluate a function in the background and to collect results as they become available. In this example, you submit a vector of multiple future requests in a for-loop and retrieve the individual future outputs as they become available.

```matlab
p = gcp();
% To request multiple evaluations, use a loop.
for idx = 1:10
    f(idx) = parfeval(p,@magic,1,idx); % Square size determined by idx
end
% Collect the results as they become available.
magicResults = cell(1,10);
for idx = 1:10
    % fetchNext blocks until next results are available.
    [completedIdx,value] = fetchNext(f);
    magicResults{completedIdx} = value;
    fprintf('Got result with index: %d.
', completedIdx);
end

Got result with index: 1.
Got result with index: 2.
Got result with index: 3.
Got result with index: 4.
Got result with index: 5.
Got result with index: 6.
Got result with index: 7.
Got result with index: 8.
Got result with index: 9.
Got result with index: 10.
```

See Also

Related Examples

- “Parfeval Blackjack”
Parallel for-Loops (parfor)

- “Decide When to Use parfor” on page 2-2
- “Convert for-Loops Into parfor-Loops” on page 2-8
- “Ensure That parfor-Loop Iterations are Independent” on page 2-12
- “Nested parfor-Loops and for-Loops” on page 2-15
- “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23
- “Use parfor-Loops for Reduction Assignments” on page 2-28
- “Use Objects and Handles in parfor-Loops” on page 2-30
- “Troubleshoot Variables in parfor-Loops” on page 2-32
- “Loop Variables” on page 2-39
- “Sliced Variables” on page 2-41
- “Broadcast Variables” on page 2-46
- “Reduction Variables” on page 2-47
- “Temporary Variables” on page 2-55
- “Ensure Transparency in parfor-Loops” on page 2-58
- “Improve parfor Performance” on page 2-61
- “Run Code on Parallel Pools” on page 2-67
- “Repeat Random Numbers in parfor-Loops” on page 2-73
Decide When to Use parfor

In this section...

“parfor-Loops in MATLAB” on page 2-2
“Deciding When to Use parfor” on page 2-2
“Example of parfor With Low Parallel Overhead” on page 2-3
“Example of parfor With High Parallel Overhead” on page 2-6

parfor-Loops in MATLAB

A parfor-loop in MATLAB executes a series of statements in the loop body in parallel. The MATLAB client issues the parfor command and coordinates with MATLAB workers to execute the loop iterations in parallel on the workers in a parallel pool. The client sends the necessary data on which parfor operates to workers, where most of the computation is executed. The results are sent back to the client and assembled.

A parfor-loop can provide significantly better performance than its analogous for-loop, because several MATLAB workers can compute simultaneously on the same loop.

Each execution of the body of a parfor-loop is an iteration. MATLAB workers evaluate iterations in no particular order and independently of each other. Because each iteration is independent, there is no guarantee that the iterations are synchronized in any way, nor is there any need for this. If the number of workers is equal to the number of loop iterations, each worker performs one iteration of the loop. If there are more iterations than workers, some workers perform more than one loop iteration; in this case, a worker might receive multiple iterations at once to reduce communication time.

Deciding When to Use parfor

A parfor-loop can be useful if you have a slow for-loop. Consider parfor if you have:

- Some loop iterations that take a long time to execute. In this case, the workers can execute the long iterations simultaneously. Make sure that the number of iterations exceeds the number of workers. Otherwise, you will not use all workers available.
- Many loop iterations of a simple calculation, such as a Monte Carlo simulation or a parameter sweep. parfor divides the loop iterations into groups so that each worker executes some portion of the total number of iterations.
A `parfor`-loop might not be useful if you have:

- Code that has vectorized out the `for`-loops. Generally, if you want to make code run faster, first try to vectorize it. For details how to do this, see “Vectorization” (MATLAB). Vectorizing code allows you to benefit from the built-in parallelism provided by the multithreaded nature of many of the underlying MATLAB libraries. However, if you have vectorized code and you have access only to `local` workers, then `parfor`-loops may run slower than `for`-loops. Do not devectorize code to allow for `parfor`; in general, this solution does not work well.

- Loop iterations that take a short time to execute. In this case, parallel overhead dominates your calculation.

You cannot use a `parfor`-loop when an iteration in your loop depends on the results of other iterations. Each iteration must be independent of all others. For help dealing with independent loops, see “Ensure That parfor-Loop Iterations are Independent” on page 2-12. The exception to this rule is to accumulate values in a loop using “Reduction Variables” on page 2-47.

In deciding when to use `parfor`, consider parallel overhead. Parallel overhead includes the time required for communication, coordination and data transfer — sending and receiving data — from client to workers and back. If iteration evaluations are fast, this overhead could be a significant part of the total time. Consider two different types of loop iterations:

- `for`-loops with a computationally demanding task. These loops are generally good candidates for conversion into a `parfor`-loop, because the time needed for computation dominates the time required for data transfer.

- `for`-loops with a simple computational task. These loops generally do not benefit from conversion into a `parfor`-loop, because the time needed for data transfer is significant compared with the time needed for computation.

**Example of parfor With Low Parallel Overhead**

In this example, you start with a computationally demanding task inside a `for`-loop. The `for`-loops are slow, and you speed up the calculation using `parfor`-loops instead. `parfor` splits the execution of `for`-loop iterations over the workers in a parallel pool.
This example calculates the spectral radius of a matrix and converts a for-loop into a parfor-loop. Find out how to measure the resulting speedup and how much data is transferred to and from the workers in the parallel pool.

1. In the MATLAB Editor, enter the following for-loop. Add tic and toc to measure the computation time.

```
tic
n = 200;
A = 500;
a = zeros(n);
for i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

2. Run the script, and note the elapsed time.

   Elapsed time is 31.935373 seconds.

3. In the script, replace the for-loop with a parfor-loop. Add ticBytes and tocBytes to measure how much data is transferred to and from the workers in the parallel pool.

```
tic
ticBytes(gcp);
n = 200;
A = 500;
a = zeros(n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
tocBytes;
toc
```
4 Run the new script on four workers, and run it again. Note that the first run is slower than the second run, because the parallel pool takes some time to start and make the code available to the workers. Note the data transfer and elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers ...

<table>
<thead>
<tr>
<th></th>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15340</td>
<td>7024</td>
</tr>
<tr>
<td>2</td>
<td>13328</td>
<td>5712</td>
</tr>
<tr>
<td>3</td>
<td>13328</td>
<td>5704</td>
</tr>
<tr>
<td>4</td>
<td>13328</td>
<td>5728</td>
</tr>
<tr>
<td>Total</td>
<td>55324</td>
<td>24168</td>
</tr>
</tbody>
</table>

Elapsed time is 10.760068 seconds.

The `parfor` run on four workers is about three times faster than the corresponding `for`-loop calculation. The speed-up is smaller than the ideal speed-up of a factor of four on four workers. This is due to parallel overhead, including the time required to transfer data from the client to the workers and back. Use the `ticBytes` and `tocBytes` results to examine the amount of data transferred. Assume that the time required for data transfer is proportional to the size of the data. This approximation allows you to get an indication of the time required for data transfer, and to compare your parallel overhead with other `parfor`-loop iterations. In this example, the data transfer and parallel overhead are small in comparison with the next example.

The current example has a low parallel overhead and benefits from conversion into a `parfor`-loop. Compare this example with the simple loop iteration in the next example, see “Example of parfor With High Parallel Overhead” on page 2-6.

For another example of a `parfor`-loop with computationally demanding tasks, see “Nested parfor-Loops and for-Loops” on page 2-15.
Example of parfor With High Parallel Overhead

In this example, you write a loop to create a simple sine wave. Replacing the for-loop with a parfor-loop does not speed up your calculation. This loop does not have a lot of iterations, it does not take long to execute and you do not notice an increase in execution speed. This example has a high parallel overhead and does not benefit from conversion into a parfor-loop.

1 Write a loop to create a sine wave. Use tic and toc to measure the time elapsed.

```matlab
tic
n = 1024;
A = zeros(n);
for i = 1:n
    A(i,:) = (1:n) .* sin(i*2*pi/1024);
end
toc
Elapsed time is 0.012501 seconds.
```

2 Replace the for-loop with a parfor-loop. Add ticBytes and tocBytes to measure how much data is transferred to and from the workers in the parallel pool.

```matlab
tic
ticBytes(gcp);
n = 1024;
A = zeros(n);
parfor (i = 1:n)
    A(i,:) = (1:n) .* sin(i*2*pi/1024);
end
tocBytes(gcp)
toc
```

3 Run the script on four workers and run the code again. Note that the first run is slower than the second run, because the parallel pool takes some time to start and make the code available to the workers. Note the data transfer and elapsed time for the second run.

<table>
<thead>
<tr>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1   13176</td>
<td>2.0615e+06</td>
</tr>
<tr>
<td>2   15188</td>
<td>2.0874e+06</td>
</tr>
<tr>
<td>3   13176</td>
<td>2.4056e+06</td>
</tr>
<tr>
<td>4   13176</td>
<td>1.8567e+06</td>
</tr>
</tbody>
</table>
Total 54716 8.4112e+06

Elapsed time is 0.743855 seconds.

Note that the elapsed time is much smaller for the serial for-loop than for the parfor-loop on four workers. In this case, you do not benefit from turning your for-loop into a parfor-loop. The reason is that the transfer of data is much greater than in the previous example, see “Example of parfor With Low Parallel Overhead” on page 2-3. In the current example, the parallel overhead dominates the computing time. Therefore the sine wave iteration does not benefit from conversion into a parfor-loop.

This example illustrates why high parallel overhead calculations do not benefit from conversion into a parfor-loop. To learn more about speeding up your code, see “Convert for-Loops Into parfor-Loops” on page 2-8

See Also
parfor | ticBytes | tocBytes

Related Examples
• “Decide When to Use parfor” on page 2-2
• “Interactively Run a Loop in Parallel Using parfor” on page 1-8
• “Convert for-Loops Into parfor-Loops” on page 2-8
• “Ensure That parfor-Loop Iterations are Independent” on page 2-12
• “Nested parfor-Loops and for-Loops” on page 2-15
• “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23
Convert for-Loops Into parfor-Loops

In some cases, you must modify the code to convert for-loops to parfor-loops. This example shows how to diagnose and fix parfor-loop problems using a simple nested for-loop. Run this code in MATLAB and examine the results.

```matlab
for x = 0:0.1:1
    for k = 2:10
        x(k) = x(k-1) + k;
    end
    x
end
```

To speed up the code, try to convert the for-loops to parfor-loops. Observe that this code produces errors.

```matlab
parfor x = 0:0.1:1
    parfor k = 2:10
        x(k) = x(k-1) + k;
    end
    x
end
```

In this case you cannot simply convert the for-loops to parfor-loops without modification. To make this work, you must change the code in several places. To diagnose the problems, look for Code Analyzer messages in the MATLAB Editor.
This code shows common problems when you try to convert for-loops to parfor-loops.

```
Noninteger loop variable
parfor x = 0:0.1:1

Nested parallel loops parfor y = 2:10

A(y) = A(y-1) + ...
end

Dependent loop body
end
```

To solve these problems, you must modify the code to use parfor. The body of the parfor-loop is executed in a parallel pool using multiple MATLAB workers in a nondeterministic order. Therefore, you have to meet these requirements for the body of the parfor-loop:
The body of the `parfor`-loop must be independent. One loop iteration cannot depend on a previous iteration, because the iterations are executed in parallel in a nondeterministic order. In the example,

\[ x(k) = x(k-1) + k; \]

is not independent, and therefore you cannot use `parfor`. For next steps in dealing with independence issues, see “Ensure That parfor-Loop Iterations are Independent” on page 2-12.

You cannot nest a `parfor`-loop inside another `parfor`-loop. The example has two nested `for`-loops, and therefore you can replace only one `for`-loop with a `parfor`-loop. Instead, you can call a function that uses a `parfor`-loop inside the body of the other `parfor`-loop. However, such nested `parfor`-loops give you no computational benefit, because all workers are used to parallelize the outermost loop. For help dealing with nested loops, see “Nested parfor-Loops and for-Loops” on page 2-15.

`parfor`-loop variables must be consecutive increasing integers. In the example,

\`
parfor x = 0:0.1:1
\`

has non-integer loop variables, and therefore you cannot use `parfor` here. You can solve this problem by changing the value of the loop variable to integer values required by the algorithm. For next steps in troubleshooting `parfor`-loop variables, see “Ensure That parfor-Loop Variables Are Consecutive Increasing Integers” on page 2-32.

You cannot break out of a `parfor`-loop early, as you can in a `for`-loop. Do not include a return or break statement in the body of your `parfor`-loop. Without communication, the other MATLAB instances running the loop do not know when to stop. As an alternative, consider `parfeval`.

If you still have problems converting `for`-loops to `parfor`-loops, see “Troubleshoot Variables in parfor-Loops” on page 2-32.

Tip You can profile a `parfor`-loops using `tic` and `toc` to measure the speedup compared to the corresponding `for`-loop. Use `ticBytes` and `tocBytes` to measure how much data is transferred to and from the workers in the parallel pool. For more information and examples, see “Profiling parfor-loops” on page 2-62.

See Also

`parfor`, `ticBytes`, `tocBytes`
Related Examples

- “Decide When to Use parfor” on page 2-2
- “Ensure That parfor-Loop Iterations are Independent” on page 2-12
- “Nested parfor-Loops and for-Loops” on page 2-15
- “Troubleshoot Variables in parfor-Loops” on page 2-32
Ensure That parfor-Loop Iterations are Independent

If you get an error when you convert for-loops to parfor-loops, ensure that your parfor-loop iterations are independent. parfor-loop iterations have no guaranteed order, while the iteration order in for-loops is sequential. Also parfor-loop iterations are performed on different MATLAB workers in the parallel pool, so that there is no sharing of information between iterations. Therefore one parfor-loop iteration must not depend on the result of a previous iteration. The only exception to this rule is to accumulate values in a loop using “Reduction Variables” on page 2-47.

The following example produces equivalent results, using a for-loop on the left and a parfor-loop on the right. Try the example in your MATLAB Command Window:

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

```plaintext
1     2     3     4     5     6     7     8
```

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

```plaintext
1     2     3     4     5     6     7     8
```

Each element of A is equal to its index. The parfor-loop works because each element is determined by the indexed loop variable only and does not depend on other variables. for-loops with independent tasks are ideal candidates for parfor-loops.

**Note** By default, parfor automatically starts a parallel pool of workers, if you have not started one already. parfor creates a pool using your default cluster profile, if you have set your parallel preferences accordingly.

In the example, the array elements are available in the client workspace after the parfor-loop, exactly as with a for-loop.

Now use a nonindexed variable inside the loop, or a variable whose indexing does not depend on the loop variable i. Try these examples, and note the values of d and i afterward:
Although the elements of A are the same in both examples, the value of d is not. In the for-loop, the iterations are executed sequentially, so afterward d has the value it held in the last iteration of the loop. In the parfor-loop, however, the iterations execute in parallel, so it is impossible to assign d a defined value at the end of the loop. This situation also applies to the loop variable i. Therefore, parfor-loop behavior is defined so that it does not affect the values d and i outside the loop. Their values remain the same before and after the loop. If the variables in your parfor-loop are not independent, then you might get different answers from those in the for-loop. In summary, a parfor-loop requires that each iteration be independent of the other iterations. All code that follows the parfor statement should not depend on the loop iteration sequence.

Code Analyzer can help diagnose whether the loop iterations are dependent. The code in the example shows iterations defined in terms of the previous iteration:

```matlab
parfor k = 2:10
    x(k) = x(k-1) + k;
end
```

Look for Code Analyzer messages in the MATLAB Editor. In this case, Code Analyzer reports the dependency problem.
In other cases, however, Code Analyzer is unable to mark dependencies.

For help with other common parfor problems, see “Nested parfor-Loops and for-Loops” on page 2-15.

See Also
parfor

Related Examples
• “Decide When to Use parfor” on page 2-2
• “Convert for-Loops Into parfor-Loops” on page 2-8
• “Nested parfor-Loops and for-Loops” on page 2-15
• “Troubleshoot Variables in parfor-Loops” on page 2-32
• “Reduction Variables” on page 2-47

More About
• “Evaluate Functions in the Background Using parfeval” on page 1-27
Nested parfor-Loops and for-Loops

In this section...

“Parallelizing Nested Loops” on page 2-15
“Convert Nested for-Loops to parfor” on page 2-16
“Nested Loops: Requirements and Limitations” on page 2-19
“Nested Functions” on page 2-21
“Nested spmd Statements” on page 2-22
“Break and Return Statements” on page 2-22
“P-Code Scripts” on page 2-22

Parallelizing Nested Loops

You cannot use a parfor-loop inside another parfor-loop. As an example, the following nesting of parfor-loops is not allowed:

```matlab
parfor i = 1:10
    parfor j = 1:5
        ...
    end
end
```

Tip You cannot nest parfor directly within another parfor-loop. A parfor-loop can call a function that contains a parfor-loop, but you do not get any additional parallelism.

Code Analyzer in the MATLAB Editor flags the use of parfor inside another parfor-loop:

![PARFOR or SPMD cannot be used inside another PARFOR loop.]

You cannot nest parfor-loops because parallelization can be performed at only one level. Therefore, choose which loop to run in parallel, and convert the other loop to a for-loop.

Consider the following performance issues when dealing with nested loops:
Parallel processing incurs overhead. Generally, you should run the outer loop in parallel, because overhead only occurs once. If you run the inner loop in parallel, then each of the multiple parfor executions incurs an overhead. See “Convert Nested for-Loops to parfor” on page 2-16 for an example how to measure parallel overhead.

- Make sure that the number of iterations exceeds the number of workers. Otherwise, you do not use all available workers.
- Try to balance the parfor-loop iteration times. parfor tries to compensate for some load imbalance.

**Tip** Always run the outermost loop in parallel, because you reduce parallel overhead.

You can also use a function that uses parfor and embed it in a parfor-loop. Parallelization occurs only at the outer level. In the following example, call a function MyFun.m inside the outer parfor-loop. The inner parfor-loop embedded in MyFun.m runs sequentially, not in parallel.

```matlab
parfor i = 1:10
    MyFun(i)
end

function MyFun(i)
    parfor j = 1:5
        ...
    end
end
```

**Tip** Nested parfor-loops generally give you no computational benefit.

**Convert Nested for-Loops to parfor**

A typical use of nested loops is to step through an array using a one-loop variable to index one dimension, and a nested-loop variable to index another dimension. The basic form is:

```matlab
X = zeros(n,m);
for a = 1:n
    for b = 1:m
        X(a,b) = fun(a,b)
    end
end
```
The following code shows a simple example. Use tic and toc to measure the computing time needed.

```matlab
A = 100;
tic
for i = 1:100
   for j = 1:100
      a(i,j) = max(abs(eig(rand(A))));
   end
end
toc
```

Elapsed time is 49.376732 seconds.

You can parallelize either of the nested loops, but you cannot run both in parallel. The reason is that the workers in a parallel pool cannot start or access further parallel pools.

If the loop counted by `i` is converted to a `parfor`-loop, then each worker in the pool executes the nested loops using the `j` loop counter. The `j` loops themselves cannot run as a `parfor` on each worker.

Because parallel processing incurs overhead, you must choose carefully whether you want to convert either the inner or the outer `for`-loop to a `parfor`-loop. The following example shows how to measure the parallel overhead.

First convert only the outer `for`-loop to a `parfor`-loop. Use tic and toc to measure the computing time needed. Use ticBytes and tocBytes to measure how much data is transferred to and from the workers in the parallel pool.

Run the new code, and run it again. The first run is slower than subsequent runs, because the parallel pool takes some time to start and make the code available to the workers.

```matlab
A = 100;
tic
ticBytes(gcp);
parfor i = 1:100
   for j = 1:100
      a(i,j) = max(abs(eig(rand(A))));
   end
end
tocBytes(gcp)
toc
```

<table>
<thead>
<tr>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Next convert only the inner loop to a parfor-loop. Measure the time needed and data transferred as in the previous case.

A = 100;
tic
 ticBytes(gcp);
 for i = 1:100
     parfor j = 1:100
         a(i,j) = max(abs(eig(rand(A))));
     end
 end
tocBytes(gcp)
toc

<table>
<thead>
<tr>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1.3496e+06</td>
<td>5.487e+05</td>
</tr>
<tr>
<td>2 1.3496e+06</td>
<td>5.4858e+05</td>
</tr>
<tr>
<td>3 1.3677e+06</td>
<td>5.6034e+05</td>
</tr>
<tr>
<td>4 1.3476e+06</td>
<td>5.4717e+05</td>
</tr>
<tr>
<td>Total 5.4144e+06</td>
<td>2.2048e+06</td>
</tr>
</tbody>
</table>

Elapsed time is 48.631737 seconds.

If you convert the inner loop to a parfor-loop, both the time and amount of data transferred are much greater than in the parallel outer loop. In this case, the elapsed time is almost the same as in the nested for-loop example. The speedup is smaller than running the outer loop in parallel, because you have more data transfer and thus more parallel overhead. Therefore if you execute the inner loop in parallel, you get no computational benefit compared to running the serial for-loop.

If you want to reduce parallel overhead and speed up your computation, run the outer loop in parallel.
If you convert the inner loop instead, then each iteration of the outer loop initiates a separate parfor-loop. That is, the inner loop conversion creates 100 parfor-loops. Each of the multiple parfor executions incurs overhead. If you want to reduce parallel overhead, you should run the outer loop in parallel instead, because overhead only occurs once.

**Tip** If you want to speed up your code, always run the outer loop in parallel, because you reduce parallel overhead.

### Nested Loops: Requirements and Limitations

If you want to convert a nested for-loop to a parfor-loop, you must ensure that your loop variables are properly classified, see “Troubleshoot Variables in parfor-Loops” on page 2-32. For proper variable classification, you must define the range of a for-loop nested in a parfor-loop by constant numbers or variables. In the following example, the code on the left does not work because you define the upper limit of the for-loop by a function call. The code on the right provides a workaround by first defining a broadcast or constant variable outside the parfor-loop:

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = zeros(100, 200); parfor i = 1:size(A, 1) for j = 1:size(A, 2) A(i, j) = i + j; end</td>
<td>A = zeros(100, 200); n = size(A, 2); parfor i = 1:size(A,1) for j = 1:n A(i, j) = i + j; end</td>
</tr>
</tbody>
</table>

The index variable for the nested for-loop must never be explicitly assigned other than in its for statement. When using the nested for-loop variable for indexing the sliced array, you must use the variable in plain form, not as part of an expression. For example, the following code on the left does not work, but the code on the right does:
Invalid

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

Valid

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

If you use a nested for-loop to index into a sliced array, you cannot use that array elsewhere in the parfor-loop. In the following example, the code on the left does not work because A is sliced and indexed inside the nested for-loop. The code on the right works because v is assigned to A outside of the nested loop:

Invalid

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

Valid

A = zeros(4, 10);
parfor i = 1:4
    v = zeros(1, 10);
    for j = 1:10
        v(j) = i + j;
    end
disp(v(1))
    A(i, :) = v;
end

Suppose that you use multiple for-loops (not nested inside each other) inside a parfor-loop, to index into a single sliced array. In this case, the for-loops must loop over the same range of values. A sliced output variable can be used in only one nested for-loop. In the following example, the code on the left does not work because j and k loop over different values. The code on the right works to index different portions of the sliced array A:

Invalid

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

Valid

A = zeros(4, 10);
parfor i = 1:4
    v = zeros(1, 10);
    for j = 1:10
        v(j) = i + j;
    end
disp(v(1))
    A(i, :) = v;
end
Nested Functions

The body of a `parfor`-loop cannot make reference to a nested function, see “Nested Functions” (MATLAB). However, it can call a nested function by a function handle. Try the following example. Note that `A(idx) = nfcn(idx)` in the `parfor`-loop does not work. You must use `feval` to invoke the `fcn` handle in the `parfor`-loop body:

```matlab
function A = pfeg
    function out = nfcn(in)
        out = 1 + in;
    end
    fcn = @nfcn;
    parfor idx = 1:10
        A(idx) = feval(fcn, idx);
    end
end
```

```plaintext
>> pfeg
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
ans =
    2     3     4     5     6     7     8     9    10    11
```

Tip If you use function handles that refer to nested functions inside a `parfor`-loop, then the values of externally scoped variables are not synchronized among the workers. For more information on handles, see Copying Objects (MATLAB).
**Nested spmd Statements**

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

**Break and Return Statements**

The body of a `parfor`-loop cannot contain `break` or `return` statements. Consider `parfeval` or `parfevalOnAll` instead.

**P-Code Scripts**

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

However, if a script introduces a variable, you cannot call this script from within a `parfor`-loop or `spmd` statement. The reason is that this script would cause a transparency violation. For more details, see “Ensure Transparency in parfor-Loops” on page 2-58.

**See Also**

`break` | `feval` | `parfeval` | `parfevalOnAll` | `parfor` | `return`

**Related Examples**

- “Decide When to Use parfor” on page 2-2
- “Convert for-Loops Into parfor-Loops” on page 2-8
- “Ensure That parfor-Loop Iterations are Independent” on page 2-12
- “Troubleshoot Variables in parfor-Loops” on page 2-32
Scale Up parfor-Loops to Cluster and Cloud

In this example, you start on your local multicore desktop and measure the time required to run a calculation, as a function of increasing numbers of workers. The test is called a *strong scaling* test. It enables you to measure the decrease in time required for the calculation if you add more workers. This dependence is known as *speedup*, and allows you to estimate the *parallel scalability* of your code. You can then decide whether it is useful to increase the number of workers in your parallel pool, and scale up to cluster and cloud computing.

1. Create the function.

   ```matlab
   edit MyCode
   ```

2. In the MATLAB Editor, enter the new `parfor`-loop and add `tic` and `toc` to measure the time elapsed.

   ```matlab
   function a = MyRand(A)
   tic
   parfor i = 1:200
       a(i) = max(abs(eig(rand(A))));
   end
   toc
   ```

3. Save the file, and close the Editor.

4. On the **Parallel > Parallel Preferences** menu, check that your **Default Cluster** is **local** (your desktop machine).

5. In the MATLAB Command Window, define a parallel pool of size 1, and run your function on one worker to calculate the elapsed time. Note the elapsed time for a single worker and shut down your parallel pool.

   ```matlab
   parpool(1);
   a = MyCode(1000);
   Elapsed time is 172.529228 seconds.
   delete(gcp);
   ```

6. Open a new parallel pool of two workers, and run the function again.

   ```matlab
   parpool(2);
   a = MyCode(1000);
   ```
Note the elapsed time; you should see that this now has decreased compared to the single worker case.

Try 4, 8, 12 and 16 workers. Measure the parallel scalability by plotting the elapsed time for each number of workers on a log-log scale.

The figure shows the scalability for a typical multicore desktop PC (blue circle data points). The strong scaling test shows almost linear speedup and significant parallel scalability for up to eight workers. Observe from the figure that, in this case, we do not achieve further speedup for more than eight workers. This result means that, on a local desktop machine, all cores are fully used for 8 workers. You can get a different result on your local desktop, depending on your hardware. To further speed up your parallel application, consider scaling up to cloud or cluster computing.
If you have exhausted your local workers, as in the previous example, you can scale up your calculation to cloud computing. Check your access to cloud computing from the **Parallel > Discover Clusters** menu.

Open a parallel pool in the cloud and run your application without changing your code.

```matlab
parpool(16);
a = MyCode(1000);
```

Note the elapsed time for increasing numbers of cluster workers. Measure the parallel scalability by plotting the elapsed time as a function of number of workers on a log-log scale.
The figure shows typical performance for workers in the cloud (red plus data points). This strong scaling test shows linear speedup and 100% parallel scalability up to 16 workers in the cloud. Consider further scaling up of your calculation by increasing the number of workers in the cloud or on a compute cluster. Note that the parallel scalability can be different, depending on your hardware, for a larger number of workers and other applications.

9 If you have direct access to a cluster, you can scale up your calculation using workers on the cluster. Check your access to clusters from the Parallel > Discover Clusters menu. If you have an account, select cluster, open a parallel pool and run your application without changing your code.

```
parpool(64);
a = MyCode(1000);
```
The figure shows typical strong scaling performance for workers on a cluster (black x data points). Observe that you achieve 100% parallel scalability, persisting up to at least 80 workers on the cluster. Note that this application scales linearly - the speedup is equal to the number of workers used.

This example shows a speedup equal to the number of workers. Not every task can achieve a similar speedup, see for example “Interactively Run a Loop in Parallel Using parfor” on page 1-8.

You might need different approaches for your particular tasks. To learn more about alternative approaches, see “Choose a Parallel Computing Solution” on page 1-20.

**Tip** You can further profile a `parfor`-loop by measuring how much data is transferred to and from the workers in the parallel pool by using `ticBytes` and `tocBytes`. For more information and examples, see “Profiling `parfor`-loops” on page 2-62.

### See Also

#### Related Examples
- “Discover Clusters” on page 6-17
- “Discover Clusters and Use Cluster Profiles” on page 6-15
- “Profiling `parfor`-loops” on page 2-62
- “Interactively Run a Loop in Parallel Using `parfor`” on page 1-8
- “Choose a Parallel Computing Solution” on page 1-20
Use `parfor`-Loops for Reduction Assignments

These two examples show `parfor`-loops using reduction assignments. A reduction is an accumulation across iterations of a loop. The example on the left uses `x` to accumulate a sum across 10 iterations of the loop. The example on the right generates a concatenated array, `1:10`. In both of these examples, the execution order of the iterations on the workers does not matter: while the workers calculate individual results for each iteration, the client properly accumulates and assembles the final loop result.

```matlab
x = 0;
parfor i = 1:10
    x = x + i;
end
x
```
```
x2 = [];
n = 10;
parfor i = 1:n
    x2 = [x2, i];
end
x2
```

If the loop iterations operate in a nondeterministic sequence, you might expect the concatenation sequence in the example on the right to be nonconsecutive. However, MATLAB recognizes the concatenation operation and yields deterministic results.

The next example, which attempts to compute Fibonacci numbers, is not a valid `parfor`-loop because the value of an element of `f` in one iteration depends on the values of other elements of `f` calculated in other iterations.

```matlab
f = zeros(1,50);
f(1) = 1;
f(2) = 2;
parfor n = 3:50
    f(n) = f(n-1) + f(n-2);
end
```

When you are finished with your loop examples, clear your workspace and delete your parallel pool of workers:

```matlab
clear
delete(gcp)
```
See Also

More About

- “Reduction Variables” on page 2-47
- “Ensure That parfor-Loop Iterations are Independent” on page 2-12
Use Objects and Handles in parfor-Loops

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Using Objects in parfor-Loops” on page 2-30</td>
</tr>
<tr>
<td>“Handle Classes” on page 2-30</td>
</tr>
<tr>
<td>“Sliced Variables Referencing Function Handles” on page 2-31</td>
</tr>
</tbody>
</table>

### Using Objects in parfor-Loops

If you are passing objects into or out of a `parfor`-loop, the objects must properly facilitate being saved and loaded. For more information, see “Save and Load Process for Objects” (MATLAB).

You cannot slice the fields of objects because of first-level indexing constraints. For details, see “Sliced Variables” on page 2-41.

For example, in the code on the left, both lines in the loop generate a classification error because of the indexing. In the code on the right, as a workaround for sliced output, you employ separate sliced arrays in the loop. Then you assign the structure fields after the loop is complete.

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parfor</code> <code>i = 1:4</code></td>
<td><code>parfor</code> <code>i = 1:4</code></td>
</tr>
<tr>
<td><code>outputData.outArray1(i) = 1/i;</code></td>
<td><code>outputData.outArray1(i) = 1/i;</code></td>
</tr>
<tr>
<td><code>outputData.outArray2(i) = i^2;</code></td>
<td><code>outputData.outArray2(i) = i^2;</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>end</code></td>
</tr>
<tr>
<td><code>outputData = struct('outArray1',outArray1,'outArray2',outArray2);</code></td>
<td><code>outputData = struct('outArray1',outArray1,'outArray2',outArray2);</code></td>
</tr>
</tbody>
</table>

### Handle Classes

You can send handle objects as inputs to the body of a `parfor`-loop. However, any changes made to handle objects on the workers during loop iterations are not automatically propagated back to the client. That is, changes made inside the loop are not automatically reflected after the loop.

To get the changes back to the client after the loop, explicitly assign the modified handle objects to output variables of the `parfor`-loop. In the following example, `maps` is a sliced input/output variable.
maps = {containers.Map(),containers.Map(),containers.Map()};
parfor ii = 1:numel(maps)
    mymap = maps{ii};  % input slice assigned to local copy
    for jj = 1:1000
        mymap(num2str(jj)) = rand;
    end
    maps{ii} = mymap;  % modified local copy assigned to output slice
end

Sliced Variables Referencing Function Handles

You cannot directly call a function handle with the loop index as an input argument, because this variable cannot be distinguished from a sliced input variable. If you must call a function handle with the loop index variable as an argument, use feval.

The following example uses a function handle and a for-loop.

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

A corresponding parfor-loop does not allow B to reference a function handle. You can work around the problem using feval.

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

See Also

More About
• “Troubleshoot Variables in parfor-Loops” on page 2-32
• “Sliced Variables” on page 2-41
Troubleshoot Variables in parfor-Loops

Ensure That parfor-Loop Variables Are Consecutive Increasing Integers

Loop variables in a parfor-loop must be consecutive increasing integers. For this reason, the following examples return errors:

parfor i = 0:0.2:1 % not integers
parfor j = 1:2:11 % not consecutive
parfor k = 12:-1:1 % not increasing

You can fix these errors by converting the loop variables into a valid range. For example, you can fix the noninteger example as follows:

iValues = 0:0.2:1;
parfor idx = 1:numel(iValues)
i = iValues(idx);
...
end

Avoid Overflows in parfor-Loops

If MATLAB detects that the parfor-loop variable can overflow, it reports an error. To identify and avoid this error, consult the next table:
### Overflow condition

The length of the `parfor`-loop range exceeds the maximum value of the loop variable type.

**Example**

Here, MATLAB reports an error because `length(-128:127)>maxint('int8')`:

```matlab
parfor idx=int8(-128:127)
    idx;
end
```

**Solution**

Use a larger data type for the `parfor`-loop variable. If you want to keep the original data type in your calculations, convert the `parfor`-loop variable inside the `parfor` loop.

```matlab
parfor idx=-128:127
    int8(idx);
end
```

### The initial value of the `parfor`-loop range equals the minimum value of the loop variable type.

**Example**

Here, MATLAB reports an error because `0=intmin('uint32')`:

```matlab
parfor idx=uint32(0:1)
    idx;
end
```

**Solution**

- Use a larger data type with a lower minimum value, as in the previous solution.
- Increment the range of values. For example:

```matlab
parfor idx=uint32(0:1)+1
    idx-1;
end
```

### Solve Variable Classification Issues in `parfor`-Loops

When MATLAB recognizes a name in a `parfor`-loop as a variable, the variable is classified in one of several categories, shown in the following table. Make sure that your variables are uniquely classified and meet the category requirements. `parfor`-loops that violate the requirement return an error.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Loop Variables” on page 2-39</td>
<td>Loop indices</td>
</tr>
<tr>
<td>“Sliced Variables” on page 2-41</td>
<td>Arrays whose segments are operated on by different iterations of the loop</td>
</tr>
<tr>
<td>“Broadcast Variables” on page 2-46</td>
<td>Variables defined before the loop whose value is required inside the loop</td>
</tr>
</tbody>
</table>
To find out which variables you have, examine the code fragment. All variable classifications in the table are represented in this code:

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

If you run into variable classification problems, consider these approaches before you resort to the more difficult method of converting the body of a `parfor`-loop into a function.

- If you use a nested `for`-loop to index into a sliced array, you cannot use that array elsewhere in the `parfor`-loop. The code on the left does not work because `A` is sliced and indexed inside the nested `for`-loop. The code on the right works because `v` is assigned to `A` outside the nested loop. You can compute an entire row, and then perform a single assignment into the sliced output.
<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
</table>
| A = zeros(4, 10); parfor i = 1:4  
  for j = 1:10  
    A(i, j) = i + j;  
  end  
  disp(A(i, 1)) | A = zeros(4, 10); parfor i = 1:4  
  v = zeros(1, 10); for j = 1:10  
    v(j) = i + j;  
  end  
  disp(v(1))  
  A(i, :) = v; |

- The code on the left does not work because the variable `x` in `parfor` cannot be classified. This variable cannot be classified because there are multiple assignments to different parts of `x`. Therefore `parfor` cannot determine whether there is a dependency between iterations of the loop. The code on the right works because you completely overwrite the value of `x`. `parfor` can now determine unambiguously that `x` is a temporary variable.

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
</table>
| parfor idx = 1:10  
  x(1) = 7;  
  x(2) = 8;  
  out(idx) = sum(x); | parfor idx = 1:10  
  x = [7, 8];  
  out(idx) = sum(x); |

- This example shows how to slice the field of a structured array. See `struct` for details. The code on the left does not work because the variable `a` in `parfor` cannot be classified. This variable cannot be classified because the form of indexing is not valid for a sliced variable. The first level of indexing is not the sliced indexing operation, even though the field `x` of `a` appears to be sliced correctly. The code on the right works because you extract the field of the `struct` into a separate variable `tmpx`. `parfor` can now determine correctly that this variable is sliced. In general, you cannot use fields of `structs` or properties of objects as sliced input or output variables in `parfor`.

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
</table>
| a.x = []; parfor idx = 1:10  
  a.x(idx) = 7; | tmpx = []; parfor idx = 1:10  
  tmpx(idx) = 7;  
  a.x = tmpx; |
Structure Arrays in parfor-Loops

Creating Structures as Temporaries

You cannot create a structure in a `parfor`-loop using dot notation assignment. In the code on the left, both lines inside the loop generate a classification error. In the code on the right, as a workaround you can use the `struct` function to create the structure in the loop or in the first field.

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
</table>
| `parfor i = 1:4`  
  `temp.myfield1 = rand();`  
  `temp.myfield2 = i;`  
  `end` | `parfor i = 1:4`  
  `temp = struct();`  
  `temp.myfield1 = rand();`  
  `temp.myfield2 = i;`  
  `end` |
| `parfor i = 1:4`  
  `outputData.outArray1(i) = 1/i;`  
  `outputData.outArray2(i) = i^2;`  
  `end` | `parfor i = 1:4`  
  `outArray1(i) = 1/i;`  
  `outArray2(i) = i^2;`  
  `end` |

Slicing Structure Fields

You cannot use structure fields as sliced `input` or `output` arrays in a `parfor`-loop. In other words, you cannot use the loop variable to index the elements of a structure field. In the code on the left, both lines in the loop generate a classification error because of the indexing. In the code on the right, as a workaround for sliced output, you employ separate sliced arrays in the loop. Then you assign the structure fields after the loop is complete.

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
</table>
| `parfor i = 1:4`  
  `outputData.outArray1(i) = 1/i;`  
  `outputData.outArray2(i) = i^2;`  
  `end` | `parfor i = 1:4`  
  `outArray1(i) = 1/i;`  
  `outArray2(i) = i^2;`  
  `end` |

The workaround for sliced input is to assign the structure field to a separate array before the loop. You can use that new array for the sliced input.

```matlab
inArray1 = inputData.inArray1;  
inArray2 = inputData.inArray2;  
parfor i = 1:4  
  temp1 = inArray1(i);  
end
```
temp2 = inArray2(i);
end

**Converting the Body of a parfor-Loop into a Function**

If all else fails, you can usually solve variable classification problems in `parfor`-loops by converting the body of the `parfor`-loop into a function. In the code on the left, Code Analyzer flags a problem with variable `y`, but cannot resolve it. In the code on the right, you solve this problem by converting the body of the `parfor`-loop into a function.

<table>
<thead>
<tr>
<th>Invalid</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>function parfor_loop_body_bad</code></td>
<td><code>function parfor_loop_body_good</code></td>
</tr>
<tr>
<td>data = rand(5,5);</td>
<td>data = rand(5,5);</td>
</tr>
<tr>
<td>means = zeros(1,5);</td>
<td>means = zeros(1,5);</td>
</tr>
<tr>
<td><code>parfor X = 1:5</code></td>
<td><code>parfor X = 1:5</code></td>
</tr>
<tr>
<td>% Code Analyzer flags problem</td>
<td>% Call a function instead</td>
</tr>
<tr>
<td>% with variable <code>y</code> below</td>
<td>means(X) = computeMeans(data(:,X));</td>
</tr>
<tr>
<td>y.mean = mean(data(:,X));</td>
<td>end</td>
</tr>
<tr>
<td>means(X) = y.mean;</td>
<td>disp(means);</td>
</tr>
<tr>
<td><code>end</code></td>
<td></td>
</tr>
<tr>
<td>disp(means);</td>
<td></td>
</tr>
</tbody>
</table>

% This function now contains the body % of the parfor-loop
`function meansX = computeMeans(dataX)`
y.mean = mean(dataX);
meansX = y.mean;

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

Unambiguous Variable Names

If you use a name that MATLAB cannot unambiguously distinguish as a variable inside a `parfor`-loop, at parse time MATLAB assumes you are referencing a function. Then at runtime, if the function cannot be found, MATLAB generates an error. See “Variable Names” (MATLAB). For example, in the following code `f(5)` could refer either to the fifth element of an array named `f`, or to a function named `f` with an argument of 5. If `f` is not clearly defined as a variable in the code, MATLAB looks for the function `f` on the path when the code runs.

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

**Transparent parfor-loops**

The body of a `parfor`-loop must be *transparent*: all references to variables must be “visible” in the text of the code. For more details about transparency, see “Ensure Transparency in parfor-Loops” on page 2-58.

**Global and Persistent Variables**

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

**See Also**

**More About**

- “Decide When to Use parfor” on page 2-2
- “Convert for-Loops Into parfor-Loops” on page 2-8
- “Ensure That parfor-Loop Iterations are Independent” on page 2-12
- “Nested parfor-Loops and for-Loops” on page 2-15
- “Ensure Transparency in parfor-Loops” on page 2-58
- “Use parfor-Loops for Reduction Assignments” on page 2-28
- “Run Parallel Simulations” (Simulink)
Loop Variables

The loop variable defines the loop index value for each iteration. You set it in the first line of a `parfor` statement.

```matlab
parfor p=1:12
```

For values across all iterations, the loop variable must evaluate to ascending consecutive integers. Each iteration is independent of all others, and each has its own loop index value.

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

This restriction is required, because changing `p` in the `parfor` body cannot guarantee the independence of iterations.

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

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

**Required (static):** You cannot index or subscript the loop variable in any way.

This restriction is required, because referencing a field of a loop variable cannot guarantee the independence of iterations.

The following code attempts to reference a field (b) of the loop variable (p) as if it were a structure. Both lines within the loop are invalid.

```matlab
parfor p = 1:n
    p.b = 3
    x(p) = fun(p.b)
end
```

Similarly, the following code is invalid because it attempts to index the loop variable as a 1-by-1 matrix:

```matlab
parfor p = 1:n
    x = p(1)
end
```
**Required (static):** You cannot use a range increment in `for`-loops nested inside a `parfor`-loop.

Consider the following example:

```matlab
N = 10;
T = 3;
A = zeros(N,T);
B = zeros(N,T);
```

The following code is invalid.

```matlab
parfor i = 1:1:N
    for t = 1:1:T
        A(i,t) = t;
    end
end
```

The following code is valid.

```matlab
parfor i = 1:1:N
    for t = 1:T
        B(i,t) = t;
    end
end
```

**See Also**

`parfor`

**More About**

- “Troubleshoot Variables in parfor-Loops” on page 2-32
Sliced Variables

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

In this example, a slice of A consists of a single element of that array.

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

Characteristics of a Sliced Variable

If a variable in a parfor-loop has all the following characteristics, then the variable is sliced:

- Type of First-Level Indexing — The first level of indexing is either parentheses, (), or braces, {}.
- Fixed Index Listing — Within the first-level parentheses or braces, the list of indices is the same for all occurrences of a given variable.
- Form of Indexing — Within the list of indices for the variable, exactly one index involves the loop variable.
- Shape of Array — The array maintains a constant shape. In assigning to a sliced variable, the right side of the assignment cannot be [], or '', because these operators attempt to delete elements.

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

Here are the forms for the first level of indexing for arrays that are sliced and not sliced.

<table>
<thead>
<tr>
<th>Reference for Variable Not Sliced</th>
<th>Reference for Sliced Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>A(…)</td>
</tr>
<tr>
<td>A.(…)</td>
<td>A{…}</td>
</tr>
</tbody>
</table>
After the first level, you can use any type of valid MATLAB indexing in the second and subsequent levels.

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

\[ A.q\{i,12\} \quad A\{i,12\}.q \]

**Fixed Index Listing.** Within the first-level indexing of a sliced variable, the list of indices is the same for all occurrences of a given variable.

The variable A on the left is not sliced because A is indexed by i and i+1 in different places. In the code on the right, variable A is sliced correctly.

<table>
<thead>
<tr>
<th>Not sliced</th>
<th>Sliced</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Not sliced}</td>
<td>\textbf{Sliced}</td>
</tr>
<tr>
<td>\textbf{parfor} i = 1:k \par B(:) = h(A(i), A(i+1)); \par end</td>
<td>\textbf{parfor} i = 1:k \par B(:) = f(A(i)); \par C(:) = g(A{i}); \par end</td>
</tr>
</tbody>
</table>

The example on the right shows occurrences of first-level indexing using both parentheses and braces in the same loop, which is acceptable.

The following example on the left does not slice A because the indexing of A is not the same in all places. The example on the right slices both A and B. The indexing of A is not the same as the indexing of B. However, the indexing of both A and B are individually consistent.

<table>
<thead>
<tr>
<th>Not sliced</th>
<th>Sliced</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Not sliced}</td>
<td>\textbf{Sliced}</td>
</tr>
<tr>
<td>\textbf{parfor} i=1:10 \par b = A(1,i) + A(2,i) \par end</td>
<td>\textbf{A} = \begin{bmatrix} 1 &amp; 2 &amp; 3 &amp; 4 &amp; 5 &amp; 6 &amp; 7 &amp; 8 &amp; 9 &amp; 10; 10 &amp; 20 &amp; 30 &amp; 40 &amp; 50 &amp; 60 &amp; 70 &amp; 80 &amp; 90 &amp; 100 \end{bmatrix}; \par \textbf{B} = \text{zeros}(1,10); \par \textbf{parfor} i=1:10 \par \textbf{for} n=1:2 \par B(i) = B(i)+A(n,i) \par \textbf{end} \par \textbf{end}</td>
</tr>
</tbody>
</table>

**Form of Indexing.** Within the list of indices for a sliced variable, exactly one of these indices is of the form \(i, i+k, i-k, \) or \(k+i\). The index \(i\) is the loop variable and \(k\) is a constant or a simple (non-indexed) broadcast variable. Every other index is a scalar constant, a simple (non-indexed) broadcast variable, a nested \texttt{for}-loop index, colon, or end.
With i as the loop variable, the A variables shown on the left are not sliced, while the A variables on the right are sliced.

<table>
<thead>
<tr>
<th>Not sliced</th>
<th>Sliced</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i+f(k),j,:,3) % f(k) invalid for slicing</td>
<td>A(i+k,j,:,3)</td>
</tr>
<tr>
<td>A(i,20:30,end) % 20:30 not scalar</td>
<td>A(i,:,end)</td>
</tr>
<tr>
<td>A(i,:,s.field1) % s.field1 not simple broadcast var</td>
<td>A(d(:,k))</td>
</tr>
</tbody>
</table>

When you use other variables along with the loop variable to index an array, you cannot set these variables inside the loop. In effect, such variables are constant over the execution of the entire `parfor` statement. You cannot combine the loop variable with itself to form an index expression.

**Shape of Array.** A sliced variable must maintain a constant shape. The variable A shown here on either line is not sliced:

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

In the first case, A is not sliced because changing the shape of a sliced array would violate assumptions governing communication between the client and workers.

In the second, A is not a sliced output because it is not indexed on the loop variable.

**Sliced Input and Output Variables**

All sliced variables have the characteristics of being input or output. A sliced variable can sometimes have both characteristics. MATLAB transmits sliced input variables from the client to the workers, and sliced output variables from workers back to the client. If a variable is both input and output, it is transmitted in both directions.

In this `parfor`-loop, r is a sliced input variable and b is a sliced output variable.

```matlab
a = 0;  
z = 0;  
r = rand(1,10);  
parfor ii = 1:10  
a = ii;  
z = z + ii;  
b(ii) = r(ii);  
end
```
However, if in each iteration, every reference to an array element is set before it is used, the variable is not a sliced input variable. In this example, all elements of \( A \) are set, and then only those fixed values are used.

```matlab
parfor ii = 1:n
    if someCondition
        A(ii) = 32;
    else
        A(ii) = 17;
    end
end
```

Sliced-output variables can grow dynamically through indexed assignments with default values inserted at intermediate index positions. In this sliced variable example, you can see that the default value of 0 has been inserted at several places in \( a \).

```matlab
a = [];
parfor idx = 1:10
    if rand < 0.5
        a(idx) = idx;
    end
end

disp(a);
```

```
0     2     0     4     5     0     0     8     9    10
```

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

```matlab
A = 1:10;
parfor ii = 1:10
    if rand < 0.5
        A(ii) = 0;
    end
end
```
Scalar Expansion with Sliced Outputs

You cannot use scalar expansion to define a set of values assigned to a sliced output array. For example, the following code attempts to expand the value idx for assignment to each element of the vector defined by \( x(:, \text{idx}) \); this action generates an error.

```matlab
x = zeros(10,12);
parfor idx = 1:12
    x(:,idx) = idx;
end
```

The following code offers a suggested workaround for this limitation.

```matlab
x = zeros(10,12);
parfor idx = 1:12
    x(:,idx) = repmat(idx,10,1);
end
```

See Also

More About

- "Troubleshoot Variables in parfor-Loops" on page 2-32
Broadcast Variables

A broadcast variable is any variable, other than the loop variable or a sliced variable, that does not change inside the loop. At the start of a parfor-loop, the values of any broadcast variables are sent to all workers. This type of variable can be useful or even essential for particular tasks. However, large broadcast variables can cause significant communication between client and workers and increase parallel overhead. Sometimes it is more efficient to use temporary variables for this purpose, creating and assigning them inside the loop.

For more details, see “Temporary Variables” on page 2-55 and “Deciding When to Use parfor” on page 2-2.

See Also

More About

- “Troubleshoot Variables in parfor-Loops” on page 2-32
- “Deciding When to Use parfor” on page 2-2
- “Temporary Variables” on page 2-55
Reduction Variables

MATLAB supports an important exception, called reduction, to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order. MATLAB allows reduction variables in `parfor`-loops.

Reduction variables appear on both sides of an assignment statement, such as any of the following, where `expr` is a MATLAB expression.

<table>
<thead>
<tr>
<th>X = X + expr</th>
<th>X = expr + X</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = X - expr</td>
<td>See Associativity in Reduction Assignments in “Requirements for Reduction Assignments” on page 2-49</td>
</tr>
<tr>
<td>X = X .* expr</td>
<td>X = expr .* X</td>
</tr>
<tr>
<td>X = X * expr</td>
<td>X = expr * X</td>
</tr>
<tr>
<td>X = X &amp; expr</td>
<td>X = expr &amp; X</td>
</tr>
<tr>
<td>X = X</td>
<td>expr</td>
</tr>
<tr>
<td>X = [X, expr]</td>
<td>X = [expr, X]</td>
</tr>
<tr>
<td>X = [X; expr]</td>
<td>X = [expr; X]</td>
</tr>
<tr>
<td>X = min(X, expr)</td>
<td>X = min(expr, X)</td>
</tr>
<tr>
<td>X = max(X, expr)</td>
<td>X = max(expr, X)</td>
</tr>
<tr>
<td>X = union(X, expr)</td>
<td>X = union(expr, X)</td>
</tr>
<tr>
<td>X = intersect(X, expr)</td>
<td>X = intersect(expr, X)</td>
</tr>
</tbody>
</table>

Each of the allowed statements listed in this table is referred to as a *reduction assignment*. By definition, a reduction variable can appear only in assignments of this type.

The general form of a reduction assignment is

| X = f(X, expr) | X = f(expr, X) |

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

```matlab
X = ...; % Do some initialization of X
parfor i = 1:n
```

2-47
\[
X = X + d(i);
\]
end

This loop is equivalent to the following, where you calculate each \(d(i)\) by a different iteration.

\[
X = X + d(1) + \ldots + d(n)
\]

In a regular `for`-loop, the variable \(X\) would get its value either before entering the loop or from the previous iteration of the loop. However, this concept does not apply to `parfor`-loops.

In a `parfor`-loop, the value of \(X\) is never transmitted from client to workers or from worker to worker. Rather, additions of \(d(i)\) are done in each worker, with \(i\) ranging over the subset of \(1:n\) being performed on that worker. The results are then transmitted back to the client, which adds the partial sums of the workers into \(X\). Thus, workers do some of the additions, and the client does the rest.

**Notes About Required and Recommended Guidelines**

If your `parfor` code does not adhere to the guidelines and restrictions labeled as **Required**, you get an error. MATLAB catches some of these errors at the time it reads the code, and others when it executes the code. These errors are labeled as **Required (static)** or **Required (dynamic)** respectively. Guidelines that do not cause errors are labeled as **Recommended**. You can use MATLAB Code Analyzer to help `parfor-loops` comply with the guidelines.

**Basic Rules for Reduction Variables**

The following requirements further define the reduction assignments associated with a given variable.

**Required (static):** For any reduction variable, the same reduction function or operation must be used in all reduction assignments for that variable.

The `parfor`-loop on the left is not valid because the reduction assignment uses \(+\) in one instance, and \([,\) in another. The `parfor`-loop on the right is valid.
Required (static): If the reduction assignment uses *, [], or ;, then X must be consistently specified as the first or second argument in every reduction assignment.

The parfor-loop on the left is not valid because the order of items in the concatenation is not consistent throughout the loop. The parfor-loop on the right is valid.

Requirements for Reduction Assignments

Reduction Assignments. In addition to the specific forms of reduction assignment listed in the table in “Reduction Variables” on page 2-47, the only other (and more general) form of a reduction assignment is

\[ X = f(X, \text{expr}) \]

\[ X = f(\text{expr}, X) \]

Required (static): \( f \) can be a function or a variable. If \( f \) is a variable, then you cannot change \( f \) in the parfor body (in other words, it is a broadcast variable).

If \( f \) is a variable, then for all practical purposes its value at run time is a function handle. However, as long as the right side can be evaluated, the resulting value is stored in \( X \).
The `parfor`-loop on the left does not execute correctly because the statement `f = @times` causes `f` to be classified as a temporary variable. Therefore `f` is cleared at the beginning of each iteration. The `parfor`-loop on the right is correct, because it does not assign `f` inside the loop.

Invalid | Valid
---|---
\[ f = @(x,k)x * k; \]
\[ \text{parfor } i = 1:n \]
\[ \quad a = f(a,i); \]
\[ \quad \text{% loop body continued} \]
\[ \quad f = @times; \quad \text{% Affects } f \]
\[ \text{end} \]
\[ f = @(x,k)x * k; \]
\[ \text{parfor } i = 1:n \]
\[ \quad a = f(a,i); \]
\[ \quad \text{% loop body continued} \]
\[ \text{end} \]

The operators `&&` and `||` are not listed in the table in “Reduction Variables” on page 2-47. Except for `&&` and `||`, all the matrix operations of MATLAB have a corresponding function `f`, such that `u op v` is equivalent to `f(u,v)`. For `&&` and `||`, such a function cannot be written because `u&&v` and `u||v` might or might not evaluate `v`. However, `f(u,v)` always evaluates `v` before calling `f`. Therefore `&&` and `||` are excluded from the table of allowed reduction assignments for a `parfor`-loop.

Every reduction assignment has an associated function `f`. The properties of `f` that ensure deterministic behavior of a `parfor` statement are discussed in the following sections.

**Associativity in Reduction Assignments.** The following practice is recommended for the function `f`, as used in the definition of a reduction variable. However, this rule does not generate an error if not adhered to. Therefore, it is up to you to ensure that your code meets this recommendation.

**Recommended:** To get deterministic behavior of `parfor`-loops, the reduction function `f` must be associative.

To be associative, the function `f` must satisfy the following for all `a`, `b`, and `c`.

\[ f(a,f(b,c)) = f(f(a,b),c) \]

The classification rules for variables, including reduction variables, are purely syntactic. They cannot determine whether the `f` you have supplied is truly associative or not. Associativity is assumed, but if you violate this rule, each execution of the loop might result in different answers.
Note The addition of mathematical real numbers is associative. However, the addition of floating-point numbers is only approximately associative. Different executions of this parfor statement might produce values of X with different round-off errors. You cannot avoid this cost of parallelism.

For example, the statement on the left yields 1, while the statement on the right returns 1 + eps:

\[(1 + \text{eps}/2) + \text{eps}/2\] \quad 1 + (\text{eps}/2 + \text{eps}/2)

Except for the minus operator (-), all special cases listed in the table in “Reduction Variables” on page 2-47 have a corresponding (approximately) associative function. MATLAB calculates the assignment \(X = X - \text{expr}\) by using \(X = X + (-\text{expr})\). (So, technically, the function for calculating this reduction assignment is plus, not minus.) However, the assignment \(X = \text{expr} - X\) cannot be written using an associative function, which explains its exclusion from the table.

Commutativity in Reduction Assignments. Some associative functions, including +, .* , min, and max, intersect, and union, are also commutative. That is, they satisfy the following for all \(a\) and \(b\):

\[f(a,b) = f(b,a)\]

Noncommutative functions include * (because matrix multiplication is not commutative for matrices in which both dimensions have size greater than one), [ , ], and [ ; ]. Noncommutativity is the reason that consistency in the order of arguments to these functions is required. As a practical matter, a more efficient algorithm is possible when a function is commutative as well as associative, and parfor is optimized to exploit commutativity.

**Recommended:** Except in the cases of *, [ , ], and [ ; ], the function \(f\) of a reduction assignment must be commutative. If \(f\) is not commutative, different executions of the loop might result in different answers.

Violating the restriction on commutativity in a function used for reduction could result in unexpected behavior, even if it does not generate an error.

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

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

Examples of identity elements for some functions are listed here.

<table>
<thead>
<tr>
<th>Function</th>
<th>Identity Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>* and .*</td>
<td>1</td>
</tr>
<tr>
<td>[,] and [;]</td>
<td>[]</td>
</tr>
<tr>
<td>&amp;</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

**Recommended:** An overload of +, .*, union, or intersect must be commutative.

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

Similarly, because of the special treatment of \( X = X - expr \), the following is recommended.

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

**Using a Custom Reduction Function**

Suppose that each iteration of a loop performs some calculation, and you are interested in finding which iteration of a loop produces the maximum value. This reduction exercise makes an accumulation across multiple iterations of a loop. Your reduction function must compare iteration results, until the maximum value can be determined after all iterations are compared.
First consider the reduction function itself. To compare one iteration result against another, the function requires as input the current result and the known maximum from other iterations so far. Each of the two inputs is a vector containing iteration results and iteration number.

```matlab
function mc = comparemax(A, B)
% Custom reduction function for 2-element vector input
if A(1) >= B(1) % Compare the two input data values
    mc = A;     % Return the vector with the larger result
else
    mc = B;
end
```

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

- The accumulated maximum and its iteration index, which is the reduction variable `cummax`
- The iteration value and index

If the data value of the current iteration is greater than the maximum in `cummax`, the function returns a vector of the new value and its iteration number. Otherwise, the function returns the existing maximum and its iteration number.

Each iteration calls the reduction function `comparemax` to compare its own data `[dat i]` to data already accumulated in `cummax`. Try the following code for this loop.

```matlab
% First element of cummax is maximum data value
% Second element of cummax is where (iteration) maximum occurs
cummax = [0 0];  % Initialize reduction variable
parfor ii = 1:100
    dat = rand(); % Simulate some actual computation
    cummax = comparemax(cummax, [dat ii]);
end
disp(cummax);
```
See Also

More About

• “Troubleshoot Variables in parfor-Loops” on page 2-32
• “Use parfor-Loops for Reduction Assignments” on page 2-28
Temporary Variables

A temporary variable is any variable that is the target of a direct, nonindexed assignment, but is not a reduction variable. In the following parfor-loop, a and d are temporary variables:

```matlab
a = 0;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;          % Variable a is temporary
    z = z + i;
    if i <= 5
        d = 2*a;     % Variable d is temporary
    end
end
```

In contrast to the behavior of a for-loop, MATLAB clears any temporary variables before each iteration of a parfor-loop. To help ensure the independence of iterations, the values of temporary variables cannot be passed from one iteration of the loop to another. Therefore, temporary variables must be set inside the body of a parfor-loop, so that their values are defined separately for each iteration.

MATLAB does not send temporary variables back to the client. A temporary variable in a parfor-loop has no effect on a variable with the same name that exists outside the loop. This behavior is different from ordinary for-loops.

Uninitialized Temporaries

Temporary variables in a parfor-loop are cleared at the beginning of every iteration. MATLAB can sometimes detect cases in which loop iterations use a temporary variable before it is set in that iteration. In this case, MATLAB issues a static error rather than a run-time error. There is little point in allowing execution to proceed if a run-time error is guaranteed to occur. This kind of error often arises because of confusion between for and parfor, especially regarding the rules of classification of variables. For example:

```matlab
b = true;
parfor i = 1:n
    if b && some_condition(i)
        do_something(i);
        b = false;
    end
end
```
This loop is acceptable as an ordinary for-loop. However, as a parfor-loop, b is a temporary variable because it occurs directly as the target of an assignment inside the loop. Therefore it is cleared at the start of each iteration, so its use in the condition of the if is guaranteed to be uninitialized. If you change parfor to for, the value of b assumes sequential execution of the loop. In that case, do_something(i) is executed only for the lower values of i until b is set false.

**Temporary Variables Intended as Reduction Variables**

Another common cause of uninitialized temporaries can arise when you have a variable that you intended to be a reduction variable. However, if you use it elsewhere in the loop, then it is classified as a temporary variable. For example:

```matlab
s = 0;
parfor i = 1:n
    s = s + f(i);
    ...
    if (s > whatever)
    ...
    end
end
```

If the only occurrences of s are the two in the first statement of the body, s would be classified as a reduction variable. But in this example, s is not a reduction variable because it has a use outside of reduction assignments in the line s > whatever. Because s is the target of an assignment (in the first statement), it is a temporary. Therefore MATLAB issues an error, but points out the possible connection with reduction.

If you change parfor to for, the use of s outside the reduction assignment relies on the iterations being performed in a particular order. In a parfor-loop, it matters that the loop “does not care” about the value of a reduction variable as it goes along. It is only after the loop that the reduction value becomes usable.

**ans Variable**

Inside the body of a parfor-loop, the ans variable is classified as a temporary variable. All considerations and restrictions for temporary variables apply to ans. For example, assignments to ans inside a parfor-loop have no effect on ans outside the loop.
See Also

More About

- “Troubleshoot Variables in parfor-Loops” on page 2-32
- “Reduction Variables” on page 2-47
- “Ensure Transparency in parfor-Loops” on page 2-58
Ensure Transparency in parfor-Loops

The body of a parfor-loop or spmd block must be transparent. Transparency means that all references to variables must be visible in the text of the code.

In the following examples, the variable X is not transferred to the workers. Only the character vector ‘X’ is passed to eval, and X is not visible as an input variable in the loop or block body. As a result, MATLAB issues an error at run time.

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

Similarly, you cannot clear variables from a workspace by executing clear inside a parfor or spmd statement:

```
parfor ii = 1:4
  <statements...>
  clear('X');  % cannot clear: transparency violation
  <statements...>
end
```

Alternatively, you can free up memory used by a variable by setting its value to empty when it is no longer needed.

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

In the case of spmd blocks, you can clear its Composite from the client workspace.

In general, the requirement for transparency restricts all dynamic access to variables, because the entire variable might not be present in any given worker. In a transparent workspace, you cannot create, delete, modify, access, or query variables if you do not explicitly specify these variables in the code.

Examples of other actions or functions that violate transparency in a parfor-loop include:
- who and whos
- evalc, evalin, and assignin with the workspace argument specified as 'caller'
- save and load, unless the output of load is assigned to a variable
- If a script attempts to read or write variables of the parent workspace, then running this script can cause a transparency violation. To avoid this issue, convert the script to a function, and call it with the necessary variables as input or output arguments.

**Note** Transparency applies only to the direct body of the `parfor` or `spmd` construct, and not to any functions called from there. The workaround for `save` and `load` is to hide the calls to `save` and `load` inside a function.

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

**Parallel Simulink Simulations**

You can run Simulink models in parallel with the `parsim` command instead of using `parfor`-loops. For more information and examples of using Simulink in parallel, see “Run Multiple Simulations” (Simulink).

- If your Simulink model requires access to variables contained in a `.mat` file, you must load these parameters in the workspace of each worker. You must do this before the `parfor`-loop, and after opening `parpool`. To achieve this, you can use `spmd` or `parfevalOnAll`, as shown in the examples.

```matlab
spmd
    evalin('base', 'load(''path/to/file'')')
end
parfevalOnAll(@evalin, 0, 'base', 'load(''path/to/file'')')
```

- If your model also requires variables defined in the body of your MATLAB script, you must use `assignin` or `evalin` to move these variables to the base workspace of each worker, in every `parfor` iteration.

**See Also**

`parfor` | `spmd`
More About

- “Troubleshoot Variables in parfor-Loops” on page 2-32
- “Run Single Programs on Multiple Data Sets” on page 3-2
- “Run Parallel Simulations” (Simulink)
Improve parfor Performance

You can improve the performance of parfor-loops in various ways. This includes parallel creation of arrays inside the loop; profiling parfor-loops; slicing arrays; and optimizing your code on local workers before running on a cluster.

Where to Create Arrays

When you create a large array in the client before your parfor-loop, and access it within the loop, you might observe slow execution of your code. To improve performance, tell each MATLAB worker to create its own arrays, or portions of them, in parallel. You can save the time of transferring data from client to workers by asking each worker to create its own copy of these arrays, in parallel, inside the loop. Consider changing your usual practice of initializing variables before a for-loop, avoiding needless repetition inside the loop. You might find that parallel creation of arrays inside the loop improves performance.

Performance improvement depends on different factors, including

- size of the arrays
- time needed to create arrays
- worker access to all or part of the arrays
- number of loop iterations that each worker performs

Consider all factors in this list when you are considering to convert for-loops to parfor-loops. For more details, see “Convert for-Loops Into parfor-Loops” on page 2-8.

As an alternative, consider the parallel.pool.Constant function to establish variables on the pool workers before the loop. These variables remain on the workers after the loop finishes, and remain available for multiple parfor-loops. You might improve performance using parallel.pool.Constant, because the data is transferred only once to the workers.

In this example, you first create a big data set D and execute a parfor-loop accessing D. Then you use D to build a parallel.pool.Constant object, which allows you to reuse the data by copying D to each worker. Measure the elapsed time using tic and toc for each case and note the difference.

function constantDemo
D = rand(1e7, 1);
tic
for i = 1:20
    a = 0;
    parfor j = 1:60
        a = a + sum(D);
    end
end
toc

 tic
 D = parallel.pool.Constant(D);
 for i = 1:20
    b = 0;
    parfor j = 1:60
        b = b + sum(D.Value);
    end
end
toc

>> constantDemo
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Elapsed time is 63.839702 seconds.
Elapsed time is 10.194815 seconds.

In the second case, you send the data only once. You can enhance the performance of the parfor-loop by using the `parallel.pool.Constant` object.

**Profiling parfor-loops**

You can profile a parfor-loop by measuring the time elapsed using tic and toc. You can also measure how much data is transferred to and from the workers in the parallel pool by using ticBytes and tocBytes. Note that this is different from profiling MATLAB code in the usual sense using the MATLAB profiler, see “Profile to Improve Performance” (MATLAB).

This example calculates the spectral radius of a matrix and converts a for-loop into a parfor-loop. Measure the resulting speedup and the amount of transferred data.

1. In the MATLAB Editor, enter the following for-loop. Add tic and toc to measure the time elapsed. Save the file as MyForLoop.m.

   ```matlab
   function a = MyForLoop(A)
   ```
tic
for i = 1:200
    a(i) = max(abs(eig(rand(A))));
end
toc

2 Run the code, and note the elapsed time.

a = MyForLoop(500);

Elapsed time is 31.935373 seconds.

3 In MyForLoop.m, replace the for-loop with a parfor-loop. Add ticBytes and tocBytes to measure how much data is transferred to and from the workers in the parallel pool. Save the file as MyParforLoop.m.

```
ticBytes(gcp);
parfor i = 1:200
    a(i) = max(abs(eig(rand(A))));
end
tocBytes(gcp)
```

4 Run the new code, and run it again. Note that the first run is slower than the second run, because the parallel pool has to be started and you have to make the code available to the workers. Note the elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

a = MyParforLoop(500);

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers ...

<table>
<thead>
<tr>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 15340</td>
<td>7024</td>
</tr>
<tr>
<td>2 13328</td>
<td>5712</td>
</tr>
<tr>
<td>3 13328</td>
<td>5704</td>
</tr>
<tr>
<td>4 13328</td>
<td>5728</td>
</tr>
<tr>
<td>Total 55324</td>
<td>24168</td>
</tr>
</tbody>
</table>

Elapsed time is 10.760068 seconds.

The elapsed time is 31.9 seconds in serial and 10.8 seconds in parallel, and shows that this code benefits from converting to a parfor-loop.
Slicing Arrays

If a variable is initialized before a \texttt{parfor}-loop, then used inside the \texttt{parfor}-loop, it has to be passed to each MATLAB worker evaluating the loop iterations. Only those variables used inside the loop are passed from the client workspace. However, if all occurrences of the variable are indexed by the loop variable, each worker receives only the part of the array it needs.

As an example, you first run a \texttt{parfor}-loop using a sliced variable and measure the elapsed time.

\begin{verbatim}
% Sliced version
M = 100;
N = 1e6;
data = rand(M, N);

tic
parfor idx = 1:M
    out2(idx) = sum(data(idx, :)) ./ N;
end
toc

Elapsed time is 2.261504 seconds.
\end{verbatim}

Now suppose that you accidentally use a reference to the variable \texttt{data} instead of \texttt{N} inside the \texttt{parfor}-loop. The problem here is that the call to \texttt{size(data, 2)} converts the sliced variable into a broadcast (non-sliced) variable.

\begin{verbatim}
% Accidentally non-sliced version
clear
M = 100;
N = 1e6;
data = rand(M, N);

tic
parfor idx = 1:M
    out2(idx) = sum(data(idx, :)) ./ size(data, 2);
end
toc

Elapsed time is 8.369071 seconds.
\end{verbatim}
Note that the elapsed time is greater for the accidentally broadcast variable.

In this case, you can easily avoid the non-sliced usage of data, because the result is a constant, and can be computed outside the loop. In general, you can perform computations that depend only on broadcast data before the loop starts, since the broadcast data cannot be modified inside the loop. In this case, the computation is trivial, and results in a scalar result, so you benefit from taking the computation out of the loop.

**Optimizing on Local vs. Cluster Workers**

Running your code on local workers might offer the convenience of testing your application without requiring the use of cluster resources. However, there are certain drawbacks or limitations with using local workers. Because the transfer of data does not occur over the network, transfer behavior on local workers might not be indicative of how it will typically occur over a network.

With local workers, because all the MATLAB worker sessions are running on the same machine, you might not see any performance improvement from a parfor-loop regarding execution time. This can depend on many factors, including how many processors and cores your machine has. The key point here is that a cluster might have more cores available than your local machine. If your code can be multithreaded by MATLAB, then the only way to go faster is to use more cores to work on the problem, using a cluster.

You might experiment to see if it is faster to create the arrays before the loop (as shown on the left below), rather than have each worker create its own arrays inside the loop (as shown on the right).

Try the following examples running a parallel pool locally, and notice the difference in time execution for each loop. First open a local parallel pool:

```matlab
parpool('local')
```

Run the following examples, and execute again. Note that the first run for each case is slower than the second run, because the parallel pool has to be started and you have to make the code available to the workers. Note the elapsed time, for each case, for the second run.
tic;
n = 200;
M = magic(n);
R = rand(n);
parfor i = 1:n
    A(i) = sum(M(i,:).*R(n+1-i,:));
end
toc

Running on a remote cluster, you might find different behavior, as workers can simultaneously create their arrays, saving transfer time. Therefore, code that is optimized for local workers might not be optimized for cluster workers, and vice versa.

See Also
parallel.pool.Constant

More About
• “Ensure Transparency in parfor-Loops” on page 2-58
• “Use parfor-Loops for Reduction Assignments” on page 2-28
Run Code on Parallel Pools

In this section...

“What Is a Parallel Pool?” on page 2-67
“Automatically Start and Stop a Parallel Pool” on page 2-68
“Alternative Ways to Start and Stop Pools” on page 2-69
“Pool Size and Cluster Selection” on page 2-70

What Is a Parallel Pool?

A parallel pool is a set of MATLAB workers on a compute cluster or desktop. By default, a parallel pool starts automatically when needed by parallel language features such as `parfor`. You can specify the default pool size and cluster in your parallel preferences. The preferences panel displays your pool size and cluster when you select Parallel Preferences in the Parallel menu. You can change pool size and cluster in the Parallel menu. Alternatively, you can choose cluster and pool size using `parcluster` and `parpool` respectively, on the MATLAB command line. See the image for more detail.

The workers in a parallel pool can be used interactively and communicate with each other during the lifetime of the job. You can view your `parpool` jobs in the “Job Monitor” on page 6-32. While these pool workers are reserved for your interactive use, they are not available to other users. You can have only one parallel pool at a time from a MATLAB client session. In MATLAB, the current parallel pool is represented by a parallel.Pool object.
Automatically Start and Stop a Parallel Pool

By default, a parallel pool starts automatically when needed by certain parallel language features. Many functions can automatically start a parallel pool, including:

- `parfor`
- `spmd`
- `distributed`
- `Composite`
- `parallel.pool.Constant`
- `parfeval`
- `parfevalOnAll`
- `gcp`
- `mapreduce`
- `mapreduceer`
- `tall`
- `ticBytes` and `tocBytes`
Your parallel preferences specify which cluster the pool runs on, and the preferred number of workers in the pool. To access your preferences, on the Home tab, in the Environment section, click Parallel > Parallel Preferences.

**Alternative Ways to Start and Stop Pools**

In your parallel preferences, you can turn off the option for the pool to open or close automatically. If you choose not to have the pool open automatically, you can control the pool with the following techniques.

**Control the Parallel Pool from the MATLAB Desktop**

You can use the parallel status indicator in the lower left corner of the MATLAB desktop to start a parallel pool manually.

Click the indicator icon, and select **Start parallel pool**. The pool size and cluster are specified by your parallel preferences and default cluster. Your default cluster is indicated by a check mark on the Parallel > Default Cluster menu.

The menu options are different when a pool is running. You can:

- View the number of workers and cluster name
- Change the time until automatic shut-down
- Shut down the parallel pool
To stop a pool, you can also select **Shut down parallel pool**.

**Programming Interface**

**Start a Parallel Pool**

You can start and stop a parallel pool programmatically by using default settings or specifying alternatives.

To open a parallel pool based on your preference settings:

```matlab
parpool
```

To open a pool of a specific size:

```matlab
parpool(4)
```

To use a cluster other than your default and specify where the pool runs:

```matlab
parpool('MyCluster',4)
```

**Shut Down a Parallel Pool**

To get the current parallel pool and use that object when you want to shut down the pool:

```matlab
p = gcp;
delete(p)
```

**Ensure That No Parallel Pool Is Running**

When you issue the command `gcp` without arguments, you might inadvertently open a pool. To avoid this problem:

```matlab
delete(gcp('nocreate'))
```

**Pool Size and Cluster Selection**

There are several places to specify pool size. Several factors might limit the size of a pool. The actual size of your parallel pool is determined by the combination of the following:
1 Licensing or cluster size

The maximum limit on the number of workers in a pool is restricted by the number of workers in your cluster. This limit might be determined by the number of MATLAB Distributed Computing Server licenses available. In the case of MATLAB job scheduler (MJS), the limit might be determined by the number of workers running in the cluster. A local cluster running on the client machine requires no licensing beyond the one for Parallel Computing Toolbox. The limit on the number of workers is high enough to support the range of known desktop hardware.

2 Cluster profile number of workers (NumWorkers)

A cluster object can set a hard limit on the number of workers, which you specify in the cluster profile. Even if you request more workers at the command line or in your preferences, you cannot exceed the limit set in the applicable profile. Attempting to exceed this number generates an error.

3 Command-line argument

If you specify a pool size at the command line, you override the setting of your preferences. This value must fall within the limits of the applicable cluster profile.

4 Parallel preferences

If you do not specify a pool size at the command line, MATLAB attempts to start a pool with size determined by your parallel preferences. This value is a preference, not a requirement or a request for a specific number of workers. So if a pool cannot start with as many workers as called for in your preferences, you get a smaller pool without any errors. You can set the value of the Preferred number of workers to a large number, so that it never limits the size of the pool that is created. If you need an exact number of workers, specify the number at the command line.

For selection of the cluster on which the pool runs, precedence is determined by the following.

1 The command-line cluster object argument overrides the default profile setting and uses the cluster identified by the profile 'MyProfile'.

```matlab
c = parcluster('MyProfile');
p = parpool(c);
```

2 The cluster is specified in the default profile.

```matlab
p = parpool;
```
See Also
delete | distributed | gcp | parcluster | parfeval | parfor | parpool | spmd

Related Examples
• “Run MATLAB Functions with Automatic Parallel Support” on page 1-24
• “Scale up from Desktop to Cluster”

More About
• “How Parallel Computing Products Run a Job” on page 6-2
• “Decide When to Use parfor” on page 2-2
• “Specify Your Parallel Preferences” on page 6-12
• “Discover Clusters and Use Cluster Profiles” on page 6-15
• “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23
Repeat Random Numbers in parfor-Loops

As described in “Control Random Number Streams” on page 6-40, each worker in a cluster has an independent random number generator stream. By default, therefore, each worker in a pool, and each iteration in a parfor-loop has a unique, independent set of random numbers. Subsequent runs of the parfor-loop generate different numbers.

In a parfor-loop, you cannot control what sequence the iterations execute in, nor can you control which worker runs which iterations. So even if you reset the random number generators, the parfor-loop can generate the same values in a different sequence.

To reproduce the same set of random numbers in a parfor-loop each time the loop runs, you must control random generation by assigning a particular substream for each iteration.

First, give each worker the same stream, one that supports substreams.

```matlab
parpool('local',4);
spmd
    rng(0,'combRecursive');
end
```

Inside the parfor-loop, set the substream index by the loop index. This ensures that each iteration uses its particular set of random numbers, regardless of which worker runs that iteration or what sequence iterations run in.

```matlab
r = zeros(1,16);
parfor i = 1:16
    stream = RandStream.getGlobalStream();
    stream.Substream = i;
    r(i) = rand;
end
r
```

```
Columns 1 through 8
0.7270 0.5582 0.1666 0.3477 0.5671 0.5502 0.0130 0.9179
Columns 9 through 16
0.1567 0.7076 0.8577 0.8923 0.0891 0.2413 0.6074 0.2369
```
Single Program Multiple Data (spmd)

- “Run Single Programs on Multiple Data Sets” on page 3-2
- “Access Worker Variables with Composites” on page 3-8
- “Distributing Arrays to Parallel Workers” on page 3-12
Run Single Programs on Multiple Data Sets

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction” on page 3-2</td>
</tr>
<tr>
<td>“When to Use spmd” on page 3-2</td>
</tr>
<tr>
<td>“Define an spmd Statement” on page 3-3</td>
</tr>
<tr>
<td>“Display Output” on page 3-5</td>
</tr>
<tr>
<td>“MATLAB Path” on page 3-5</td>
</tr>
<tr>
<td>“Error Handling” on page 3-5</td>
</tr>
<tr>
<td>“spmd Limitations” on page 3-6</td>
</tr>
</tbody>
</table>

Introduction

The single program multiple data (spmd) language construct allows seamless interleaving of serial and parallel programming. The `spmd` statement lets you define a block of code to run simultaneously on multiple workers. Variables assigned inside the `spmd` statement on the workers allow direct access to their values from the client by reference via `Composite` objects.

This chapter explains some of the characteristics of `spmd` statements and `Composite` objects.

When to Use `spmd`

The “single program” aspect of `spmd` means that the identical code runs on multiple workers. You run one program in the MATLAB client, and those parts of it labeled as `spmd` blocks run on the workers. When the `spmd` block is complete, your program continues running in the client.

The “multiple data” aspect means that even though the `spmd` statement runs identical code on all workers, each worker can have different, unique data for that code. So multiple data sets can be accommodated by multiple workers.

Typical applications appropriate for `spmd` are those that require running simultaneous execution of a program on multiple data sets, when communication or synchronization is required between the workers. Some common cases are:
• Programs that take a long time to execute — `spmd` lets several workers compute solutions simultaneously.
• Programs operating on large data sets — `spmd` lets the data be distributed to multiple workers.

**Define an `spmd` Statement**

The general form of an `spmd` statement is:

```matlab
spmd
    <statements>
end
```

**Note** If a parallel pool is not running, `spmd` creates a pool using your default cluster profile, if your parallel preferences are set accordingly.

The block of code represented by `<statements>` executes in parallel simultaneously on all workers in the parallel pool. If you want to limit the execution to only a portion of these workers, specify exactly how many workers to run on:

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

This statement requires that `n` workers run the `spmd` code. `n` must be less than or equal to the number of workers in the open parallel pool. If the pool is large enough, but `n` workers are not available, the statement waits until enough workers are available. If `n` is 0, the `spmd` statement uses no workers, and runs locally on the client, the same as if there were not a pool currently running.

You can specify a range for the number of workers:

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

In this case, the `spmd` statement requires a minimum of `m` workers, and it uses a maximum of `n` workers.
If it is important to control the number of workers that execute your `spmd` statement, set the exact number in the cluster profile or with the `spmd` statement, rather than using a range.

For example, create a random matrix on three workers:

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

**Note** All subsequent examples in this chapter assume that a parallel pool is open and remains open between sequences of `spmd` statements.

Unlike a `parfor`-loop, the workers used for an `spmd` statement each have a unique value for `labindex`. This lets you specify code to be run on only certain workers, or to customize execution, usually for the purpose of accessing unique data.

For example, create different sized arrays depending on `labindex`:

```matlab
spmd (3)
    if labindex==1
        R = rand(9,9);
    else
        R = rand(4,4);
    end
end
```

Load unique data on each worker according to `labindex`, and use the same function on each worker to compute a result from the data:

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

The workers executing an `spmd` statement operate simultaneously and are aware of each other. As with a communicating job, you are allowed to directly control communications between the workers, transfer data between them, and use codistributed arrays among them.

For example, use a codistributed array in an `spmd` statement:
spmd (3)
    RR = rand(30, codistributor());
end

Each worker has a 30-by-10 segment of the codistributed array RR. For more information about codistributed arrays, see “Working with Codistributed Arrays” on page 5-5.

**Display Output**

When running an spmd statement on a parallel pool, all command-line output from the workers displays in the client Command Window. Because the workers are MATLAB sessions without displays, any graphical output (for example, figure windows) from the pool does not display at all.

**MATLAB Path**

All workers executing an spmd statement must have the same MATLAB search path as the client, so that they can execute any functions called in their common block of code. Therefore, whenever you use cd, addpath, or rmpath on the client, it also executes on all the workers, if possible. For more information, see the parpool reference page. When the workers are running on a different platform than the client, use the function pctRunOnAll to properly set the MATLAB path on all workers.

**Error Handling**

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

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

The behavior of lastwarn is unspecified at the end of an spmd if used within its body.
spmd Limitations

Nested Functions

Inside a function, the body of an spmd statement cannot make any direct reference to a nested function (MATLAB). However, it can call a nested function by means of a variable defined as a function handle to the nested function.

Because the spmd body executes on workers, variables that are updated by nested functions called inside an spmd statement do not get updated in the workspace of the outer function.

Anonymous Functions

The body of an spmd statement cannot define an anonymous function (MATLAB). However, it can reference an anonymous function by means of a function handle.

Nested spmd Statements

The body of an spmd statement cannot directly contain another spmd. However, it can call a function that contains another spmd statement. The inner spmd statement does not run in parallel in another parallel pool, but runs serially in a single thread on the worker running its containing function.

Nested parfor-Loops

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

Break and Return Statements

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

Global and Persistent Variables

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

More About

• “Ensure Transparency in parfor-Loops” on page 2-58
Access Worker Variables with Composites

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction to Composites” on page 3-8</td>
</tr>
<tr>
<td>“Create Composites in spmd Statements” on page 3-8</td>
</tr>
<tr>
<td>“Variable Persistence and Sequences of spmd” on page 3-10</td>
</tr>
<tr>
<td>“Create Composites Outside spmd Statements” on page 3-11</td>
</tr>
</tbody>
</table>

Introduction to Composites

Composite objects in the MATLAB client session let you directly access data values on the workers. Most often you assigned these variables within `spmd` statements. In their display and usage, Composites resemble cell arrays. There are two ways to create Composites:

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

Create Composites in spmd Statements

When you define or assign values to variables inside an `spmd` statement, the data values are stored on the workers.

After the `spmd` statement, those data values are accessible on the client as Composites. Composite objects resemble cell arrays, and behave similarly. On the client, a Composite has one element per worker. For example, suppose you create a parallel pool of three local workers and run an `spmd` statement on that pool:

```matlab
parpool('local',3)

spmd  % Uses all 3 workers
    MM = magic(labindex+2);  % MM is a variable on each worker
end
MM{1}  % In the client, MM is a Composite with one element per worker
```

8     1     6
3     5     7
4     9     2
A variable might not be defined on every worker. For the workers on which a variable is not defined, the corresponding Composite element has no value. Trying to read that element throws an error.

```matlab
spmd
  if labindex > 1
    HH = rand(4);
  end
end
HH
```

Lab 1: No data
Lab 2: class = double, size = [4  4]
Lab 3: class = double, size = [4  4]

You can also set values of Composite elements from the client. This causes a transfer of data, storing the value on the appropriate worker even though it is not executed within an `spmd` statement:

```matlab
MM{3} = eye(4);
```

In this case, `MM` must already exist as a Composite, otherwise MATLAB interprets it as a cell array.

Now when you do enter an `spmd` statement, the value of the variable `MM` on worker 3 is as set:

```matlab
spmd
  if labindex == 3, MM, end
end
```

Lab 3:

```
MM =
   1   0   0   0
   0   1   0   0
   0   0   1   0
   0   0   0   1
```
Data transfers from worker to client when you explicitly assign a variable in the client workspace using a Composite element:

\[
M = MM\{1\} \quad \% \text{Transfer data from worker 1 to variable M on the client}
\]

\[
\begin{matrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2 \\
\end{matrix}
\]

Assigning an entire Composite to another Composite does not cause a data transfer. Instead, the client merely duplicates the Composite as a reference to the appropriate data stored on the workers:

\[
NN = MM \quad \% \text{Set entire Composite equal to another, without transfer}
\]

However, accessing a Composite’s elements to assign values to other Composites does result in a transfer of data from the workers to the client, even if the assignment then goes to the same worker. In this case, \(NN\) must already exist as a Composite:

\[
NN\{1\} = MM\{1\} \quad \% \text{Transfer data to the client and then to worker}
\]

When finished, you can delete the pool:

\[
delete(gcp)
\]

**Variable Persistence and Sequences of spmd**

The values stored on the workers are retained between `spmd` statements. This allows you to use multiple `spmd` statements in sequence, and continue to use the same variables defined in previous `spmd` blocks.

The values are retained on the workers until the corresponding Composites are cleared on the client, or until the parallel pool is deleted. The following example illustrates data value lifespan with `spmd` blocks, using a pool of four workers:

\[
\text{parpool('local',4)}
\]

\[
\text{spmd}
\]

\[
AA = \text{labindex}; \quad \% \text{Initial setting}
\]

\[
\text{end}
\]

\[
AA(:) \quad \% \text{Composite}
\]

\[
[1] \\
[2]
\]
Create Composites Outside spmd Statements

The Composite function creates Composite objects without using an spmd statement. This might be useful to prepopulate values of variables on workers before an spmd statement begins executing on those workers. Assume a parallel pool is already running:

```matlab
PP = Composite()
```

By default, this creates a Composite with an element for each worker in the parallel pool. You can also create Composites on only a subset of the workers in the pool. See the Composite reference page for more details. The elements of the Composite can now be set as usual on the client, or as variables inside an spmd statement. When you set an element of a Composite, the data is immediately transferred to the appropriate worker:

```matlab
for ii = 1:numel(PP)
    PP{ii} = ii;
end
```
Distributing Arrays to Parallel Workers

Using Distributed Arrays to Partition Data Across Workers

Depending on how your data fits in memory, choose one of the following methods:

- If your data is currently in the memory of your local machine, you can use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool. This option can be useful for testing or before performing operations which significantly increase the size of your arrays, such as `repmat`.

- If your data does not fit in the memory of your local machine, but does fit in the memory of your cluster, you can use `datastore` with the `distributed` function to read data into the memory of the workers of a parallel pool.

- If your data does not fit in the memory of your cluster, you can use `datastore` with `tall` arrays to partition and process your data in chunks. See also “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60.

Load Distributed Arrays in Parallel Using datastore

If your data does not fit in the memory of your local machine, but does fit in the memory of your cluster, you can use `datastore` with the `distributed` function to create distributed arrays and partition the data among your workers.

This example shows how to create and load distributed arrays using `datastore`. Create a datastore using a tabular file of airline flight data. This data set is too small to show equal partitioning of the data over the workers. To simulate a large data set, artificially increase the size of the datastore using `repmat`.

```matlab
files = repmat({'airlinesmall.csv'}, 10, 1);
ds = tabularTextDatastore(files);
```

Select the example variables.
ds.SelectedVariableNames = {'DepTime','DepDelay'};
ds.TreatAsMissing = 'NA';

Create a distributed table by reading the datastore in parallel. Partition the datastore with one partition per worker. Each worker then reads all data from the corresponding partition. The files must be in a shared location that is accessible by the workers.

dt = distributed(ds);
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

Display summary information about the distributed table.

summary(dt)
Variables:

DepTime: 1,235,230×1 double
  Values:
    min          1
    max       2505
    NaNs    23,510

DepDelay: 1,235,230×1 double
  Values:
    min      -1036
    max       1438
    NaNs    23,510

Determine the size of the tall table.

size(dt)
ans =
    1235230         2

Return the first few rows of dt.

head(dt)
ans =
    DepTime     DepDelay
Finally, check how much data each worker has loaded.

```matlab
spmd, dt, end
```

Lab 1:

This worker stores `dt2(1:370569,:)`.

- **LocalPart:** `{370569×2 table}`
- **Codistributor:** `{1×1 codistributor1d}`

Lab 2:

This worker stores `dt2(370570:617615,:)`.

- **LocalPart:** `{247046×2 table}`
- **Codistributor:** `{1×1 codistributor1d}`

Lab 3:

This worker stores `dt2(617616:988184,:)`.

- **LocalPart:** `{370569×2 table}`
- **Codistributor:** `{1×1 codistributor1d}`

Lab 4:

This worker stores `dt2(988185:1235230,:)`.

- **LocalPart:** `{247046×2 table}`
- **Codistributor:** `{1×1 codistributor1d}`
Note that the data is partitioned equally over the workers. For more details on datastore, see “What Is a Datastore?” (MATLAB)

For more details about workflows for big data, see “Choose a Parallel Computing Solution” on page 1-20.

**Alternative Methods for Creating Distributed and Codistributed Arrays**

If your data fits in the memory of your local machine, you can use distributed arrays to partition the data among your workers. Use the distributed function to create a distributed array in the MATLAB client, and store its data on the workers of the open parallel pool. A distributed array is distributed in one dimension, and as evenly as possible along that dimension among the workers. You cannot control the details of distribution when creating a distributed array.

You can create a distributed array in several ways:

- Use the distributed function to distribute an existing array from the client workspace to the workers of a parallel pool.
- Use any of the distributed functions to directly construct a distributed array on the workers. This technique does not require that the array already exists in the client, thereby reducing client workspace memory requirements. Functions include eye(___,'distributed') and rand(___,'distributed'). For a full list, see the distributed object reference page.
- Create a codistributed array inside an spmd statement, and then access it as a distributed array outside the spmd statement. This technique lets you use distribution schemes other than the default.

The first two techniques do not involve spmd in creating the array, but you can use spmd to manipulate arrays created this way. For example:

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

```matlab
parpool('local',2) % Create pool
W = ones(6,6);
W = distributed(W); % Distribute to the workers
spmd
    T = W*2; % Calculation performed on workers, in parallel.
    % T and W are both codistributed arrays here.
end
```
T           % View results in client.
whos         % T and W are both distributed arrays here.
delete(gcp)  % Stop pool

Alternatively, you can use the `codistributed` function, which allows you to control more options such as dimensions and partitions, but is often more complicated. You can create a `codistributed` array by executing on the workers themselves, either inside an `spmd` statement, in `pmode`, or inside a communicating job. When creating a `codistributed` array, you can control all aspects of distribution, including dimensions and partitions.

The relationship between distributed and codistributed arrays is one of perspective.Codistributed arrays are partitioned among the workers from which you execute code to create or manipulate them. When you create a distributed array in the client, you can access it as a codistributed array inside an `spmd` statement. When you create a codistributed array in an `spmd` statement, you can access it as a distributed array in the client. Only `spmd` statements let you access the same array data from two different perspectives.

You can create a `codistributed` array in several ways:

• Use the `codistributed` function inside an `spmd` statement, a communicating job, or `pmode` to codistribute data already existing on the workers running that job.
• Use any of the codistributed functions to directly construct a codistributed array on the workers. This technique does not require that the array already exists in the workers. Functions include `eye(___,'codistributed')` and `rand(___,'codistributed')`. For a full list, see the `codistributed` object reference page.
• Create a distributed array outside an `spmd` statement, then access it as a codistributed array inside the `spmd` statement running on the same parallel pool.

Create a codistributed array inside an `spmd` statement using a nondefault distribution scheme. First, define 1-D distribution along the third dimension, with 4 parts on worker 1, and 12 parts on worker 2. Then create a 3-by-3-by-16 array of zeros.

```matlab
parpool('local',2) % Create pool
spmd
    codist = codistributor1d(3,[4,12]);
    Z = zeros(3,3,16,codist);
    Z = Z + labindex;
end
Z  % View results in client.
```
% Z is a distributed array here.
delete(gcp) % Stop pool

For more details on codistributed arrays, see “Working with Codistributed Arrays” on page 5-5.

See Also
codistributed | datastore | distributed | eye | rand | repmat | spmd | tall

Related Examples
• “Run MATLAB Functions with Distributed Arrays” on page 5-24
• “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
• “What Is a Datastore?” (MATLAB)
• “Choose a Parallel Computing Solution” on page 1-20
• “Use Tall Arrays on a Parallel Pool” on page 6-63

More About
• “Datastore” (MATLAB)
• “Tall Arrays” (MATLAB)
Interactive Parallel Computation with pmode

This chapter describes interactive pmode in the following sections:

• “pmode Versus spmd” on page 4-2
• “Run Communicating Jobs Interactively Using pmode” on page 4-3
• “Parallel Command Window” on page 4-10
• “Running pmode Interactive Jobs on a Cluster” on page 4-15
• “Plotting Distributed Data Using pmode” on page 4-16
• “pmode Limitations and Unexpected Results” on page 4-18
• “pmode Troubleshooting” on page 4-19
pmode Versus spmd

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

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

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

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

1. Start the pmode with the pmode command.

   ```matlab
   pmode start local 4
   ```

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

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

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

   ```matlab
   P>> help magic
   ```

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

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

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

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

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

Create a replicated array on all the workers.

```
P>> segment = [1 2; 3 4; 5 6]
```
Assign a unique value to the array on each worker, dependent on the worker number (labindex). With a different value on each worker, this is a variant array.

```
P>> segment = segment + 10*labindex
```
Until this point in the example, the variant arrays are independent, other than having
the same name. Use the codistributed.build function to aggregate the array
segments into a coherent array, distributed among the workers.

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

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

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

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

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

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

Thus, section is now a variant array because it is different on each worker.
If you need the entire array in one workspace, use the `gather` function.

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

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

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

```
pmode lab2client combined 1
```

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

```
whos combined
```

To see the array, type its name.

```
combined
```

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

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

Calling the `codistributor1d` function without arguments specifies the default distribution, which is by columns in this case, distributed as evenly as possible.
If you require distribution along a different dimension, you can use the `redistribute` function. In this example, the argument 1 to `codistributor1d` specifies distribution of the array along the first dimension (rows).

```matlab
P>> distobj = codistributor1d(1);
P>> I = redistribute(I, distobj)
P>> getLocalPart(I)
```
Exit pmode and return to the regular MATLAB desktop.

P>> pmode exit
Parallel Command Window

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

```
pmode start local 4
```

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

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

When you select one or more lines in the command history and right-click, you see the following context menu.
You have several options for how to arrange the tiles showing your worker outputs. Usually, you will choose an arrangement that depends on the format of your data. For example, the data displayed until this point in this section, as in the previous figure, is distributed by columns. It might be convenient to arrange the tiles side by side.

This arrangement results in the following figure, which might be more convenient for viewing data distributed by columns.
Alternatively, if the data is distributed by rows, you might want to stack the worker tiles vertically. For the following figure, the data is reformatted with the command

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

When you rearrange the tiles, you see the following.

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

You can have multiple workers send their output to the same tile or tab. This allows you to have fewer tiles or tabs than workers.
In this case, the window provides shading to help distinguish the outputs from the various workers.
Running pmode Interactive Jobs on a Cluster

When you run pmode on a cluster of workers, you are running a job that is much like any other communicating job, except it is interactive. The cluster can be heterogeneous, but with certain limitations described at System Requirements; carefully locate your scheduler on that page and note that pmode sessions run as jobs described as “parallel applications that use inter-worker communication.”

Many of the job's properties are determined by the cluster profile. For more details about creating and using profiles, see “Discover Clusters and Use Cluster Profiles” on page 6-15.

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

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

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

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

For details on all the command options, see the `pmode` reference page.
Plotting Distributed Data Using pmode

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

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

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

The following example illustrates this technique.

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

```matlab
P>> D = zeros(1,100,codistributor1d())
Lab 1: This lab stores D(1:25).
Lab 2: This lab stores D(26:50).
Lab 3: This lab stores D(51:75).
Lab 4: This lab stores D(76:100).
```

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

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

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

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

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

```matlab
pmode lab2client P 1
plot(P)
```
This is not the only way to plot codistributed data. One alternative method, especially useful when running noninteractive communicating jobs, is to plot the data to a file, then view it from a later MATLAB session.
**pmode Limitations and Unexpected Results**

**Using Graphics in pmode**

**Displaying a GUI**

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

```
pmode lab2client var labindex
```

Then use the graphical tool on the MATLAB client.

**Using Simulink Software**

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

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

For testing connectivity between the client machine and the machines of your compute cluster, you can use Admin Center. For more information about Admin Center, including how to start it and how to test connectivity, see “Start Admin Center” (MATLAB Distributed Computing Server) and “Test Connectivity” (MATLAB Distributed Computing Server).

Hostname Resolution

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

Socket Connections

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

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

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

- “Nondistributed Versus Distributed Arrays” on page 5-2
- “Working with Codistributed Arrays” on page 5-5
- “Looping Over a Distributed Range (for-drange)” on page 5-20
- “Run MATLAB Functions with Distributed Arrays” on page 5-24
Nondistributed Versus Distributed Arrays

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction” on page 5-2</td>
</tr>
<tr>
<td>“Nondistributed Arrays” on page 5-2</td>
</tr>
<tr>
<td>“Codistributed Arrays” on page 5-4</td>
</tr>
</tbody>
</table>

**Introduction**

Many built-in data types and data structures supported by MATLAB software are also supported in the MATLAB parallel computing environment. This includes arrays of any number of dimensions containing numeric, character, logical values, cells, or structures. In addition to these basic building blocks, the MATLAB parallel computing environment also offers different types of arrays.

**Nondistributed Arrays**

When you create a nondistributed array, MATLAB constructs a separate array in the workspace of each worker, using the same variable name on all workers. Any operation performed on that variable affects all individual arrays assigned to it. If you display from worker 1 the value assigned to this variable, all workers respond by showing the array of that name that resides in their workspace.

The state of a nondistributed array depends on the value of that array in the workspace of each worker:

- “Replicated Arrays” on page 5-2
- “Variant Arrays” on page 5-3
- “Private Arrays” on page 5-3

**Replicated Arrays**

A replicated array resides in the workspaces of all workers, and its size and content are identical on all workers. When you create the array, MATLAB assigns it to the same variable on all workers. If you display in spmd the value assigned to this variable, all workers respond by showing the same array.

```
spmd, A = magic(3), end
```
Variant Arrays

A **variant array** also resides in the workspaces of all workers, but its content differs on one or more workers. When you create the array, MATLAB assigns a different value to the same variable on all workers. If you display the value assigned to this variable, all workers respond by showing their version of the array.

```matlab
spmd, A = magic(3) + labindex - 1, end
```

A replicated array can become a variant array when its value becomes unique on each worker.

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

**Private Arrays**

A **private array** is defined on one or more, but not all workers. You could create this array by using `labindex` in a conditional statement, as shown here:

```matlab
spmd
    if labindex >= 3, A = magic(3) + labindex - 1, end
end
```

```plaintext
WORKER 1  |  WORKER 2  |  WORKER 3  |  WORKER 4
-----------|------------|------------|------------
8 1 6     | 8 1 6     | 8 1 6     | 8 1 6
3 5 7     | 3 5 7     | 3 5 7     | 3 5 7
4 9 2     | 4 9 2     | 4 9 2     | 4 9 2

A is undefined | A is undefined | 10 3 8 | 11 4 9
A is undefined | A is undefined | 5 7 9 | 6 8 10
A is undefined | A is undefined | 6 11 4 | 7 12 5
```
**Codistributed Arrays**

With replicated and variant arrays, the full content of the array is stored in the workspace of each worker. *Codistributed arrays*, on the other hand, are partitioned into segments, with each segment residing in the workspace of a different worker. Each worker has its own array segment to work with. Reducing the size of the array that each worker has to store and process means a more efficient use of memory and faster processing, especially for large data sets.

This example distributes a 3-by-10 replicated array A across four workers. The resulting array D is also 3-by-10 in size, but only a segment of the full array resides on each worker.

```
spmd
    A = [11:20; 21:30; 31:40];
    D = codistributed(A);
    getLocalPart(D)
end
```

<table>
<thead>
<tr>
<th>WORKER 1</th>
<th>WORKER 2</th>
<th>WORKER 3</th>
<th>WORKER 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 12 13</td>
<td>14 15 16</td>
<td>17 18</td>
<td>19 20</td>
</tr>
<tr>
<td>21 22 23</td>
<td>24 25 26</td>
<td>27 28</td>
<td>29 30</td>
</tr>
<tr>
<td>31 32 33</td>
<td>34 35 36</td>
<td>37 38</td>
<td>39 40</td>
</tr>
</tbody>
</table>

For more details on using codistributed arrays, see “Working with Codistributed Arrays” on page 5-5.
Working with Codistributed Arrays

How MATLAB Software Distributes Arrays

When you distribute an array to a number of workers, MATLAB software partitions the array into segments and assigns one segment of the array to each worker. You can partition a two-dimensional array horizontally, assigning columns of the original array to the different workers, or vertically, by assigning rows. An array with N dimensions can be partitioned along any of its N dimensions. You choose which dimension of the array is to be partitioned by specifying it in the array constructor command.

For example, to distribute an 80-by-1000 array to four workers, you can partition it either by columns, giving each worker an 80-by-250 segment, or by rows, with each worker getting a 20-by-1000 segment. If the array dimension does not divide evenly over the number of workers, MATLAB partitions it as evenly as possible.

The following example creates an 80-by-1000 replicated array and assigns it to variable A. In doing so, each worker creates an identical array in its own workspace and assigns it to variable A, where A is local to that worker. The second command distributes A, creating a single 80-by-1000 array D that spans all four workers. Worker 1 stores columns 1 through 250, worker 2 stores columns 251 through 500, and so on. The default distribution is by the last nonsingleton dimension, thus, columns in this case of a 2-dimensional array.

```matlab
spmd
    A = zeros(80, 1000);
    D = codistributed(A)
end
```
Lab 1: This lab stores \( D(:,1:250) \).
Lab 2: This lab stores \( D(:,251:500) \).
Lab 3: This lab stores \( D(:,501:750) \).
Lab 4: This lab stores \( D(:,751:1000) \).

Each worker has access to all segments of the array. Access to the local segment is faster than to a remote segment, because the latter requires sending and receiving data between workers and thus takes more time.

**How MATLAB Displays a Codistributed Array**

For each worker, the MATLAB Parallel Command Window displays information about the codistributed array, the local portion, and the codistributor. For example, an 8-by-8 identity matrix codistributed among four workers, with two columns on each worker, displays like this:

```matlab
>> spmd
II = eye(8,'codistributed')
end
Lab 1:
  This lab stores II(:,1:2).
  LocalPart: [8x2 double]
  Codistributor: [1x1 codistributor1d]
Lab 2:
  This lab stores II(:,3:4).
  LocalPart: [8x2 double]
  Codistributor: [1x1 codistributor1d]
Lab 3:
  This lab stores II(:,5:6).
  LocalPart: [8x2 double]
  Codistributor: [1x1 codistributor1d]
Lab 4:
  This lab stores II(:,7:8).
  LocalPart: [8x2 double]
  Codistributor: [1x1 codistributor1d]
```

To see the actual data in the local segment of the array, use the `getLocalPart` function.

**How Much Is Distributed to Each Worker**

In distributing an array of \( N \) rows, if \( N \) is evenly divisible by the number of workers, MATLAB stores the same number of rows \( (N/\text{numlabs}) \) on each worker. When this
number is not evenly divisible by the number of workers, MATLAB partitions the array as evenly as possible.

MATLAB provides codistributor object properties called Dimension and Partition that you can use to determine the exact distribution of an array. See “Indexing into a Codistributed Array” on page 5-14 for more information on indexing with codistributed arrays.

Distribution of Other Data Types

You can distribute arrays of any MATLAB built-in data type, and also numeric arrays that are complex or sparse, but not arrays of function handles or object types.

Creating a Codistributed Array

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

- “Partitioning a Larger Array” on page 5-7 — Start with a large array that is replicated on all workers, and partition it so that the pieces are distributed across the workers. This is most useful when you have sufficient memory to store the initial replicated array.
- “Building from Smaller Arrays” on page 5-8 — Start with smaller variant or replicated arrays stored on each worker, and combine them so that each array becomes a segment of a larger codistributed array. This method reduces memory requirements as it lets you build a codistributed array from smaller pieces.
- “Using MATLAB Constructor Functions” on page 5-9 — Use any of the MATLAB constructor functions like rand or zeros with a codistributor object argument. These functions offer a quick means of constructing a codistributed array of any size in just one step.

Partitioning a Larger Array

If you have a large array already in memory that you want MATLAB to process more quickly, you can partition it into smaller segments and distribute these segments to all of the workers using the codistributed function. Each worker then has an array that is a fraction the size of the original, thus reducing the time required to access the data that is local to each worker.

As a simple example, the following line of code creates a 4-by-8 replicated matrix on each worker assigned to the variable A:
A =
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48

The next line uses the codistributed function to construct a single 4-by-8 matrix D that is distributed along the second dimension of the array:

spmd
    D = codistributed(A);
    getLocalPart(D)
end

1: Local Part | 2: Local Part | 3: Local Part | 4: Local Part
  11    12   |     13    14   |     15    16   |     17    18
  21    22   |     23    24   |     25    26   |     27    28
  31    32   |     33    34   |     35    36   |     37    38
  41    42   |     43    44   |     45    46   |     47    48

Arrays A and D are the same size (4-by-8). Array A exists in its full size on each worker, while only a segment of array D exists on each worker.

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

Examining the variables in the client workspace, an array that is codistributed among the workers inside an spmd statement, is a distributed array from the perspective of the client outside the spmd statement. Variables that are not codistributed inside the spmd, are Composites in the client outside the spmd.

whos
Name      Size            Bytes  Class
A         1x4               613  Composite
D         4x8               649  distributed

See the codistributed function reference page for syntax and usage information.

Building from Smaller Arrays

The codistributed function is less useful for reducing the amount of memory required to store data when you first construct the full array in one workspace and then partition it into distributed segments. To save on memory, you can construct the smaller pieces (local
part) on each worker first, and then use **codistributed.build** to combine them into a single array that is distributed across the workers.

This example creates a 4-by-250 variant array **A** on each of four workers and then uses **codistributor** to distribute these segments across four workers, creating a 4-by-1000 codistributed array. Here is the variant array, **A**:

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

<table>
<thead>
<tr>
<th>WORKER 1</th>
<th>WORKER 2</th>
<th>WORKER 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 ... 250</td>
<td>251 252 ... 500</td>
<td>501 502 ... 750</td>
</tr>
<tr>
<td>251 252 ... 500</td>
<td>501 502 ... 750</td>
<td>751 752 ...1000</td>
</tr>
<tr>
<td>501 502 ... 750</td>
<td>751 752 ...1000</td>
<td>1001 1002 ...1250</td>
</tr>
<tr>
<td>751 752 ...1000</td>
<td>1001 1002 ...1250</td>
<td>1251 1252 ...1500</td>
</tr>
</tbody>
</table>

Now combine these segments into an array that is distributed by the first dimension (rows). The array is now 16-by-250, with a 4-by-250 segment residing on each worker:

```matlab
spmd
D = codistributed.build(A, codistributor1d(1,[4 4 4 4],[16 250]));
end
```

Lab 1:

This lab stores **D**(1:4,:).

LocalPart: [4x250 double]
Codistributor: [1x1 codistributor1d]

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

**Using MATLAB Constructor Functions**

MATLAB provides several array constructor functions that you can use to build codistributed arrays of specific values, sizes, and classes. These functions operate in the same way as their nondistributed counterparts in the MATLAB language, except that they distribute the resultant array across the workers using the specified codistributor object, **codist**.
Constructor Functions

The codistributed constructor functions are listed here. Use the codist argument (created by the codistributor function: codist=codistributor()) to specify over which dimension to distribute the array. See the individual reference pages for these functions for further syntax and usage information.

- `eye(___,codist)`
- `false(___,codist)`
- `Inf(___,codist)`
- `NaN(___,codist)`
- `ones(___,codist)`
- `rand(___,codist)`
- `randi(___,codist)`
- `randn(___,codist)`
- `true(___,codist)`
- `zeros(___,codist)`

- `codistributed.cell(m,n,...,codist)`
- `codistributed.colon(a,d,b)`
- `codistributed.linspace(m,n,...,codist)`
- `codistributed.logspace(m,n,...,codist)`
- `sparse(m,n,codist)`
- `codistributed.speye(m,...,codist)`
- `codistributed.sprand(m,n,density,codist)`
- `codistributed.sprandn(m,n,density,codist)`

Local Arrays

That part of a codistributed array that resides on each worker is a piece of a larger array. Each worker can work on its own segment of the common array, or it can make a copy of that segment in a variant or private array of its own. This local copy of a codistributed array segment is called a local array.

Creating Local Arrays from a Codistributed Array

The `getLocalPart` function copies the segments of a codistributed array to a separate variant array. This example makes a local copy L of each segment of codistributed array D. The size of L shows that it contains only the local part of D for each worker. Suppose you distribute an array across four workers:

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

returns on each worker:

3    80  
3    20

Each worker recognizes that the codistributed array D is 3-by-80. However, notice that the size of the local part, L, is 3-by-20 on each worker, because the 80 columns of D are distributed over four workers.

**Creating a Codistributed from Local Arrays**

Use the `codistributed.build` function to perform the reverse operation. This function, described in “Building from Smaller Arrays” on page 5-8, combines the local variant arrays into a single array distributed along the specified dimension.

Continuing the previous example, take the local variant arrays L and put them together as segments to build a new codistributed array X.

```matlab
spmd
codist = codistributor1d(2,[20 20 20 20],[3 80]);
X = codistributed.build(L,codist);
    size(X)
end
```

returns on each worker:

3    80

**Obtaining information About the Array**

MATLAB offers several functions that provide information on any particular array. In addition to these standard functions, there are also two functions that are useful solely with codistributed arrays.

**Determining Whether an Array Is Codistributed**

The `iscodistributed` function returns a logical 1 (true) if the input array is codistributed, and logical 0 (false) otherwise. The syntax is
where D is any MATLAB array.

**Determining the Dimension of Distribution**

The codistributor object determines how an array is partitioned and its dimension of distribution. To access the codistributor of an array, use the `getCodistributor` function. This returns two properties, `Dimension` and `Partition`:

```matlab
spmd, getCodistributor(X), end
```

```
Dimension: 2
Partition: [20 20 20 20]
```

The `Dimension` value of 2 means the array X is distributed by columns (dimension 2); and the `Partition` value of `[20 20 20 20]` means that twenty columns reside on each of the four workers.

To get these properties programmatically, return the output of `getCodistributor` to a variable, then use dot notation to access each property:

```matlab
spmd
    C = getCodistributor(X);
    part = C.Partition
    dim  = C.Dimension
end
```

**Other Array Functions**

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

- `length` — Returns the length of a specific dimension.
- `ndims` — Returns the number of dimensions.
- `numel` — Returns the number of elements in the array.
- `size` — Returns the size of each dimension.
- `is*` — Many functions that have names beginning with 'is', such as `ischar` and `issparse`. 
Changing the Dimension of Distribution

When constructing an array, you distribute the parts of the array along one of the array's dimensions. You can change the direction of this distribution on an existing array using the `redistribute` function with a different codistributor object.

Construct an 8-by-16 codistributed array D of random values distributed by columns on four workers:

```matlab
spmd
    D = rand(8,16,codistributor());
    size(getLocalPart(D))
end
```
returns on each worker:

```
8  4
```

Create a new codistributed array distributed by rows from an existing one already distributed by columns:

```matlab
spmd
    X = redistribute(D, codistributor1d(1));
    size(getLocalPart(X))
end
```
returns on each worker:

```
2  16
```

Restoring the Full Array

You can restore a codistributed array to its undistributed form using the `gather` function. `gather` takes the segments of an array that reside on different workers and combines them into a replicated array on all workers, or into a single array on one worker.

Distribute a 4-by-10 array to four workers along the second dimension:

```matlab
spmd,  A = [11:20; 21:30; 31:40; 41:50],  end
A =
    11  12  13  14  15  16  17  18  19  20
    21  22  23  24  25  26  27  28  29  30
    31  32  33  34  35  36  37  38  39  40
```
spmd, D = codistributed(A), end

<table>
<thead>
<tr>
<th>WORKER 1</th>
<th>WORKER 2</th>
<th>WORKER 3</th>
<th>WORKER 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 12 13</td>
<td>14 15 16</td>
<td>17 18</td>
<td>19 20</td>
</tr>
<tr>
<td>21 22 23</td>
<td>24 25 26</td>
<td>27 28</td>
<td>29 30</td>
</tr>
<tr>
<td>31 32 33</td>
<td>34 35 36</td>
<td>37 38</td>
<td>39 40</td>
</tr>
<tr>
<td>41 42 43</td>
<td>44 45 46</td>
<td>47 48</td>
<td>49 50</td>
</tr>
</tbody>
</table>

spmd, size(getLocalPart(D)), end

Lab 1:
4 3
Lab 2:
4 3
Lab 3:
4 2
Lab 4:
4 2

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

spmd, X = gather(D), end

X =
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

spmd, size(X), end
4 10

Indexing into a Codistributed Array

While indexing into a nondistributed array is fairly straightforward, codistributed arrays require additional considerations. Each dimension of a nondistributed array is indexed within a range of 1 to the final subscript, which is represented in MATLAB by the end keyword. The length of any dimension can be easily determined using either the size or length function.
With codistributed arrays, these values are not so easily obtained. For example, the second segment of an array (that which resides in the workspace of worker 2) has a starting index that depends on the array distribution. For a 200-by-1000 array with a default distribution by columns over four workers, the starting index on worker 2 is 251. For a 1000-by-200 array also distributed by columns, that same index would be 51. As for the ending index, this is not given by using the end keyword, as end in this case refers to the end of the entire array; that is, the last subscript of the final segment. The length of each segment is also not given by using the length or size functions, as they only return the length of the entire array.

The MATLAB colon operator and end keyword are two of the basic tools for indexing into nondistributed arrays. For codistributed arrays, MATLAB provides a version of the colon operator, called codistributed.colon. This actually is a function, not a symbolic operator like colon.

**Note** When using arrays to index into codistributed arrays, you can use only replicated or codistributed arrays for indexing. The toolbox does not check to ensure that the index is replicated, as that would require global communications. Therefore, the use of unsupported variants (such as labindex) to index into codistributed arrays might create unexpected results.

**Example: Find a Particular Element in a Codistributed Array**

Suppose you have a row vector of 1 million elements, distributed among several workers, and you want to locate its element number 225,000. That is, you want to know what worker contains this element, and in what position in the local part of the vector on that worker. The globalIndices function provides a correlation between the local and global indexing of the codistributed array.

```matlab
D = rand(1,1e6,'distributed'); %Distributed by columns
spmd
  globalInd = globalIndices(D,2);
  pos = find(globalInd == 225e3);
  if ~isempty(pos)
    fprintf(...
      'Element is in position %d on worker %d\n', pos, labindex);
  end
end
```

If you run this code on a pool of four workers you get this result:
Lab 1:
Element is in position 225000 on worker 1.

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

Lab 2:
Element is in position 25000 on worker 2.

Notice if you use a pool of a different size, the element ends up in a different location on a different worker, but the same code can be used to locate the element.

2-Dimensional Distribution

As an alternative to distributing by a single dimension of rows or columns, you can distribute a matrix by blocks using '2dbc' or two-dimensional block-cyclic distribution. Instead of segments that comprise a number of complete rows or columns of the matrix, the segments of the codistributed array are 2-dimensional square blocks.

For example, consider a simple 8-by-8 matrix with ascending element values. You can create this array in an spmd statement, communicating job, or pmode. This example uses pmode for a visual display.

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

The result is the replicated array:

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

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

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

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

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

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

```
```

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

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

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

The first “row” of the lab grid is distributed to worker 1 and worker 2, but that contains only six of the eight columns of the original matrix. Therefore, the next two columns are
distributed to worker 1. This process continues until all columns in the first rows are distributed. Then a similar process applies to the rows as you proceed down the matrix, as shown in the following distribution scheme:

The diagram above shows a scheme that requires four overlays of the lab grid to accommodate the entire original matrix. The following pmode session shows the code and resulting distribution of data to each of the workers:
The following points are worth noting:

- '2dbc' distribution might not offer any performance enhancement unless the block size is at least a few dozen. The default block size is 64.
- The lab grid should be as close to a square as possible.
- Not all functions that are enhanced to work on '1d' codistributed arrays work on '2dbc' codistributed arrays.
Looping Over a Distributed Range (for-drang)
Note that the length of the `for` iteration and the length of the codistributed array `results` need to match in order to index into `results` within a `for drange` loop. This way, no communication is required between the workers. If `results` was simply a replicated array, as it would have been when running the original code in parallel, each worker would have assigned into its part of `results`, leaving the remaining parts of `results` 0. At the end, `results` would have been a variant, and without explicitly calling `labSend` and `labReceive` or `gcat`, there would be no way to get the total results back to one (or all) workers.

When using the `load` function, you need to be careful that the data files are accessible to all workers if necessary. The best practice is to use explicit paths to files on a shared file system.

Correspondingly, when using the `save` function, you should be careful to only have one worker save to a particular file (on a shared file system) at a time. Thus, wrapping the code in `if labindex == 1` is recommended.

Because `results` is distributed across the workers, this example uses `gather` to collect the data onto worker 1.

A worker cannot plot a visible figure, so the `print` function creates a viewable file of the plot.

**Codistributed Arrays in a for-drange Loop**

When a `for`-loop over a distributed range is executed in a communicating job, each worker performs its portion of the loop, so that the workers are all working simultaneously. Because of this, no communication is allowed between the workers while executing a `for drange` loop. In particular, a worker has access only to its partition of a codistributed array. Any calculations in such a loop that require a worker to access portions of a codistributed array from another worker will generate an error.

To illustrate this characteristic, you can try the following example, in which one for loop works, but the other does not.

At the pmode prompt, create two codistributed arrays, one an identity matrix, the other set to zeros, distributed across four workers.
D = eye(8, 8, codistributor())
E = zeros(8, 8, codistributor())

By default, these arrays are distributed by columns; that is, each of the four workers contains two columns of each array. If you use these arrays in a for-drange loop, any calculations must be self-contained within each worker. In other words, you can only perform calculations that are limited within each worker to the two columns of the arrays that the workers contain.

For example, suppose you want to set each column of array E to some multiple of the corresponding column of array D:

```matlab
for j = drange(1:size(D,2)); E(:,j) = j*D(:,j); end
```

This statement sets the j-th column of E to j times the j-th column of D. In effect, while D is an identity matrix with 1s down the main diagonal, E has the sequence 1, 2, 3, etc., down its main diagonal.

This works because each worker has access to the entire column of D and the entire column of E necessary to perform the calculation, as each worker works independently and simultaneously on two of the eight columns.

Suppose, however, that you attempt to set the values of the columns of E according to different columns of D:

```matlab
for j = drange(1:size(D,2)); E(:,j) = j*D(:,j+1); end
```

This method fails, because when j is 2, you are trying to set the second column of E using the third column of D. These columns are stored in different workers, so an error occurs, indicating that communication between the workers is not allowed.

**Restrictions**

To use for-drange on a codistributed array, the following conditions must exist:

- The codistributed array uses a 1-dimensional distribution scheme (not 2dbc).
- The distribution complies with the default partition scheme.
- The variable over which the for-drange loop is indexing provides the array subscript for the distribution dimension.
- All other subscripts can be chosen freely (and can be taken from for-loops over the full range of each dimension).
To loop over all elements in the array, you can use `for-drange` on the dimension of distribution, and regular `for`-loops on all other dimensions. The following example executes in an `spmd` statement running on a parallel pool of 4 workers:

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

    for ii = 1:6
        for jj = 1:8
            for kk = drange(1:12)
                PP(ii,jj,kk) = RR(ii,jj,kk) + labindex;
            end
        end
    end
end
```

To view the contents of the array, type:

```
PP
```
Run MATLAB Functions with Distributed Arrays

Hundreds of functions in MATLAB and other toolboxes are enhanced so that they operate on distributed arrays.

\[ D = \text{distributed}(\text{gallery('lehmer',n))}; \]
\[ e = \text{eig}(D); \]

If any of the input arguments to these distributed-enabled functions is a distributed array, their output arrays are distributed, unless returning MATLAB data is more appropriate (for example, \( \text{numel} \)).

Distributed arrays are well suited for large mathematical computations, such as large problems of linear algebra. You can also use distributed arrays for big data processing. For more information on distributing arrays, see “Distributing Arrays to Parallel Workers” on page 3-12.

Check Distributed Array Support in Functions

If a MATLAB function has distributed array support, you can consult additional distributed array usage information on its function page. See Distributed Arrays in the Extended Capabilities section at the end of the function page.

You can also browse distributed support for functions, and filter by product. On the Help bar, click Functions and select a product. In the function list, at the bottom of the left pane, select Distributed Arrays.

For information about updates to individual distributed-enabled functions, see the release notes.

To check support for special distributed data types, consult the following sections.

Support for Sparse Distributed Arrays

The following list shows functions that can help you work with sparse distributed arrays. In addition to this list, most element-wise functions in MATLAB also work for distributed arrays.
Support for Distributed calendarDuration Arrays

<table>
<thead>
<tr>
<th>calendarDuration</th>
<th>calquarters</th>
<th>cellstr</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>caldays</td>
<td>calweeks</td>
<td>datevec</td>
<td></td>
</tr>
<tr>
<td>calmonths</td>
<td>calyears</td>
<td>iscalendarDuration</td>
<td></td>
</tr>
</tbody>
</table>

Support for Distributed categorical Arrays

<table>
<thead>
<tr>
<th>categorical</th>
<th>iscategorical</th>
<th>isundefined</th>
<th>setcats</th>
</tr>
</thead>
<tbody>
<tr>
<td>addcats</td>
<td>iscategory</td>
<td>removecats</td>
<td></td>
</tr>
<tr>
<td>categories</td>
<td>isordinal</td>
<td>renamecats</td>
<td></td>
</tr>
<tr>
<td>countcats</td>
<td>isprotected</td>
<td>reordercats</td>
<td></td>
</tr>
</tbody>
</table>

Support for Distributed datetime Arrays

<table>
<thead>
<tr>
<th>datetime</th>
<th>exceltime</th>
<th>isweekend</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>between</td>
<td>hms</td>
<td>juliandate</td>
<td>timeofday</td>
</tr>
<tr>
<td>cellstr</td>
<td>hour</td>
<td>minute</td>
<td>tzoffset</td>
</tr>
<tr>
<td>datenum</td>
<td>isbetween</td>
<td>month</td>
<td>week</td>
</tr>
<tr>
<td>dateshift</td>
<td>isdatetime</td>
<td>posixtime</td>
<td>year</td>
</tr>
<tr>
<td>datevec</td>
<td>isdst</td>
<td>quarter</td>
<td>ymd</td>
</tr>
<tr>
<td>day</td>
<td>isnat</td>
<td>second</td>
<td>yyyymmdd</td>
</tr>
</tbody>
</table>

Support for Distributed duration Arrays

<table>
<thead>
<tr>
<th>duration</th>
<th>datevec</th>
<th>hours</th>
<th>minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellstr</td>
<td>days</td>
<td>isduration</td>
<td>seconds</td>
</tr>
<tr>
<td>datenum</td>
<td>hms</td>
<td>milliseconds</td>
<td>years</td>
</tr>
</tbody>
</table>
### Support for Distributed string Arrays

<table>
<thead>
<tr>
<th>string</th>
<th>erase</th>
<th>insertBefore</th>
<th>replaceBetween</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellstr</td>
<td>eraseBetween</td>
<td>ismissing</td>
<td>reverse</td>
</tr>
<tr>
<td>compose</td>
<td>extractAfter</td>
<td>isstring</td>
<td>startsWith</td>
</tr>
<tr>
<td>contains</td>
<td>extractBefore</td>
<td>lower</td>
<td>strip</td>
</tr>
<tr>
<td>count</td>
<td>extractBetween</td>
<td>pad</td>
<td>strlength</td>
</tr>
<tr>
<td>endsWith</td>
<td>insertAfter</td>
<td>replace</td>
<td>upper</td>
</tr>
</tbody>
</table>

### Support for Distributed table Arrays

<table>
<thead>
<tr>
<th>table</th>
<th>istable</th>
<th>table2array</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>standardizeMissing</td>
<td>tail</td>
</tr>
<tr>
<td>ismissing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Support for Distributed timetable Arrays

<table>
<thead>
<tr>
<th>head</th>
<th>properties</th>
<th>table2timetable</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>size</td>
<td>tail</td>
</tr>
<tr>
<td>istimetable</td>
<td>sort</td>
<td>timetable</td>
</tr>
<tr>
<td>ndims</td>
<td>subsref</td>
<td>timetable2table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>width</td>
</tr>
</tbody>
</table>
This chapter provides information you need for programming with Parallel Computing Toolbox software. Further details of evaluating functions in a cluster, programming independent jobs, and programming communicating jobs are covered in later chapters. This chapter describes features common to programming all kinds of jobs. The sections are as follows.

- “How Parallel Computing Products Run a Job” on page 6-2
- “Program a Job on a Local Cluster” on page 6-10
- “Specify Your Parallel Preferences” on page 6-12
- “Discover Clusters and Use Cluster Profiles” on page 6-15
- “Apply Callbacks to MJS Jobs and Tasks” on page 6-28
- “Job Monitor” on page 6-32
- “Programming Tips” on page 6-35
- “Control Random Number Streams” on page 6-40
- “Profiling Parallel Code” on page 6-45
- “HPC Challenge Benchmarks” on page 6-54
- “Troubleshooting and Debugging” on page 6-55
- “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
- “Use Tall Arrays on a Parallel Pool” on page 6-63
- “Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
- “Run mapreduce on a Parallel Pool” on page 6-71
- “Run mapreduce on a Hadoop Cluster” on page 6-75
- “Partition a Datastore in Parallel” on page 6-78
How Parallel Computing Products Run a Job

In this section...
“Overview” on page 6-2
“Toolbox and Server Components” on page 6-3
“Life Cycle of a Job” on page 6-7

Overview

Parallel Computing Toolbox and MATLAB Distributed Computing Server software let you solve computationally and data-intensive problems using MATLAB and Simulink on multicore and multiprocessor computers. Parallel processing constructs such as parallel for-loops and code blocks, distributed arrays, parallel numerical algorithms, and message-passing functions let you implement task-parallel and data-parallel algorithms at a high level in MATLAB without programming for specific hardware and network architectures.

A job is some large operation that you need to perform in your MATLAB session. A job is broken down into segments called tasks. You decide how best to divide your job into tasks. You could divide your job into identical tasks, but tasks do not have to be identical.

The MATLAB session in which the job and its tasks are defined is called the client session. Often, this is on the machine where you program MATLAB. The client uses Parallel Computing Toolbox software to perform the definition of jobs and tasks and to run them on a cluster local to your machine. MATLAB Distributed Computing Server software is the product that performs the execution of your job on a cluster of machines.

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

See the Glossary for definitions of the parallel computing terms used in this manual.
Basic Parallel Computing Setup

Toolbox and Server Components

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

MJS, Workers, and Clients

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

Each worker is given a task from the running job by the MJS, executes the task, returns the result to the MJS, and then is given another task. When all tasks for a running job have been assigned to workers, the MJS starts running the next job on the next available worker.
A MATLAB Distributed Computing Server software setup usually includes many workers that can all execute tasks simultaneously, speeding up execution of large MATLAB jobs. It is generally not important which worker executes a specific task. In an independent job, the workers evaluate tasks one at a time as available, perhaps simultaneously, perhaps not, returning the results to the MJS. In a communicating job, the workers evaluate tasks simultaneously. The MJS then returns the results of all the tasks in the job to the client session.

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

**Interactions of Parallel Computing Sessions**

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

Local Cluster

A feature of Parallel Computing Toolbox software is the ability to run a local cluster of workers on the client machine, so that you can run jobs without requiring a remote cluster or MATLAB Distributed Computing Server software. In this case, all the processing required for the client, scheduling, and task evaluation is performed on the same computer. This gives you the opportunity to develop, test, and debug your parallel applications before running them on your network cluster.

Third-Party Schedulers

As an alternative to using the MJS, you can use a third-party scheduler. This could be a Microsoft Windows HPC Server (including CCS), Platform LSF scheduler, PBS Pro® scheduler, TORQUE scheduler, or a generic scheduler.

Choosing Between a Third-Party Scheduler and an MJS

You should consider the following when deciding to use a third-party scheduler or the MATLAB job scheduler (MJS) for distributing your tasks:

- Does your cluster already have a scheduler?

  If you already have a scheduler, you may be required to use it as a means of controlling access to the cluster. Your existing scheduler might be just as easy to use as an MJS, so there might be no need for the extra administration involved.
• Is the handling of parallel computing jobs the only cluster scheduling management you need?

The MJS is designed specifically for MathWorks® parallel computing applications. If other scheduling tasks are not needed, a third-party scheduler might not offer any advantages.

• Is there a file sharing configuration on your cluster already?

The MJS can handle all file and data sharing necessary for your parallel computing applications. This might be helpful in configurations where shared access is limited.

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

When you use an MJS, worker processes usually remain running at all times, dedicated to their MJS. With a third-party scheduler, workers are run as applications that are started for the evaluation of tasks, and stopped when their tasks are complete. If tasks are small or take little time, starting a worker for each one might involve too much overhead time.

• Are there security concerns?

Your own scheduler might be configured to accommodate your particular security requirements.

• How many nodes are on your cluster?

If you have a large cluster, you probably already have a scheduler. Consult your MathWorks representative if you have questions about cluster size and the MJS.

• Who administers your cluster?

The person administering your cluster might have a preference for how jobs are scheduled.

• Do you need to monitor your job's progress or access intermediate data?

A job run by the MJS supports events and callbacks, so that particular functions can run as each job and task progresses from one state to another.

Components on Mixed Platforms or Heterogeneous Clusters

Parallel Computing Toolbox software and MATLAB Distributed Computing Server software are supported on Windows, UNIX®, and Macintosh operating systems. Mixed platforms are supported, so that the clients, MJS, and workers do not have to be on the same platform. Other limitations are described at System Requirements.
In a mixed-platform environment, system administrators should be sure to follow the proper installation instructions for the local machine on which you are installing the software.

**mdce Service**

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

The mdce service controls the worker and MJS sessions and recovers them when their host machines crash. If a worker or MJS machine crashes, when the mdce service starts up again (usually configured to start at machine boot time), it automatically restarts the MJS and worker sessions to resume their sessions from before the system crash. More information about the mdce service is available in the MATLAB Distributed Computing Server documentation.

**Components Represented in the Client**

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

When you create a job in the client session, the job actually exists in the MJS job storage location. The client session has access to the job through a *job object*. Likewise, tasks that you define for a job in the client session exist in the MJS data location, and you access them through *task objects*.

**Life Cycle of a Job**

When you create and run a job, it progresses through a number of stages. Each stage of a job is reflected in the value of the job object’s *State* property, which can be *pending*, *queued*, *running*, or *finished*. Each of these stages is briefly described in this section.

The figure below illustrates the stages in the life cycle of a job. In the MJS (or other scheduler), the jobs are shown categorized by their state. Some of the functions you use for managing a job are *createJob*, *submit*, and *fetchOutputs*. 
Stages of a Job

The following table describes each stage in the life cycle of a job.

<table>
<thead>
<tr>
<th>Job Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending</td>
<td>You create a job on the scheduler with the <code>createJob</code> function in your client session of Parallel Computing Toolbox software. The job's first state is pending. This is when you define the job by adding tasks to it.</td>
</tr>
<tr>
<td>Queued</td>
<td>When you execute the <code>submit</code> function on a job, the MJS or scheduler places the job in the queue, and the job's state is queued. The scheduler executes jobs in the queue in the sequence in which they are submitted, all jobs moving up the queue as the jobs before them are finished. You can change the sequence of the jobs in the queue with the <code>promote</code> and <code>demote</code> functions.</td>
</tr>
<tr>
<td>Running</td>
<td>When a job reaches the top of the queue, the scheduler distributes the job's tasks to worker sessions for evaluation. The job's state is now running. If more workers are available than are required for a job's tasks, the scheduler begins executing the next job. In this way, there can be more than one job running at a time.</td>
</tr>
<tr>
<td>Finished</td>
<td>When all of a job's tasks have been evaluated, the job is moved to the finished state. At this time, you can retrieve the results from all the tasks in the job with the function <code>fetchOutputs</code>.</td>
</tr>
<tr>
<td>Job Stage</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Failed</td>
<td>When using a third-party scheduler, a job might fail if the scheduler encounters an error when attempting to execute its commands or access necessary files.</td>
</tr>
<tr>
<td>Deleted</td>
<td>When a job’s data has been removed from its data location or from the MJS with the delete function, the state of the job in the client is deleted. This state is available only as long as the job object remains in the client.</td>
</tr>
</tbody>
</table>

Note that when a job is finished, its data remains in the MJS’s JobStorageLocation folder, even if you clear all the objects from the client session. The MJS or scheduler keeps all the jobs it has executed, until you restart the MJS in a clean state. Therefore, you can retrieve information from a job later or in another client session, so long as the MJS has not been restarted with the -clean option.

You can permanently remove completed jobs from the MJS or scheduler's storage location using the Job Monitor GUI or the delete function.
Program a Job on a Local Cluster

In some situations, you might need to define the individual tasks of a job, perhaps because they might evaluate different functions or have uniquely structured arguments. To program a job like this, the typical Parallel Computing Toolbox client session includes the steps shown in the following example.

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

1. Identify a cluster. Use `parallel.defaultClusterProfile` to indicate that you are using the local cluster; and use `parcluster` to create the object `c` to represent this cluster. (For more information, see “Create a Cluster Object” on page 7-4.)

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

2. Create a job. Create job `j` on the cluster. (For more information, see “Create a Job” on page 7-4.)

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

3. Create three tasks within the job `j`. Each task evaluates the `sum` of the array that is passed as an input argument. (For more information, see “Create Tasks” on page 7-5.)

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

4. Submit the job to the queue for evaluation. The scheduler then distributes the job’s tasks to MATLAB workers that are available for evaluating. The local cluster might now start MATLAB worker sessions. (For more information, see “Submit a Job to the Cluster” on page 7-5.)

   ```matlab
   submit(j);
   ```

5. Wait for the job to complete, then get the results from all the tasks of the job. (For more information, see “Fetch the Job Results” on page 7-6.)

   ```matlab
   wait(j)
   results = fetchOutputs(j)
   results =
   [2]
   [4]
   [6]
Delete the job. When you have the results, you can permanently remove the job from the scheduler's storage location.

delete(j)
Specify Your Parallel Preferences

You can access your parallel preferences in any of the following ways:

- On the **Home** tab in the **Environment** section, select **Parallel > Parallel Preferences**.
- Select the desktop pool indicator icon, and select **Parallel preferences**.
- In the command window, type **preferences**.

In the navigation tree of the Preferences dialog box, select **Parallel Computing Toolbox**.

You can control your parallel preference settings as follows:

- **Default Cluster** — Choose the cluster you want to use. The default cluster is **local**. For more information, see “Add and Modify Cluster Profiles” on page 6-20.
• **Preferred number of workers** — Specify the number of workers in your parallel pool. The actual pool size is limited by licensing, cluster size, and cluster profile settings on page 6-20. See “Pool Size and Cluster Selection” on page 2-70. For the local profile, do not choose a preferred number of workers larger than 512. See also “Add and Modify Cluster Profiles” on page 6-20. Check your access to cloud computing from the **Parallel > Discover Clusters** menu.

• **Automatically create a parallel pool** — Select this option to start a pool automatically (if a pool does not yet exist). Many functions can automatically start a parallel pool, including:
  - `parfor`
  - `spmd`
  - `distributed`
  - `Composite`
  - `parfeval`
  - `parfevalOnAll`
  - `gcp`
  - `mapreduce`
  - `mapreducer`

  If you have selected **Automatically create a parallel pool**, you do not need to open a pool manually using the `parpool` function. If a pool automatically opens, you can still access the pool object with `gcp`.

• **Shut down and delete a parallel pool** — To shut down a parallel pool automatically if the pool has been idle for the specified amount of time, use the `IdleTimeout` setting. If you use the pool (for example, using `parfor` or `parfeval`), the timeout counter is reset. When the timeout is about to expire, a tooltip on the desktop pool indicator warns you and allows you to reset the timer.

### See Also

**Related Examples**

- “Run MATLAB Functions with Automatic Parallel Support” on page 1-24
- “Scale up from Desktop to Cluster”
More About

- “Decide When to Use parfor” on page 2-2
- “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23
- “Add and Modify Cluster Profiles” on page 6-20
Discover Clusters and Use Cluster Profiles

In this section...

“Create and Manage Cluster Profiles” on page 6-16
“Discover Clusters” on page 6-17
“Create Cloud Cluster” on page 6-19
“Add and Modify Cluster Profiles” on page 6-20
“Import and Export Cluster Profiles” on page 6-24
“Edit Number of Workers and Cluster Settings” on page 6-25
“Use Your Cluster from MATLAB” on page 6-26

Parallel Computing Toolbox comes pre-configured with the cluster profile \texttt{local} for running parallel code on your local desktop machine.

Control parallel behavior using the \texttt{Parallel} menu on the MATLAB \texttt{Home} tab.

You can use the \texttt{Parallel} menu to:

- Discover other clusters running on your network or on Amazon EC2. Click \texttt{Parallel >> Discover Clusters}. For more information, see “Discover Clusters” on page 6-17.
- Create and manage cluster profiles using the Cluster Profile Manager. Click \texttt{Parallel >> Create and Manage Clusters}. For more information, see “Create and Manage Cluster Profiles” on page 6-16.
Create and Manage Cluster Profiles

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

- batch
- parpool
- parcluster

Manage cluster profiles using the Cluster Profile Manager. To open the Cluster Profile Manager, on the Home tab, in the Environment section, select Parallel > Create and Manage Clusters.

You can use the Cluster Profile Manager to:
• Discover other clusters running on your network or on Amazon® AWS®. For more information, see “Discover Clusters” on page 6-17.
• Create a cluster in the cloud, such as Amazon AWS. For more information, see “Create Cloud Cluster” on page 6-19.
• Add cluster profiles and modify their properties. For more information, see “Add and Modify Cluster Profiles” on page 6-20.
• Import and export cluster profiles. For more information, see “Import and Export Cluster Profiles” on page 6-24.
• Specify profile properties. For more information, see “Edit Number of Workers and Cluster Settings” on page 6-25.
• Validate that a cluster profile is ready for use in MATLAB.

Discover Clusters

You can let MATLAB discover clusters for you. Use either of the following techniques to discover those clusters which are available for you to use:

• On the Home tab in the Environment section, select Parallel > Discover Clusters
• In the Cluster Profile Manager, select Discover Clusters

This opens the Discover Clusters dialog box, where you can search for MATLAB Distributed Computing Server clusters:
If you select **On your network**, you see a new window. Select this option if your clusters use a MATLAB job scheduler (MJS) or Microsoft Windows HPC server. As clusters are discovered, they populate a list for your selection. If you already have a profile for any of the listed clusters, those profile names are included in the list. If you want to create a new profile for one of the discovered clusters, select the name of the cluster you want to use, and select **Next**. The subsequent dialog box lets you choose if you want to set the created profile as your default.

If you select **On Amazon EC2**, you search for clusters running on Amazon EC2. To access these clusters, you must provide your MathWorks Account login information.

**Requirements for Cluster Discovery**

Cluster discovery is supported only for MATLAB job schedulers (MJS), Microsoft Windows HPC Server, and Amazon EC2 cloud clusters. If you need to integrate your scheduler with MATLAB Distributed Computing Server, or create a cluster profile for a different supported scheduler, see “Getting Started with MATLAB Distributed Computing Server” (MATLAB Distributed Computing Server). The following requirements apply to cluster discovery:

- MJS — MJS clusters support two different means of discovery:
• Multicast: The discover clusters functionality uses the multicast networking protocol from the client to search for head nodes where an MJS is running. This requires that the multicast networking protocol is enabled and working on the network that connects the MJS head nodes (where the schedulers are running) and the client machines. This form of discovery might be limited to the client local subnet, and therefore not always able to discover an MJS elsewhere in your network.

• DNS SRV: An alternative discovery technique is to search for clusters by DNS service records.

The Domain Name System (DNS) is a standard for identifying host names with IP addresses, either on the Internet or in a private network. Using DNS allows discovery of MJS clusters by identifying specific hosts rather than broadcasting across your network.

A DNS service (SRV) record defines the location of hosts and ports of services, such as those related to the clusters you want to discover. Your system administrator creates DNS SRV records in your organization’s DNS infrastructure. For a description of the required record, and validation information, see “DNS SRV Record” (MATLAB Distributed Computing Server).

• HPC Server — The discover clusters functionality uses Active Directory Domain Services to discover head nodes. HPC Server head nodes are added to the Active Directory during installation of the HPC Server software.

• Amazon EC2 — The discover clusters functionality requires a working network connection between the client and the Cloud Center web services running in mathworks.com.

Create Cloud Cluster

You can create clusters in cloud services, such as Amazon AWS, directly from the Cluster Profile Manager. In the Cluster Profile Manager, select Create Cloud Cluster. After you sign up with your MathWorks account and complete the requirements steps, you can then create a cloud cluster and configure parameters, such as the number of machines or the number of workers per machine. For more information on each of the available parameters, see “Create a Cloud Cluster” (MathWorks Cloud Center). When you complete all the steps, MATLAB creates a new cluster profile for you. You can modify its properties from the Cluster Profile Manager.
If you want to manage your licenses, test cloud connectivity or manage your cloud clusters in MathWorks Cloud Center, see the CLOUD section, in the Cluster Profile Manager toolstrip.

**Add and Modify Cluster Profiles**

With the Cluster Profile Manager, you can add a cluster profile for a MATLAB job scheduler or a third-party scheduler. If you need to set up your cluster for use with MATLAB, see “Getting Started with MATLAB Distributed Computing Server” (MATLAB Distributed Computing Server).

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

Suppose you want to create a profile to set several properties for jobs to run in a MATLAB job scheduler (MJS) cluster. The following example illustrates a possible workflow, where you create two profiles differentiated only by the number of workers they use.

1. In the Cluster Profile Manager, select **Add Cluster Profile** > **MATLAB Job Scheduler (MJS)**. This specifies that you want a new profile for an MJS cluster.
This creates and displays a new profile, called MJSProfile1.

2 Double-click the new profile name in the listing, and modify the profile name to be MyMJSProfile1.

3 Select **Edit** in the tool strip so that you can set your profile property values.

In the Description field, enter the text **MJS with 4 workers**, as shown in the following figure. Enter the host name for the machine on which the MJS is running, and the name of the MJS. If you are entering information for an actual MJS already running on your network, enter the actual names. If you are unsure about the MJS names and locations on your network, ask your system administrator for help.
Note If the MJS is using a nondefault BASE_PORT setting as defined in the mdce_def file, the Host property in the cluster profile must be appended with this BASE_PORT number. For example, MJS-Host:40000.

4 Scroll down to the Workers section, and for the Range of number of workers, enter the two-element vector [4 4]. This specifies that jobs using this profile require at least four workers and no more than four workers. Therefore, a job using this profile runs on exactly four workers, even if it has to wait until four workers are available before starting.
You might want to edit other properties depending on your particular network and cluster situation.

5 Select **Done** to save the profile settings.

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

1 In the Cluster Profile Manager, right-click the profile name **MyMJSProfile1** in the list and select **Duplicate**.

   This creates a duplicate profile with a name based on the original profile name appended with _Copy_.

2 Double-click the new profile name and edit its name to be **MyMJSProfile2**.

3 Select **Edit** to allow you to change the profile property values.
4 Edit the description field to change its text to **MJS with any workers**.
5 Scroll down to the Workers section, and for the Range of number of workers, clear the [4 4] and leave the field blank.
6 Select **Done** to save the profile settings and to close the properties editor.

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

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

You can see examples of profiles for different kinds of supported schedulers in the MATLAB Distributed Computing Server installation instructions at “Configure Your Cluster” (MATLAB Distributed Computing Server).

### Import and Export Cluster Profiles

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

To export a cluster profile:

1. In the Cluster Profile Manager, select (highlight) the profile you want to export.
2. Select **Export > Export**. (Alternatively, you can right-click the profile in the listing and select **Export**.)

   If you want to export all your profiles to a single file, select **Export > Export All**

3. In the Export profiles to file dialog box, specify a location and name for the file. The default file name is the same as the name of the profile it contains, with a `.settings` extension appended; you can alter the names if you want to.

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

1. In the Cluster Profile Manager, select **Import**.
2. In the Import profiles from file dialog box, browse to find the `.settings` file for the profile you want to import. Select the file and select **Open**.

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

You can also export and import profiles programmatically with the parallel.exportProfile and parallel.importProfile functions.

**Export Profiles for MATLAB Compiler**

You can use an exported profile with MATLAB Compiler and MATLAB Compiler SDK to identify cluster setup information for running compiled applications on a cluster. For example, the setmcruerdata function can use the exported profile file name to set the value for the key ParallelProfile. For more information and examples of deploying parallel applications, see “Pass Parallel Computing Toolbox Profile at Run Time” (MATLAB Compiler), and “Use Parallel Computing Toolbox in Deployed Applications” (MATLAB Compiler SDK).

A compiled application has the same default profile and the same list of alternative profiles that the compiling user had when the application was compiled. This means that in many cases the profile file is not needed, as might be the case when using the local profile for local workers. If an exported file is used, the first profile in the file becomes the default when imported. If any of the imported profiles have the same name as any of the existing profiles, they are renamed during import (though their names in the file remain unchanged).

**Edit Number of Workers and Cluster Settings**

After you create a cluster profile, you can specify the number of workers and other profile properties:

- **NumWorkers**: the number of workers to start a pool. The actual pool size might be limited by licensing, cluster size, and cluster profile settings. See “Pool Size and Cluster Selection” on page 2-70

- **NumThreads**: the number of computational threads to use on each worker. You can change NumThreads, so that your workers can run in multithreaded mode and use all the cores on your cluster. This allows you to increase the number of computational threads NumThreads on each worker, without increasing the number of workers NumWorkers. If you have more cores available, increase NumThreads to take full advantage of the built-in parallelism provided by the multithreaded nature of many of the underlying MATLAB libraries. For details, see Run MATLAB on multicore and multiprocessor machines.
**Note** Do not increase the number of threads across all workers on a machine to exceed the number of physical cores. In other words, make sure that NumWorkers \(\times\) NumThreads \(\leq\) number of physical cores on your machine. Otherwise you might have reduced performance.

## Use Your Cluster from MATLAB

To run parallel language functions, such as `parpool` or `batch`, on a cluster, set the cluster profile as default, or use cluster objects.

### Specify Default Cluster

To set a cluster profile as the default, use one of the following ways:

- On the **Home** tab in the **Environment** section, select **Parallel > Select a Default Cluster**, and from there, all your profiles are available. The default profile is indicated. You can select any profile in the list as the default.
- The Cluster Profile Manager indicates which is the default profile. You can select any profile in the list, then select **Set as Default**.
- You can get or set the default profile programmatically by using the `parallel.defaultClusterProfile` function. The following sets of commands achieve the same thing:

  ```matlab
  parallel.defaultClusterProfile('MyMJSProfile1')
  parpool
  
  or
  
  parpool('MyMJSProfile1')
  ```

### Specify Cluster Programmatically (parcluster)

The `parcluster` function creates a cluster object in your workspace according to the specified profile. The profile identifies a particular cluster and applies property values. For example,

```matlab
c = parcluster('MyMJSProfile1')
```

This command finds the cluster defined by the settings of the profile named `MyMJSProfile1` and sets property values on the cluster object based on settings in the profile. Use a cluster object in functions such as `parpool` or `batch`. By applying different
profiles, you can alter your cluster choices without changing your MATLAB application code.

**See Also**

batch | createJob | parallel.exportProfile | parallel.importProfile | parcluster | parpool | setmcruserdata

**Related Examples**

- “Run Code on Parallel Pools” on page 2-67
- “Scale up from Desktop to Cluster”
- “Pass Parallel Computing Toolbox Profile at Run Time” (MATLAB Compiler)
- “Use Parallel Computing Toolbox in Deployed Applications” (MATLAB Compiler SDK)
- “Verify Network Communications for Cluster Discovery” (MATLAB Distributed Computing Server)

**More About**

- “Getting Started with MATLAB Distributed Computing Server” (MATLAB Distributed Computing Server)
- “Clusters and Clouds”

**External Websites**

- https://www.mathworks.com/help/cloudcenter/
- https://www.mathworks.com/licensecenter
Apply Callbacks to MJS Jobs and Tasks

The MATLAB job scheduler (MJS) has the ability to trigger callbacks in the client session whenever jobs or tasks in the MJS cluster change to specific states.

Client objects representing jobs and tasks in an MJS cluster include the following properties:

<table>
<thead>
<tr>
<th>Callback Property</th>
<th>Object</th>
<th>Cluster Profile Manager Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueuedFcn</td>
<td>Job only</td>
<td>JobQueuedFcn</td>
<td>Specifies the function to execute in the client when a job is submitted to the MJS queue</td>
</tr>
<tr>
<td>RunningFcn</td>
<td>Job or task</td>
<td>JobRunningFcn, TaskRunningFcn</td>
<td>Specifies the function to execute in the client when a job or task begins its execution</td>
</tr>
<tr>
<td>FinishedFcn</td>
<td>Job or task</td>
<td>JobFinishedFcn, TaskFinishedFcn</td>
<td>Specifies the function to execute in the client when a job or task completes its execution</td>
</tr>
</tbody>
</table>

You can set each of these properties to any valid MATLAB callback value in the Cluster Profile Manager, see the table and “Add and Modify Cluster Profiles” on page 6-20. The callback follows the same behavior for Handle Graphics®, passing into the callback function the object (job or task) that makes the call and an empty argument of event data.

These properties apply only in the client MATLAB session in which they are set. Later sessions that access the same job or task objects do not inherit the settings from previous sessions. You can apply the properties to existing jobs and tasks at the command-line, but the cluster profile settings apply only at the time these objects are first created.
Note  The callback properties are available only when using an MJS cluster.

Example 6.1. Create Callbacks at the Command Line

This example shows how to create job and task callbacks at the client session command line.

Create and save a callback function clientTaskCompleted.m on the path of the MATLAB client, with the following content:

```matlab
function clientTaskCompleted(task,eventdata)
    disp(["Finished task: ' num2str(task.ID)]
```

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

```matlab
c = parcluster('MyMJS');
j = createJob(c,'Name','Job_52a');
j.QueuedFcn = @(job,eventdata) disp([job.Name ' now ' job.State]);
j.RunningFcn = @(job,eventdata) disp([job.Name ' now ' job.State]);
j.FinishedFcn = @(job,eventdata) disp([job.Name ' now ' job.State]);
```

Create a task whose FinishedFcn is a function handle to the separate function.

```matlab
createTask(j,@rand,1,{2,4}, ...
    'FinishedFcn',@clientTaskCompleted);
```

Run the job and note the output messages from both the job and task callbacks.

```matlab
submit(j)
```

```
Job_52a now queued
Job_52a now running
Finished task: 1
Job_52a now finished
```

To use the same callbacks for any jobs and tasks on a given cluster, you should set these properties in the cluster profile. For details on editing profiles in the profile manager, see “Discover Clusters and Use Cluster Profiles” on page 6-15. These property settings apply to any jobs and tasks created using a cluster derived from this profile. The sequence is important, and must occur in this order:

1   Set the callback property values for the profile in the profile manager.
2   Use the cluster profile to create a cluster object in MATLAB.
3 Use the cluster object to create jobs and then tasks.

**Example 6.2. Set Callbacks in a Cluster Profile**

This example shows how to set several job and task callback properties using the profile manager.

Edit your MJS cluster profile in the profile manager so that you can set the callback properties to the same values in the previous example. The saves profile looks like this:

Create and save a callback function `clientTaskCompleted.m` on the path of the MATLAB client, with the following content. (If you created this function for the previous example, you can use that.)

```matlab
function clientTaskCompleted(task,eventdata)
    disp(['Finished task: ' num2str(task.ID)])
end
```

Create objects for the cluster, job, and task. Then submit the job. All the callback properties are set from the profile when the objects are created.

```matlab
c = parcluster('MyMJS');
j = createJob(c,'Name','Job_52a');
createTask(j,@rand,1,{2,4});
submit(j)
```
Tips

- You should avoid running code in your callback functions that might cause conflicts. For example, if every task in a job has a callback that plots its results, there is no guarantee to the order in which the tasks finish, so the plots might overwrite each other. Likewise, the FinishFcn callback for a job might be triggered to start before the FinishFcn callbacks for all its tasks are complete.

- Submissions made with `batch` use applicable job and task callbacks. Parallel pools can trigger job callbacks defined by their cluster profile.
Job Monitor

The Job Monitor displays the jobs in the queue for the scheduler determined by your selection of a cluster profile. Open the Job Monitor from the MATLAB desktop on the Home tab in the Environment section, by selecting Parallel > Monitor Jobs.

The job monitor lists all the jobs that exist for the cluster specified in the selected profile. You can choose any one of your profiles (those available in your current session Cluster Profile Manager), and whether to display jobs from all users or only your own jobs.

Typical Use Cases

The Job Monitor lets you accomplish many different goals pertaining to job tracking and queue management. Using the Job Monitor, you can:

- Discover and monitor all jobs submitted by a particular user
- Determine the status of a job
- Determine the cause of errors in a job
- Delete old jobs you no longer need
- Create a job object in MATLAB for access to a particular job in the queue
Manage Jobs Using the Job Monitor

Using the Job Monitor you can manage the listed jobs for your cluster. Right-click on any job in the list, and select any of the following options from the context menu. The available options depend on the type of job.

- **Cancel** — Stops a running job and changes its state to 'finished'. If the job is pending or queued, the state changes to 'finished' without its ever running. This is the same as the command-line `cancel` function for the job.
- **Delete** — Deletes the job data and removes the job from the queue. This is the same as the command-line `delete` function for the job. Also closes and deletes an interactive pool job.
- **Show details** — This displays detailed information about the job in the Command Window.
- **Show errors** — This displays all the tasks that generated an error in that job, with their error properties.
- **Fetch outputs** — This collects all the task output arguments from the job into the client workspace.

Identify Task Errors Using the Job Monitor

Because the Job Monitor indicates if a job had a run-time error, you can use it to identify the tasks that generated the errors in that job. For example, the following script generates an error because it attempts to perform a matrix inverse on a vector:

```
A = [2 4 6 8];
B = inv(A);
```

If you save this script in a file named `invert_me.m`, you can try to run the script as a batch job on the default cluster:

```
bash
batch('invert_me')
```

When updated after the job runs, the Job Monitor includes the job created by the `batch` command, with an error icon (⚠️) for this job. Right-click the job in the list, and select **Show Errors**. For all the tasks with an error in that job, the task information, including properties related to the error, display in the MATLAB command window:

```
Task ID 1 from Job ID 2 Information
```

---

6-33
State : finished
Function : @parallel.internal.cluster.executeScript
Running Duration : 0 days 0h 0m 1s

- Task Result Properties
  ErrorIdentifier : MATLAB:square
  ErrorMessage : Matrix must be square.
  Error Stack : invert_me (line 2)
Programming Tips

Program Development Guidelines

When writing code for Parallel Computing Toolbox software, you should advance one step at a time in the complexity of your application. Verifying your program at each step prevents your having to debug several potential problems simultaneously. If you run into any problems at any step along the way, back up to the previous step and reverify your code.

The recommended programming practice for distributed or parallel computing applications is

1. **Run code normally on your local machine.** First verify all your functions so that as you progress, you are not trying to debug the functions and the distribution at the same time. Run your functions in a single instance of MATLAB software on your local computer. For programming suggestions, see “Techniques to Improve Performance” (MATLAB).

2. **Decide whether you need an independent or communicating job.** If your application involves large data sets on which you need simultaneous calculations performed, you might benefit from a communicating job with distributed arrays. If your application involves looped or repetitive calculations that can be performed independently of each other, an independent job might be appropriate.
3 Modify your code for division. Decide how you want your code divided. For an independent job, determine how best to divide it into tasks; for example, each iteration of a for-loop might define one task. For a communicating job, determine how best to take advantage of parallel processing; for example, a large array can be distributed across all your workers.

4 Use pmode to develop parallel functionality. Use pmode with the local scheduler to develop your functions on several workers in parallel. As you progress and use pmode on the remote cluster, that might be all you need to complete your work.

5 Run the independent or communicating job with a local scheduler. Create an independent or communicating job, and run the job using the local scheduler with several local workers. This verifies that your code is correctly set up for batch execution, and in the case of an independent job, that its computations are properly divided into tasks.

6 Run the independent job on only one cluster node. Run your independent job with one task to verify that remote distribution is working between your client and the cluster, and to verify proper transfer of additional files and paths.

7 Run the independent or communicating job on multiple cluster nodes. Scale up your job to include as many tasks as you need for an independent job, or as many workers as you need for a communicating job.

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

Current Working Directory of a MATLAB Worker

The current directory of a MATLAB worker at the beginning of its session is

CHECKPOINTBASE\HOSTNAME_WORKERNAME_mlworker_log\work

where CHECKPOINTBASE is defined in the mdce_def file, HOSTNAME is the name of the node on which the worker is running, and WORKERNAME is the name of the MATLAB worker session.

For example, if the worker named worker22 is running on host nodeA52, and its CHECKPOINTBASE value is C:\TEMP\MDCE\Checkpoint, the starting current directory for that worker session is

C:\TEMP\MDCE\Checkpoint\nodeA52_worker22_mlworker_log\work
Writing to Files from Workers

When multiple workers attempt to write to the same file, you might end up with a race condition, clash, or one worker might overwrite the data from another worker. This might be likely to occur when:

- There is more than one worker per machine, and they attempt to write to the same file.
- The workers have a shared file system, and use the same path to identify a file for writing.

In some cases an error can result, but sometimes the overwriting can occur without error. To avoid an issue, be sure that each worker or parfor iteration has unique access to any files it writes or saves data to. There is no problem when multiple workers read from the same file.

Saving or Sending Objects

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

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

Also, system objects (e.g., Java classes, .NET classes, shared libraries, etc.) that are loaded, imported, or added to the Java search path in the MATLAB client, are not available on the workers unless explicitly loaded, imported, or added on the workers, respectively. Other than in the task function code, typical ways of loading these objects might be in taskStartup, jobStartup, and in the case of workers in a parallel pool, in poolStartup and using pctRunOnAll.

Using clear functions

Executing clear functions
clears all Parallel Computing Toolbox objects from the current MATLAB session. They still remain in the MJS. For information on recreating these objects in the client session, see “Recover Objects” on page 7-14.

Running Tasks That Call Simulink Software

The first task that runs on a worker session that uses Simulink software can take a long time to run, as Simulink is not automatically started at the beginning of the worker session. Instead, Simulink starts up when first called. Subsequent tasks on that worker session will run faster, unless the worker is restarted between tasks.

Using the pause Function

On worker sessions running on Macintosh or UNIX operating systems, `pause(Inf)` returns immediately, rather than pausing. This is to prevent a worker session from hanging when an interrupt is not possible.

Transmitting Large Amounts of Data

Operations that involve transmitting many objects or large amounts of data over the network can take a long time. For example, getting a job's Tasks property or the results from all of a job's tasks can take a long time if the job contains many tasks. See also “Attached Files Size Limitations” on page 6-55.

Interrupting a Job

Because jobs and tasks are run outside the client session, you cannot use Ctrl+C (^C) in the client session to interrupt them. To control or interrupt the execution of jobs and tasks, use such functions as cancel, delete, demote, promote, pause, and resume.

Speeding Up a Job

You might find that your code runs slower on multiple workers than it does on one desktop computer. This can occur when task startup and stop time is significant relative to the task run time. The most common mistake in this regard is to make the tasks too small, i.e., too fine-grained. Another common mistake is to send large amounts of input or output data with each task. In both of these cases, the time it takes to transfer data and
initialize a task is far greater than the actual time it takes for the worker to evaluate the task function.
Control Random Number Streams

By default, each worker in a cluster working on the same job has a unique random number stream. This example uses two workers in a parallel pool to show they generate unique random number sequences.

```matlab
p = parpool(2);
spmd
    R = rand(1,4); % Different on each worker
end
R{1},R{2}
```

If you need all workers to generate the same sequence of numbers, you can seed their generators all the same.

```matlab
p = parpool(2);
spmd
    rng('Twister'); % Default seed 0.
    R = rand(1,4); % Same on all workers
end
R{1},R{2}
```

If you need all workers to generate the same sequence of numbers, you can seed their generators all the same.
Note: Because `rng('shuffle')` seeds the random number generator based on the current time, you should not use this command to set the random number stream on different workers if you want to assure independent streams. This is especially true when the command is sent to multiple workers simultaneously, such as inside a `parfor`, `spmd`, or a communicating job. For independent streams on the workers, use the default behavior; or if that is not sufficient for your needs, consider using a unique substream on each worker.

For instructions on how to generate a reproducible set of random numbers in a `parfor`-loop, see “Repeat Random Numbers in parfor-Loops” on page 2-73.

**Client and Workers**

By default, the MATLAB client and MATLAB workers use different random number generators, even if the workers are part of a local cluster on the same machine with the client. For the client, the default is the Mersenne Twister generator ('Twister'), and for the workers the default is the Combined Multiple Recursive generator ('CombRecursive' or 'mrg32k3a'). If it is necessary to generate the same stream of numbers in the client and workers, you can set one to match the other.

For example, you might run a script as a batch job on a worker, and need the same generator or sequence as the client. Suppose you start with a script file named `randScript1.m` that contains the line:

```matlab
R = rand(1,4);
```

You can run this script in the client, and then as a batch job on a worker. Notice that the default generated random number sequences in the results are different.

```matlab
randScript1; % In client
R
```

```matlab
R =
0.8147    0.9058    0.1270    0.9134
```

```matlab
parallel.defaultClusterProfile('local')
c = parcluster();
j = batch(c,'randScript1'); % On worker
wait(j);load(j);
R
```
R =
    0.3246    0.6618    0.6349    0.6497

For identical results, you can set the client and worker to use the same generator and seed. Here the file `randScript2.m` contains the following code:

```matlab
rng(1,'CombRecursive');
R = rand(1,4);
```

Now, run the new script in the client and on a worker:

```matlab
randScript2; % In client
R

R =
    0.4957    0.2243    0.2073    0.6823
```

```matlab
j = batch(c,'randScript2'); % On worker
wait(j); load(j);
R

R =
    0.4957    0.2243    0.2073    0.6823
```

## Client and GPU

By default MATLAB clients use different random generators than code running on a GPU. GPUs are more like workers in this regard, and use the Combined Multiple Recursive generator (`'CombRecursive'` or `'mrg32k3a'`) unless otherwise specified.

This example shows a default generation of random numbers comparing CPU and GPU in a fresh session.

```matlab
Rc = rand(1,4)

Rc =
    0.8147    0.9058    0.1270    0.9134

Rg = rand(1,4,'gpuArray')

Rg =
    0.7270    0.4522    0.9387    0.2360
```
Be aware that the GPU supports only three generators ('CombRecursive', 'Philox', and 'Threefry'). The following table lists the algorithms for these generators and their properties.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Generator</th>
<th>Multiple Stream and Substream Support</th>
<th>Approximate Period In Full Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>'CombRecursive' or 'mrg32k3a'</td>
<td>Combined multiple recursive generator</td>
<td>Yes</td>
<td>$2^{191}$ ($2^{63}$ streams of length $2^{127}$)</td>
</tr>
<tr>
<td>'Philox' or 'Philox4x32_10'</td>
<td>Philox 4x32 generator with 10 rounds</td>
<td>Yes</td>
<td>$2^{193}$ ($2^{64}$ streams of length $2^{129}$)</td>
</tr>
<tr>
<td>'Threefry' or 'Threefry4x64_20'</td>
<td>Threefry 4x64 generator with 20 rounds</td>
<td>Yes</td>
<td>$2^{514}$ ($2^{256}$ streams of length $2^{258}$)</td>
</tr>
</tbody>
</table>

None of these is the default client generator for the CPU. To generate the same sequence on CPU and GPU, you must set the generator to one supported on both. 'CombRecursive', 'Philox', and 'Threefry' are all supported for both CPU and GPU calculations.

```matlab
sc = rng(1, 'CombRecursive');
Rc = rand(1,4)
Rc =  
0.4957    0.2243    0.2073    0.6823
sg = gpurng(1, 'CombRecursive');
Rg = rand(1,4, 'gpuArray')
Rg =  
0.4957    0.2243    0.2073    0.6823
```

For normally distributed random numbers created by `randn`, CPU code by default uses a random stream with a 'NormalTransform' setting of 'Ziggurat'. By default, GPU code uses a setting of 'Inversion' with the generator 'CombRecursive', and the 'BoxMuller' transform for the 'Philox' and 'Threefry' generators.

You can set the same generators and transforms on the CPU and the GPU to get the same `randn` sequences. The only transform supported on both the CPU and GPU is the 'Inversion' transform.

```matlab
sc = RandStream('CombRecursive','NormalTransform','Inversion','Seed',1);
RandStream.setGlobalStream(sc)
```
sg = parallel.gpu.RandStream('CombRecursive','NormalTransform','Inversion','Seed',1);
parallel.gpu.RandStream.setGlobalStream(sg);

Rc = randn(1,4)
Rc =
   -0.0108  -0.7577  -0.8159   0.4742

Rg = randn(1,4,'gpuArray')
Rg =
   -0.0108  -0.7577  -0.8159   0.4742

**Worker CPU and Worker GPU**

Code running on a worker’s CPU uses the same generator to create random numbers as code running on a worker’s GPU, but they do not share the same stream. You can use a common seed to generate the same sequence of numbers, as shown in this example, where each worker creates the same sequence on GPU and CPU, but different from the sequence on the other worker.

p = parpool(2);
spmd
    rng(labindex,'CombRecursive');
    Rc = rand(1,4)
    gpurng(labindex,'CombRecursive');
    Rg = rand(1,4,'gpuArray')
end
delete(p)

For normally distributed random numbers from `randn`, by default a worker CPU uses a `NormalTransform` setting of `Ziggurat` while a worker GPU uses a setting of `Inversion`. You can set them both to use `Inversion` if you need the same sequence from CPU and GPU.
Profiling Parallel Code

**In this section...**

“Introduction” on page 6-45
“Collecting Parallel Profile Data” on page 6-45
“Viewing Parallel Profile Data” on page 6-46

**Introduction**

The parallel profiler provides an extension of the `profile` command and the profile viewer specifically for communicating jobs, to enable you to see how much time each worker spends evaluating each function and how much time communicating or waiting for communications with the other workers. Before using the parallel profiler, familiarize yourself with the standard profiler and its views, as described in “Profile to Improve Performance” (MATLAB).

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

**Collecting Parallel Profile Data**

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

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

```
mpiprofile on
```

Now the profiler is collecting information about the execution of code on each worker and the communications between the workers. Such information includes:

- Execution time of each function on each worker
- Execution time of each line of code in each function
- Amount of data transferred between each worker
• Amount of time each worker spends waiting for communications

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

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

P>> mpiprofile status

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

**Viewing Parallel Profile Data**

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

P>> mpiprofile viewer

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

pmode start local 4

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

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

The last command opens the Profiler window, first showing the Parallel Profile Summary (or function summary report) for worker (lab) 1.
The function summary report displays the data for each function executed on a worker in sortable columns with the following headers:

<table>
<thead>
<tr>
<th>Column Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls</td>
<td>How many times the function was called on this worker</td>
</tr>
<tr>
<td>Total Time</td>
<td>The total amount of time this worker spent executing this function</td>
</tr>
<tr>
<td>Self Time</td>
<td>The time this worker spent inside this function, not within children or local functions</td>
</tr>
<tr>
<td>Total Comm Time</td>
<td>The total time this worker spent transferring data with other workers, including waiting time to receive data</td>
</tr>
<tr>
<td>Self Comm Waiting Time</td>
<td>The time this worker spent during this function waiting to receive data from other workers</td>
</tr>
<tr>
<td>Total Interlab Data</td>
<td>The amount of data transferred to and from this worker for this function</td>
</tr>
<tr>
<td>Computation Time Ratio</td>
<td>The ratio of time spent in computation for this function vs. total time (which includes communication time) for this function</td>
</tr>
<tr>
<td>Column Header</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Total Time Plot</td>
<td>Bar graph showing relative size of Self Time, Self Comm Waiting Time, and Total Time for this function on this worker</td>
</tr>
</tbody>
</table>

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

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

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

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

The following figure shows a summary of all the functions executed during the profile collection time. The **Manual Comparison Selection of max Time Aggregate** means that data is considered from all the workers for all functions to determine which worker spent the maximum time on each function. Next to each function's name is the worker that took the longest time to execute that function. The other columns list the data from that worker.
The next figure shows a summary report for the workers that spend the most versus least time for each function. A Manual Comparison Selection of max Time Aggregate against min Time >0 Aggregate generated this summary. Both aggregate settings indicate that the profiler should consider data from all workers for all functions, for both maximum and minimum. This report lists the data for codistributed.mtimes from workers 3 and 1, because they spent the maximum and minimum times on this function. Similarly, other functions are listed.
Select a function name in the summary listing of a comparison to get a detailed comparison. The detailed comparison for `codistributed.mtimes` looks like this, displaying line-by-line data from both workers:
To see plots of communication data, select **Plot All PerLab Communication** in the **Show Figures** menu. The top portion of the plot view report plots how much data each worker receives from each other worker for all functions.

To see only a plot of interworker communication times, select **Plot CommTimePerLab** in the **Show Figures** menu.
Plots like those in the previous two figures can help you determine the best way to balance work among your workers, perhaps by altering the partition scheme of your codistributed arrays.
HPC Challenge Benchmarks

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

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Attached Files Size Limitations” on page 6-55</td>
</tr>
<tr>
<td>“File Access and Permissions” on page 6-55</td>
</tr>
<tr>
<td>“No Results or Failed Job” on page 6-57</td>
</tr>
<tr>
<td>“Connection Problems Between the Client and MJS” on page 6-57</td>
</tr>
<tr>
<td>“SFTP Error: Received Message Too Long” on page 6-58</td>
</tr>
</tbody>
</table>

**Attached Files Size Limitations**

The combined size of all attached files for a job is limited to 4 GB.

**File Access and Permissions**

**Ensuring That Workers on Windows Operating Systems Can Access Files**

By default, a worker on a Windows operating system is installed as a service running as `LocalSystem`, so it does not have access to mapped network drives.

Often a network is configured to not allow services running as `LocalSystem` to access UNC or mapped network shares. In this case, you must run the mdce service under a different user with rights to log on as a service. See the section “Set the User” (MATLAB Distributed Computing Server) in the MATLAB Distributed Computing Server System Administrator's Guide.

**Task Function Is Unavailable**

If a worker cannot find the task function, it returns the error message

```
Error using ==> feval
    Undefined command/function 'function_name'.
```

The worker that ran the task did not have access to the function `function_name`. One solution is to make sure the location of the function’s file, `function_name.m`, is included in the job’s `AdditionalPaths` property. Another solution is to transfer the function file to the worker by adding `function_name.m` to the `AttachedFiles` property of the job.
Load and Save Errors

If a worker cannot save or load a file, you might see the error messages

```plaintext
??? Error using ==> save
Unable to write file myfile.mat: permission denied.
??? Error using ==> load
Unable to read file myfile.mat: No such file or directory.
```

In determining the cause of this error, consider the following questions:

- What is the worker’s current folder?
- Can the worker find the file or folder?
- What user is the worker running as?
- Does the worker have permission to read or write the file in question?

Tasks or Jobs Remain in Queued State

A job or task might get stuck in the queued state. To investigate the cause of this problem, look for the scheduler’s logs:

- Platform LSF schedulers might send emails with error messages.
- Microsoft Windows HPC Server (including CCS), LSF®, PBS Pro, and TORQUE save output messages in a debug log. See the `getDebugLog` reference page.
- If using a generic scheduler, make sure the submit function redirects error messages to a log file.

Possible causes of the problem are:

- The MATLAB worker failed to start due to licensing errors, the executable is not on the default path on the worker machine, or is not installed in the location where the scheduler expected it to be.
- MATLAB could not read/write the job input/output files in the scheduler’s job storage location. The storage location might not be accessible to all the worker nodes, or the user that MATLAB runs as does not have permission to read/write the job files.
- If using a generic scheduler:
  - The environment variable `MDCE_DECODE_FUNCTION` was not defined before the MATLAB worker started.
  - The decode function was not on the worker’s path.
No Results or Failed Job

Task Errors

If your job returned no results (i.e., `fetchOutputs(job)` returns an empty cell array), it is probable that the job failed and some of its tasks have their `Error` properties set.

You can use the following code to identify tasks with error messages:

```matlab
ermsgs = get(yourjob.Tasks, {'ErrorMessage'});
nonempty = ~cellfun(@isempty,errmsgs); celldisp(errmsgs(nonempty));
```

This code displays the nonempty error messages of the tasks found in the job object `yourjob`.

Debug Logs

If you are using a supported third-party scheduler, you can use the `getDebugLog` function to read the debug log from the scheduler for a particular job or task.

For example, find the failed job on your LSF scheduler, and read its debug log:

```matlab
c = parcluster('my_lsf_profile')
failedjob = findJob(c, 'State', 'failed');
message = getDebugLog(c, failedjob(1))
```

Connection Problems Between the Client and MJS

For testing connectivity between the client machine and the machines of your compute cluster, you can use Admin Center. For more information about Admin Center, including how to start it and how to test connectivity, see “Start Admin Center” (MATLAB Distributed Computing Server) and “Test Connectivity” (MATLAB Distributed Computing Server).

Detailed instructions for other methods of diagnosing connection problems between the client and MJS can be found in some of the Bug Reports listed on the MathWorks Web site.

The following sections can help you identify the general nature of some connection problems.
Client Cannot See the MJS

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

- The MJS is currently not running.
- Firewalls do not allow traffic from the client to the MJS.
- The client and the MJS are not running the same version of the software.
- The client and the MJS cannot resolve each other’s short hostnames.
- The MJS is using a nondefault BASE_PORT setting as defined in the mdce_def file, and the Host property in the cluster profile does not specify this port.

MJS Cannot See the Client

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

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

SFTP Error: Received Message Too Long

The example code for generic schedulers with non-shared file systems contacts an sftp server to handle the file transfer to and from the cluster’s file system. This use of sftp is subject to all the normal sftp vulnerabilities. One problem that can occur results in an error message similar to this:

Caused by:

```
Error using ==> RemoteClusterAccess>RemoteClusterAccess.waitForChoreToFinishOrError at 780
The following errors occurred in the
    com.mathworks.toolbox.distcomp.clusteraccess.UploadFilesChore:
Could not send Job3.common.mat for job 3:
One of the shell's init files contains a command that is writing to stdout,
interfering with sftp. Access help
com.mathworks.toolbox.distcomp.remote.spi.plugin.SftpExtraBytesFromShellException:
One of your shell's init files contains a command that is writing to stdout,
interfering with sftp.
Find and wrap the command with a conditional test, such as

    if ($?TERM != 0) then
      if ("$TERM" != "dumb") then
        /your command/
      endif
    endif

: 4: Received message is too long: 1718579037
```
The telling symptom is the phrase "Received message is too long:" followed by a very large number.

The sftp server starts a shell, usually bash or tcsh, to set your standard read and write permissions appropriately before transferring files. The server initializes the shell in the standard way, calling files like .bashrc and .cshrc. This problem happens if your shell emits text to standard out when it starts. That text is transferred back to the sftp client running inside MATLAB, and is interpreted as the size of the sftp server's response message.

To work around this error, locate the shell startup file code that is emitting the text, and either remove it or bracket it within if statements to see if the sftp server is starting the shell:

```bash
if ($?TERM != 0) then
    if ("$TERM" != "dumb") then
        /your command/
    endif
endif
```

You can test this outside of MATLAB with a standard UNIX or Windows sftp command-line client before trying again in MATLAB. If the problem is not fixed, the error message persists:

```
> sftp yourSubmitMachine
Connecting to yourSubmitMachine...
Received message too long 1718579042
```

If the problem is fixed, you should see:

```
> sftp yourSubmitMachine
Connecting to yourSubmitMachine...
```
Big Data Workflow Using Tall Arrays and Datastores

The illustration shows a typical workflow that uses tall arrays to analyze a large data set. In this workflow, you analyze a small subset of the data before scaling up to analyze the entire data set. Parallel computing can help you scale up from steps six to seven. That is, after checking that your code works on the small data set, run it on the whole data set. You can use MATLAB to enhance this workflow.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Required Products</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is your data too big?</td>
<td>To work with out-of-memory data with any number of rows, use tall arrays.</td>
<td>MATLAB</td>
<td>“Tall Arrays” (MATLAB)</td>
</tr>
<tr>
<td></td>
<td>This workflow is well suited to data analytics and machine learning.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use tall arrays in parallel on your local machine.</td>
<td>MATLAB</td>
<td>Parallel Computing Toolbox</td>
<td>“Use Tall Arrays on a Parallel Pool” on page 6-63</td>
</tr>
<tr>
<td>Use tall arrays in parallel on your cluster.</td>
<td>MATLAB</td>
<td>Parallel Computing Toolbox</td>
<td>“Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67</td>
</tr>
<tr>
<td>If your data is large in multiple dimensions, use distributed instead.</td>
<td>MATLAB</td>
<td>Distributed Computing Server</td>
<td>“Distributing Arrays to Parallel Workers” on page 3-12</td>
</tr>
</tbody>
</table>

**Running Tall Arrays in Parallel**

Parallel Computing Toolbox can immediately speed up your tall array calculations by using the full processing power of multicore computers to execute applications with a parallel pool of workers. If you already have Parallel Computing Toolbox installed, then you probably do not need to do anything special to take advantage of these capabilities. For more information about using tall arrays with Parallel Computing Toolbox, see “Use Tall Arrays on a Parallel Pool” on page 6-63.
Use mapreducer to Control Where Your Code Runs

When you execute tall arrays, the default execution environment uses either the local MATLAB session, or a local parallel pool if you have Parallel Computing Toolbox. The default pool uses local workers, typically one worker for each core in your machine. Use the mapreducer function to change the execution environment of tall arrays to use a different cluster.

One of the benefits of developing your algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, then use mapreducer to scale up and take advantage of the capabilities offered by Parallel Computing Toolbox and MATLAB Distributed Computing Server.

See Also
datastore|gather|mapreducer|tall

Related Examples
• “Use Tall Arrays on a Parallel Pool” on page 6-63
• “Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
• “Tall Arrays” (MATLAB)
• “Choose a Parallel Computing Solution” on page 1-20

More About
• “Datastore” (MATLAB)
Use Tall Arrays on a Parallel Pool

If you have Parallel Computing Toolbox, you can use tall arrays in your local MATLAB session, or on a local parallel pool. You can also run tall array calculations on a cluster if you have MATLAB Distributed Computing Server installed. This example uses the workers in a local cluster on your machine. You can develop code locally, and then scale up, to take advantage of the capabilities offered by Parallel Computing Toolbox and MATLAB Distributed Computing Server without having to rewrite your algorithm. See also “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60.

Create a datastore and convert it into a tall table.

ds = datastore('airlinesmall.csv');
varnames = {'ArrDelay', 'DepDelay'};
ds.SelectedVariableNames = varnames;
ds.TreatAsMissing = 'NA';

If you have Parallel Computing Toolbox installed, when you use the tall function, MATLAB automatically starts a parallel pool of workers, unless you turn off the default parallel pool preference. The default cluster uses local workers on your machine.

**Note** If you want to turn off automatically opening a parallel pool, change your parallel preferences. If you turn off the *Automatically create a parallel pool* option, then you must explicitly start a pool if you want the tall function to use it for parallel processing. See “Specify Your Parallel Preferences” on page 6-12.

If you have Parallel Computing Toolbox, you can run the same code as the MATLAB tall table example (MATLAB) and automatically execute it in parallel on the workers of your local machine.

Create a tall table tt from the datastore.

```
tt = tall(ds)
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
tt =
   Mx2 tall table
      ArrDelay     DepDelay
```
The display indicates that the number of rows, $M$, is not yet known. $M$ is a placeholder until the calculation completes.

Extract the arrival delay $\text{ArrDelay}$ from the tall table. This action creates a new tall array variable to use in subsequent calculations.

$$a = \texttt{tt.ArrDelay};$$

You can specify a series of operations on your tall array, which are not executed until you call `gather`. Doing so enables you to batch up commands that might take a long time. For example, calculate the mean and standard deviation of the arrival delay. Use these values to construct the upper and lower thresholds for delays that are within 1 standard deviation of the mean.

$$m = \text{mean}(a, \text{'omitnan'});$$
$$s = \text{std}(a, \text{'omitnan'});$$
$$\text{one_sigma_bounds} = [m-s \ m \ m+s];$$

Use `gather` to calculate `one_sigma_bounds`, and bring the answer into memory.

$$\text{sig1} = \text{gather}\left(\text{one_sigma_bounds}\right)$$

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4.5 sec
Evaluation completed in 6.3 sec

$$\text{sig1} =$$

$$\begin{bmatrix}
-23.4572 & 7.1201 & 37.6975
\end{bmatrix}$$
You can specify multiple inputs and outputs to `gather` if you want to evaluate several things at once. Doing so is faster than calling `gather` separately on each tall array. As an example, calculate the minimum and maximum arrival delay.

```
[max_delay, min_delay] = gather(max(a), min(a))
```

```
max_delay =
   1014

min_delay =
   -64
```

If you want to develop in serial and not use local workers or your specified cluster, enter the following command.

```
mapreducer(0);
```

If you use `mapreducer` to change the execution environment after creating a tall array, then the tall array is invalid and you must recreate it. To use local workers or your specified cluster again, enter the following command.

```
mapreducer(gcp);
```

**Note** One of the benefits of developing algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, and then use `mapreducer` to scale up to a cluster, without needing to rewrite your algorithm. For an example, see “Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67.

**See Also**
- `datastore`
- `gather`
- `mapreducer`
- `parpool`
- `table`
- `tall`

**Related Examples**
- “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
- “Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
- “Tall Arrays” (MATLAB)
More About

• “Datastore” (MATLAB)
Use Tall Arrays on a Spark Enabled Hadoop Cluster

Creating and Using Tall Tables

This example shows how to modify a MATLAB example of creating a tall table to run on a Spark enabled Hadoop® cluster. You can use this tall table to create tall arrays and calculate statistical properties. You can develop code locally and then scale up, to take advantage of the capabilities offered by Parallel Computing Toolbox and MATLAB Distributed Computing Server without having to rewrite your algorithm. See also “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60 and “Configure a Hadoop Cluster” (MATLAB Distributed Computing Server).

First, you must set environment variables and cluster properties as appropriate for your specific Spark enabled Hadoop cluster configuration. See your system administrator for the values for these and other properties necessary for submitting jobs to your cluster.

```matlab
setenv('HADOOP_HOME', '/path/to/hadoop/install')
setenv('SPARK_HOME', '/path/to/spark/install');
cluster = parallel.cluster.Hadoop;

% Optionally, if you want to control the exact number of workers:
cluster.SparkProperties('spark.executor.instances') = '16';
mapreducer(cluster);
```

**Note** In the setup step, you use `mapreducer` to set the cluster execution environment. In the next step, you create a tall array. If you modify or delete the cluster execution environment after creating a tall array, then the tall array is invalid and you must recreate it.

**Note** If you want to develop in serial and not use local workers, enter the following command.

```matlab
mapreducer(0);
```

After setting your environment variables and cluster properties, you can run the MATLAB tall table example (MATLAB) on the Spark enabled Hadoop cluster instead of on your
local machine. Create a datastore and convert it into a tall table. MATLAB automatically starts a Spark job to run subsequent calculations on the tall table.

```matlab
ds = datastore('airlinesmall.csv');
varnames = {'ArrDelay', 'DepDelay'};
ds.SelectedVariableNames = varnames;
ds.TreatAsMissing = 'NA';

Create a tall table tt from the datastore.

```matlab
tt = tall(ds)

Starting a Spark Job on the Hadoop cluster. This could take a few minutes ...done.

```
```
tt =

M×2 tall table

<table>
<thead>
<tr>
<th>ArrDelay</th>
<th>DepDelay</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>59</td>
<td>63</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The display indicates that the number of rows, M, is not yet known. M is a placeholder until the calculation completes.

Extract the arrival delay ArrDelay from the tall table. This action creates a new tall array variable to use in subsequent calculations.

```matlab
a = tt.ArrDelay;
```

You can specify a series of operations on your tall array, which are not executed until you call gather. Doing so allows you to batch up commands that might take a long time. As an example, calculate the mean and standard deviation of the arrival delay. Use these
values to construct the upper and lower thresholds for delays that are within 1 standard deviation of the mean.

\[
m = \text{mean}(a, \text{'omitnan'});
\]
\[
s = \text{std}(a, \text{'omitnan'});
\]
\[
\text{one_sigma_bounds} = [m-s \ m \ m+s];
\]

Use \texttt{gather} to calculate \texttt{one_sigma_bounds}, and bring the answer into memory.

\[
\text{sig1} = \text{gather}(\text{one_sigma_bounds})
\]

Evaluating tall expression using the Spark Cluster:
- Pass 1 of 1: Completed in 0.95 sec
Evaluation completed in 1.3 sec

\[
\text{sig1} =
\]
\[
-23.4572 \ 7.1201 \ 37.6975
\]

You can specify multiple inputs and outputs to \texttt{gather} if you want to evaluate several things at once. Doing so is faster than calling \texttt{gather} separately on each tall array. For example, calculate the minimum and maximum arrival delay.

\[
[\text{max_delay}, \text{min_delay}] = \text{gather}(\text{max}(a),\text{min}(a))
\]

\[
\text{max_delay} =
\]
\[
1014
\]

\[
\text{min_delay} =
\]
\[
-64
\]

\textbf{Note} These examples take more time to complete the first time if MATLAB is starting on the cluster workers.

When using tall arrays on a Spark enabled Hadoop cluster, compute resources from the Hadoop cluster will be reserved for the lifetime of the mapreducer execution environment. To clear these resources, you must delete the mapreducer:

\[
delete(gcmr);
\]
Alternatively, you can change to a different execution environment, for example:

mapreduce(0);

**See Also**
datastore | gather | mapreduce | parallel.cluster.Hadoop | table | tall

**Related Examples**
- “Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
- “Use Tall Arrays on a Parallel Pool” on page 6-63
- “Configure a Hadoop Cluster” (MATLAB Distributed Computing Server)
- “Tall Arrays” (MATLAB)
- “Read and Analyze Hadoop Sequence File” (MATLAB)

**More About**
- “Datastore” (MATLAB)
Run mapreduce on a Parallel Pool

In this section...

“Start Parallel Pool” on page 6-71
“Compare Parallel mapreduce” on page 6-71

Start Parallel Pool

If you have Parallel Computing Toolbox installed, execution of `mapreduce` can open a parallel pool on the cluster specified by your default profile, for use as the execution environment.

You can set your parallel preferences so that a pool does not automatically open. In this case, you must explicitly start a pool if you want `mapreduce` to use it for parallelization of its work. See “Specify Your Parallel Preferences” on page 6-12.

For example, the following conceptual code starts a pool, and some time later uses that open pool for the `mapreducer` configuration.

```matlab
p = parpool('local',n);
mr = mapreducer(p);
outds = mapreduce(tds,@MeanDistMapFun,@MeanDistReduceFun,mr)
```

Note `mapreduce` can run on any cluster that supports parallel pools. The examples in this topic use a local cluster, which works for all Parallel Computing Toolbox installations.

Compare Parallel mapreduce

The following example calculates the mean arrival delay from a datastore of airline data. First it runs `mapreduce` in the MATLAB client session, then it runs in parallel on a local cluster. The `mapreducer` function explicitly controls the execution environment.

Begin by starting a parallel pool on a local cluster:

```matlab
p = parpool('local',4);
```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Create two MapReducer objects for specifying the different execution environments for `mapreduce`.

```matlab
inMatlab = mapreducer(0);
inPool = mapreducer(p);
```

Create and preview the datastore. The data set used in this example is available in `matlabroot/toolbox/matlab/demos`.

```matlab
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',
    'SelectedVariableNames','ArrDelay','ReadSize',1000);
preview(ds)
```

```
ArrDelay
________
  8
  8
 21
 13
  4
 59
  3
 11
```

Next, run the `mapreduce` calculation in the MATLAB client session. The map and reduce functions are available in `matlabroot/toolbox/matlab/demos`.

```matlab
meanDelay = mapreduce(ds,@meanArrivalDelayMapper,@meanArrivalDelayReducer,inMatlab);
```

```
*******************************
*      MAPREDUCE PROGRESS      *
*******************************
Map   0% Reduce   0%
Map  10% Reduce   0%
Map  20% Reduce   0%
Map  30% Reduce   0%
Map  40% Reduce   0%
Map  50% Reduce   0%
Map  60% Reduce   0%
Map  70% Reduce   0%
Map  80% Reduce   0%
Map  90% Reduce   0%
Map 100% Reduce 100%
```

```matlab
readall(meanDelay)
```
Then, run the calculation on the current parallel pool. Note that the output text indicates a parallel `mapreduce`.

```matlab
meanDelay = mapreduce(ds,@meanArrivalDelayMapper,@meanArrivalDelayReducer,inPool);
```

Parallel mapreduce execution on the parallel pool:

```
*******************************
* MAPREDUCE PROGRESS         *
*******************************
Map  0% Reduce  0%
Map 100% Reduce  50%
Map 100% Reduce 100%
```

`readall(meanDelay)`

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'MeanArrivalDelay'</td>
<td>[7.1201]</td>
</tr>
</tbody>
</table>

With this relatively small data set, a performance improvement with the parallel pool is not likely. This example is to show the mechanism for running `mapreduce` on a parallel pool. As the data set grows, or the map and reduce functions themselves become more computationally intensive, you might expect to see improved performance with the parallel pool, compared to running `mapreduce` in the MATLAB client session.

**Note** When running parallel `mapreduce` on a cluster, the order of the key-value pairs in the output is different compared to running `mapreduce` in MATLAB. If your application depends on the arrangement of data in the output, you must sort the data according to your own requirements.

**See Also**

**Functions**

`datastore` | `mapreduce` | `mapreducer`
Related Examples

- “Getting Started with MapReduce” (MATLAB)
- “Run mapreduce on a Hadoop Cluster” on page 6-75

More About

- “MapReduce” (MATLAB)
- “Datastore” (MATLAB)
Run mapreduce on a Hadoop Cluster

In this section...

“Cluster Preparation” on page 6-75
“Output Format and Order” on page 6-75
“Calculate Mean Delay” on page 6-75

Cluster Preparation

Before you can run mapreduce on a Hadoop cluster, make sure that the cluster and client machine are properly configured. Consult your system administrator, or see “Configure a Hadoop Cluster” (MATLAB Distributed Computing Server).

Output Format and Order

When running mapreduce on a Hadoop cluster with binary output (the default), the resulting KeyValueDatastore points to Hadoop Sequence files, instead of binary MAT-files as generated by mapreduce in other environments. For more information, see the 'OutputType' argument description on the mapreduce reference page.

When running mapreduce on a Hadoop cluster, the order of the key-value pairs in the output is different compared to running mapreduce in other environments. If your application depends on the arrangement of data in the output, you must sort the data according to your own requirements.

Calculate Mean Delay

This example shows how to modify the MATLAB example for calculating mean airline delays to run on a Hadoop cluster.

First, you must set environment variables and cluster properties as appropriate for your specific Hadoop configuration. See your system administrator for the values for these and other properties necessary for submitting jobs to your cluster.

```matlab
setenv('HADOOP_HOME', '/path/to/hadoop/install')
cluster = parallel.cluster.Hadoop;
```
Note The specified outputFolder must not already exist. The mapreduce output from a Hadoop cluster cannot overwrite an existing folder.

You will lose your data, if mapreducer is changed or deleted.

Create a MapReducer object to specify that mapreduce should use your Hadoop cluster.

mr = mapreducer(cluster);

Create and preview the datastore. The data set is available in matlabroot/toolbox/matlab/demos.

ds = datastore('airlinesmall.csv','TreatAsMissing','NA', ... 
    'SelectedVariableNames','ArrDelay','ReadSize',1000);
preview(ds)

ArrDelay

     ________
     8
     8
    21
    13
     4
    59
     3
    11

Next, specify your output folder, output outds and call mapreduce to execute on the Hadoop cluster specified by mr. The map and reduce functions are available in matlabroot/toolbox/matlab/demos.

outputFolder = 'hdfs:///home/myuser/out1';
outds = mapreduce(ds,@myMapperFcn,@myReducerFcn,'OutputFolder',outputFolder);
meanDelay = mapreduce(ds,@meanArrivalDelayMapper,@meanArrivalDelayReducer,mr,... 
    'OutputFolder',outputFolder)

Parallel mapreduce execution on the Hadoop cluster:

*** MAPREDUCE PROGRESS ***

* Map 0% Reduce 0%
* Map 66% Reduce 0%
* Map 100% Reduce 66%
Map 100% Reduce 100%

`meanDelay =`  

*KeyValueDatastore with properties:**

- **Files:**
  - `.../tmp/alafleur/tpc00621b1_4eef_4abc_8078_646aa916e7d9/part0.seq`

- **ReadSize:** 1 key-value pairs
- **FileType:** 'seq'

Read the result.

`readall(meanDelay)`

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'MeanArrivalDelay'</td>
<td>[7.1201]</td>
</tr>
</tbody>
</table>

Although for demonstration purposes this example uses a local data set, it is likely when using Hadoop that your data set is stored in an HDFS™ file system. Likewise, you might be required to store the mapreduce output in HDFS. For details about accessing HDFS in MATLAB, see “Work with Remote Data” (MATLAB).

**See Also**

**Functions**

datastore | mapreduce | mapreducer | parallel.cluster.Hadoop

**Related Examples**

- “Getting Started with MapReduce” (MATLAB)
- “Run mapreduce on a Parallel Pool” on page 6-71

**More About**

- “MapReduce” (MATLAB)
- “Datastore” (MATLAB)
Partition a Datastore in Parallel

Partitioning a datastore in parallel, with a portion of the datastore on each worker in a parallel pool, can provide benefits in many cases:

- Perform some action on only one part of the whole datastore, or on several defined parts simultaneously.
- Search for specific values in the data store, with all workers acting simultaneously on their own partitions.
- Perform a reduction calculation on the workers across all partitions.

This example shows how to use partition to parallelize the reading of data from a datastore. It uses a small datastore of airline data provided in MATLAB, and finds the mean of the non-NaN values from its 'ArrDelay' column.

A simple way to calculate the mean is to divide the sum of all the non-NaN values by the number of non-NaN values. The following code does this for the datastore first in a non-parallel way. To begin, you define a function to amass the count and sum. If you want to run this example, copy and save this function in a folder on the MATLAB command search path.

```matlab
function [total,count] = sumAndCountArrivalDelay(ds)
  total = 0;
  count = 0;
  while hasdata(ds)
    data = read(ds);
    total = total + sum(data.ArrDelay,1,'OmitNaN');
    count = count + sum(~isnan(data.ArrDelay));
  end
end
```

The following code creates a datastore, calls the function, and calculates the mean without any parallel execution. The tic and toc functions are used to time the execution, here and in the later parallel cases.

```matlab
ds = datastore(repmat({'airlinesmall.csv'},20,1),'TreatAsMissing','NA');
ds-selectedVariableNames = 'ArrDelay';
```
reset(ds);
tic
    [total,count] = sumAndCountArrivalDelay(ds)
sumtime = toc
mean = total/count

total =
    17211680
count =
    2417320
sumtime =
    12.6408
mean =
    7.1201

The **partition** function allows you to partition the datastore into smaller parts, each represented as a datastore itself. These smaller datastores work completely independently of each other, so that you can work with them inside of parallel language features such as **parfor** loops and **spmd** blocks.

The number of partitions in the following code is set by the `numpartitions` function, based on the datastore itself (ds) and the parallel pool (gcp) size. This does not necessarily equal the number of workers in the pool. In this case, the number of loop iterations is then set to the number of partitions (N).

The following code starts a parallel pool on a local cluster, then partitions the datastore among workers for iterating over the loop. Again, a separate function is called, which includes the **parfor** loop to amass the count and sum totals. Copy and save this function if you want to run the example.
function [total, count] = parforSumAndCountArrivalDelay(ds)

N = numpartitions(ds,gcp);
total = 0;
count = 0;
parfor ii = 1:N
    % Get partition ii of the datastore.
    subds = partition(ds,N,ii);

    [localTotal,localCount] = sumAndCountArrivalDelay(subds);
    total = total + localTotal;
    count = count + localCount;
end
end

Now the MATLAB code calls this new function, so that the counting and summing of the non-NAN values can occur in parallel loop iterations.

p = parpool('local',4);
reset(ds);
tic
    [total,count] = parforSumAndCountArrivalDelay(ds)
parfortime = toc
mean = total/count

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

    total =
            17211680

    count =
            2417320

parfortime =
Rather than let the software calculate the number of partitions, you can explicitly set this value, so that the data can be appropriately partitioned to fit your algorithm. For example, to parallelize data from within an `spmd` block, you can specify the number of workers (`numlabs`) as the number of partitions to use. The following function uses an `spmd` block to perform a parallel read, and explicitly sets the number of partitions equal to the number of workers. To run this example, copy and save the function.

```matlab
% Copyright 2015 The MathWorks, Inc.

function [total,count] = spmdSumAndCountArrivalDelay(ds)
    spmd
        subds = partition(ds,numlabs,labindex);
        [total,count] = sumAndCountArrivalDelay(subds);
    end
    total = sum([total{:}]);
    count = sum([count{:}]);
end
```

Now the MATLAB code calls the function that uses an `spmd` block.

```matlab
reset(ds);
 tic
    [total,count] = spmdSumAndCountArrivalDelay(ds)
 spmdtime = toc
 mean = total/count

    total = 
     17211680

    count = 
```
2417320

spmdtime =
5.3852

mean =
7.1201

delete(p);
Parallel pool using the 'local' profile is shutting down.

You might get some idea of modest performance improvements by comparing the times recorded in the variables sumtime, parfortime, and spmdtime. Your results might vary, as the performance can be affected by the datastore size, parallel pool size, hardware configuration, and other factors.
Program Independent Jobs

• “Program Independent Jobs” on page 7-2
• “Program Independent Jobs on a Local Cluster” on page 7-3
• “Program Independent Jobs for a Supported Scheduler” on page 7-8
• “Share Code with the Workers” on page 7-16
• “Integration Scripts for Generic Schedulers” on page 7-21
Program Independent Jobs

The tasks in an independent job do not directly communicate with each other and are independent. The tasks do not need to run simultaneously, and a worker can run several tasks of the same job in succession. Typically, all tasks perform the same or similar functions on different data sets in an *embarrassingly parallel* configuration.

Some of the details of a job and its tasks can depend on the type of scheduler you are using:

- “Program Independent Jobs on a Local Cluster” on page 7-3
- “Program Independent Jobs for a Supported Scheduler” on page 7-8
- “Integration Scripts for Generic Schedulers” on page 7-21
Program Independent Jobs on a Local Cluster

Create and Run Jobs with a Local Cluster

Some jobs require more control than the functionality offered by high-level constructs like `spmd` and `parfor`. In such cases, you have to program all the steps for creating and running the job. Using the local cluster (or local scheduler) on your machine lets you create and test your jobs without using the resources of your network cluster. Distributing tasks to workers that are all running on your client machine do not offer any performance enhancement. Therefore this feature is provided primarily for code development, testing, and debugging.

**Note** Workers running in a local cluster on a Microsoft Windows operating system can display Simulink graphics and the output from certain functions such as `uigetfile` and `uigetdir`. (With other platforms or schedulers, workers cannot display any graphical output.) This behavior is subject to removal in a future release.

This section details the steps of a typical programming session with Parallel Computing Toolbox software using a local cluster:

- “Create a Cluster Object” on page 7-4
- “Create a Job” on page 7-4
- “Create Tasks” on page 7-5
- “Submit a Job to the Cluster” on page 7-5
- “Fetch the Job Results” on page 7-6

The objects used by the client session to interact with the cluster are only references to data in the cluster job storage location, not in the client session. After jobs and tasks are created, you can close your client session and restart it, and your job still resides in the storage location. You can find existing jobs using the `findJob` function or the `Jobs` property of the cluster object.
Create a Cluster Object

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

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

Create a Job

You create a job with the `createJob` function. This statement creates a job in the cluster job storage location and creates the job object `job1` in the client session. If you omit the semicolon at the end of the command, it displays some information about the job.

```matlab
job1 = createJob(c)
```

Job

Properties:

- **ID**: 2
- **Type**: Independent
- **Username**: eng864
- **State**: pending
- **SubmitTime**: 
- **StartTime**: 
- **Running Duration**: 0 days 0h 0m 0s

- **AutoAttachFiles**: true
- **Auto Attached Files**: List files
  - **AttachedFiles**: {}
  - **AdditionalPaths**: {}

Associated Tasks:

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

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

The scheduler display now indicates the existence of your job, which is the pending one, as appears in this partial listing:
Create Tasks

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

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

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

```plaintext
job1.Tasks
```

<table>
<thead>
<tr>
<th>ID</th>
<th>State</th>
<th>FinishTime</th>
<th>Function</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
</tbody>
</table>

Submit a Job to the Cluster

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

```plaintext
submit(job1)
```

The local scheduler starts the workers on your machine, and distributes the tasks of `job1` to these workers for evaluation.
**Fetch the Job Results**

The results of each task evaluation are stored in the task object `OutputArguments` property as a cell array. After waiting for the job to complete, use the function `fetchOutputs` to retrieve the results from all the tasks in the job.

```matlab
wait(job1)
results = fetchOutputs(job1);
```

Display the results from each task.

```matlab
results{1:5}
```

0.9501    0.4860    0.4565
0.2311    0.8913    0.0185
0.6068    0.7621    0.8214
0.4447    0.9218    0.4057
0.6154    0.7382    0.9355
0.7919    0.1763    0.9169
0.4103    0.3529    0.1389
0.8936    0.8132    0.2028
0.0579    0.0099    0.1987
0.6038    0.0153    0.9318
0.2722    0.7468    0.4660
0.1988    0.4451    0.4186
0.8462    0.6721    0.6813
0.5252    0.8381    0.3795
0.2026    0.0196    0.8318

After the job is complete, you can repeat the commands to examine the updated status of the cluster, job, and task objects:

```matlab
c
job1
job1.Tasks
```

**Local Cluster Behavior**

The local scheduler runs in the MATLAB client session, so you do not have to start any separate scheduler or MJS process for the local scheduler. When you submit a job to the
local cluster, the scheduler starts a MATLAB worker for each task in the job. You can do this for as many workers as allowed by the local profile. If your job has more tasks than allowed workers, the scheduler waits for one of the current tasks to complete before starting another MATLAB worker to evaluate the next task. You can modify the number of allowed workers in the local cluster profile. If not specified, the default is to run only as many workers as computational cores on the machine.

The local cluster has no interaction with any other scheduler or MJS, nor with any other workers that can also be running on your client machine under the mdc service. Multiple MATLAB sessions on your computer can each start its own local scheduler with its own workers, but these groups do not interact with each other.

When you end your MATLAB client session, its local scheduler and any workers that happen to be running also stop immediately.
Program Independent Jobs for a Supported Scheduler

In this section...

“Create and Run Jobs” on page 7-8
“Manage Objects in the Scheduler” on page 7-13

Create and Run Jobs

This section details the steps of a typical programming session with Parallel Computing Toolbox software using a supported job scheduler on a cluster. Supported schedulers include the MATLAB job scheduler (MJS), Platform LSF (Load Sharing Facility), Microsoft Windows HPC Server (including CCS), PBS Pro, or a TORQUE scheduler.

This section assumes that you have an MJS, LSF, PBS Pro, TORQUE, or Windows HPC Server (including CCS and HPC Server 2008) scheduler installed and running on your network. With all of these cluster types, the basic job programming sequence is the same:

- “Define and Select a Profile” on page 7-8
- “Find a Cluster” on page 7-9
- “Create a Job” on page 7-10
- “Create Tasks” on page 7-11
- “Submit a Job to the Job Queue” on page 7-12
- “Retrieve Job Results” on page 7-12

Note that the objects that the client session uses to interact with the MJS are only references to data that is actually contained in the MJS, not in the client session. After jobs and tasks are created, you can close your client session and restart it, and your job is still stored in the MJS. You can find existing jobs using the findJob function or the Jobs property of the MJS cluster object.

Define and Select a Profile

A cluster profile identifies the type of cluster to use and its specific properties. In a profile, you define how many workers a job can access, where the job data is stored, where MATLAB is accessed and many other cluster properties. The exact properties are determined by the type of cluster.

The step in this section all assume the profile with the name MyProfile identifies the cluster you want to use, with all necessary property settings. With the proper use of a
profile, the rest of the programming is the same, regardless of cluster type. After you define or import your profile, you can set it as the default profile in the Profile Manager GUI, or with the command:

```matlab
parallel.defaultClusterProfile('MyProfile')
```

A few notes regarding different cluster types and their properties:

**Notes** In a shared file system, all nodes require access to the folder specified in the cluster object's `JobStorageLocation` property.

Because Windows HPC Server requires a shared file system, all nodes require access to the folder specified in the cluster object's `JobStorageLocation` property.

In a shared file system, MATLAB clients on many computers can access the same job data on the network. Properties of a particular job or task should be set from only one client computer at a time.

When you use an LSF scheduler in a nonshared file system, the scheduler might report that a job is in the finished state even though the LSF scheduler might not yet have completed transferring the job's files.

**Find a Cluster**

You use the `parcluster` function to identify a cluster and to create an object representing the cluster in your local MATLAB session.

To find a specific cluster, user the cluster profile to match the properties of the cluster you want to use. In this example, `MyProfile` is the name of the profile that defines the specific cluster.

```matlab
c = parcluster('MyProfile');
```

### MJS Cluster

**Properties**

- Name: `my_mjs`
- Profile: `MyProfile`
- Modified: `false`
- Host: `node345`
- Username: `mylogin`

- NumWorkers: 1
- NumBusyWorkers: 0
Create a Job

You create a job with the `createJob` function. Although this command executes in the client session, it actually creates the job on the cluster, `c`, and creates a job object, `job1`, in the client session.

```matlab
job1 = createJob(c)
```

Job

Properties:

- ID: 1
- Type: Independent
- Username: mylogin
- State: pending
- SubmitTime:
- StartTime:
- Running Duration: 0 days 0h 0m 0s
- AutoAttachFiles: true
- Auto Attached Files: List files
- AttachedFiles: {}
- AdditionalPaths: {}

Associated Tasks:

- Number Pending: 0
- Number Running: 0
- Number Finished: 0
- Task ID of Errors: []
Note that the job’s State property is pending. This means the job has not been queued for running yet, so you can now add tasks to it.

The cluster’s display now includes one pending job, as shown in this partial listing:

c

Associated Jobs

<table>
<thead>
<tr>
<th>Number Pending: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Queued: 0</td>
</tr>
<tr>
<td>Number Running: 0</td>
</tr>
<tr>
<td>Number Finished: 0</td>
</tr>
</tbody>
</table>

You can transfer files to the worker by using the AttachedFiles property of the job object. For details, see “Share Code with the Workers” on page 7-16.

**Create Tasks**

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

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

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

```matlab
job1.Tasks
```

<table>
<thead>
<tr>
<th>ID</th>
<th>State</th>
<th>FinishTime</th>
<th>Function</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>pending</td>
<td></td>
<td>@rand</td>
<td></td>
</tr>
</tbody>
</table>

Alternatively, you can create the five tasks with one call to `createTask` by providing a cell array of five cell arrays defining the input arguments to each task.
T = createTask(job1, @rand, 1, {{3,3} {3,3} {3,3} {3,3} {3,3}});

In this case, T is a 5-by-1 matrix of task objects.

**Submit a Job to the Job Queue**

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

submit(job1)

The job manager distributes the tasks of job1 to its registered workers for evaluation.

Each worker performs the following steps for task evaluation:

1. Receive AttachedFiles and AdditionalPaths from the job. Place files and modify the path accordingly.
2. Run the jobStartup function the first time evaluating a task for this job. You can specify this function in AttachedFiles or AdditionalPaths. When using an MJS, if the same worker evaluates subsequent tasks for this job, jobStartup does not run between tasks.
3. Run the taskStartup function. You can specify this function in AttachedFiles or AdditionalPaths. This runs before every task evaluation that the worker performs, so it could occur multiple times on a worker for each job.
4. If the worker is part of forming a new parallel pool, run the poolStartup function. (This occurs when executing parpool or when running other types of jobs that form and use a parallel pool, such as batch.)
5. Receive the task function and arguments for evaluation.
6. Evaluate the task function, placing the result in the task’s OutputArguments property. Any error information goes in the task’s Error property.
7. Run the taskFinish function.

**Retrieve Job Results**

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

wait(job1)
results = fetchOutputs(job1);
Display the results from each task.

\[
\text{results}\{1:5\} = \\
\begin{bmatrix}
0.9501 & 0.4860 & 0.4565 \\
0.2311 & 0.8913 & 0.0185 \\
0.6068 & 0.7621 & 0.8214 \\
0.4447 & 0.9218 & 0.4057 \\
0.6154 & 0.7382 & 0.9355 \\
0.7919 & 0.1763 & 0.9169 \\
0.4103 & 0.3529 & 0.1389 \\
0.8936 & 0.8132 & 0.2028 \\
0.0579 & 0.0099 & 0.1987 \\
0.6038 & 0.0153 & 0.9318 \\
0.2722 & 0.7468 & 0.4660 \\
0.1988 & 0.4451 & 0.4186 \\
0.8462 & 0.6721 & 0.6813 \\
0.5252 & 0.8381 & 0.3795 \\
0.2026 & 0.0196 & 0.8318 \\
\end{bmatrix}
\]

Manage Objects in the Scheduler

Because all the data of jobs and tasks resides in the cluster job storage location, these objects continue to exist even if the client session that created them has ended. The following sections describe how to access these objects and how to permanently remove them:

- “What Happens When the Client Session Ends” on page 7-13
- “Recover Objects” on page 7-14
- “Reset Callback Properties (MJS Only)” on page 7-14
- “Remove Objects Permanently” on page 7-15

What Happens When the Client Session Ends

When you close the client session of Parallel Computing Toolbox software, all of the objects in the workspace are cleared. However, the objects in MATLAB Distributed Computing Server software or other cluster resources remain in place. When the client
session ends, only the local reference objects are lost, not the actual job and task data in the cluster.

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

Recover Objects

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

You create cluster objects in the client session by using the parcluster function.

```matlab
c = parcluster('MyProfile');
```

When you have access to the cluster by the object `c`, you can create objects that reference all those job contained in that cluster. The jobs are accessible in cluster object’s Jobs property, which is an array of job objects:

```matlab
all_jobs = c.Jobs
```

You can index through the array `all_jobs` to locate a specific job.

Alternatively, you can use the `findJob` function to search in a cluster for any jobs or a particular job identified by any of its properties, such as its State.

```matlab
all_jobs = findJob(c);
finished_jobs = findJob(c,'State','finished')
```

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

Reset Callback Properties (MJS Only)

When restarting a client session, you lose the settings of any callback properties (for example, the `FinishedFcn` property) on jobs or tasks. These properties are commonly used to get notifications in the client session of state changes in their objects. When you create objects in a new client session that reference existing jobs or tasks, you must reset these callback properties if you intend to use them.
Remove Objects Permanently

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

- “Delete Selected Objects” on page 7-15
- “Start an MJS from a Clean State” on page 7-15

Delete Selected Objects

From the command line in the MATLAB client session, you can call the delete function for any job or task object. If you delete a job, you also remove all tasks contained in that job.

For example, find and delete all finished jobs in your cluster that belong to the user joep.

```matlab
c = parcluster('MyProfile')
finished_jobs = findJob(c,'State','finished','Username','joep')
delete(finished_jobs)
clear finished_jobs
```

The delete function permanently removes these jobs from the cluster. The clear function removes the object references from the local MATLAB workspace.

Start an MJS from a Clean State

When an MJS starts, by default it starts so that it resumes its former session with all jobs intact. Alternatively, an MJS can start from a clean state with all its former history deleted. Starting from a clean state permanently removes all job and task data from the MJS of the specified name on a particular host.

As a network administration feature, the -clean flag of the startjobmanager script is described in “Start in a Clean State” (MATLAB Distributed Computing Server) in the MATLAB Distributed Computing Server System Administrator's Guide.
Share Code with the Workers

Because the tasks of a job are evaluated on different machines, each machine must have access to all the files needed to evaluate its tasks. The basic mechanisms for sharing code are explained in the following sections:

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Workers Access Files Directly” on page 7-16</td>
</tr>
<tr>
<td>“Pass Data to and from Worker Sessions” on page 7-17</td>
</tr>
<tr>
<td>“Pass MATLAB Code for Startup and Finish” on page 7-19</td>
</tr>
</tbody>
</table>

Workers Access Files Directly

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

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

- By using the job’s `AdditionalPaths` property. This is the preferred method for setting the path, because it is specific to the job.

  `AdditionalPaths` identifies folders to be added to the top of the command search path of worker sessions for this job. If you also specify `AttachedFiles`, the `AttachedFiles` are above `AdditionalPaths` on the workers’ path.

  When you specify `AdditionalPaths` at the time of creating a job, the settings are combined with those specified in the applicable cluster profile. Setting `AdditionalPaths` on a job object after it is created does not combine the new setting with the profile settings, but overwrites existing settings for that job.

  `AdditionalPaths` is empty by default. For a mixed-platform environment, the character vectors can specify both UNIX and Microsoft Windows style paths; those setting that are not appropriate or not found for a particular machine generate warnings and are ignored.

  This example sets the MATLAB worker path in a mixed-platform environment to use functions in both the central repository `/central/funcs` and the department archive `/dept1/funcs`, which each also have a Windows UNC path.
c = parcluster(); % Use default
job1 = createJob(c);
ap = {'/central/funcs','/dept1/funcs', ...
\OurDomain\central\funcs','\OurDomain\dept1\funcs'};
job1.AdditionalPaths = ap;

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

Access to these files can be passed to the worker by the job's AttachedFiles or AdditionalPaths property. Otherwise, the version of each of these files that is used is the one highest on the worker’s path.

Access to files among shared resources can depend upon permissions based on the user name. You can set the user name with which the MJS and worker services of MATLAB Distributed Computing Server software run by setting the MDCEUSER value in the mdce_def file before starting the services. For Microsoft Windows operating systems, there is also MDCEPASS for providing the account password for the specified user. For an explanation of service default settings and the mdce_def file, see “Define Script Defaults” (MATLAB Distributed Computing Server) in the MATLAB Distributed Computing Server System Administrator's Guide.

**Pass Data to and from Worker Sessions**

A number of properties on task and job objects are designed for passing code or data from client to scheduler to worker; and back. This information could include MATLAB code necessary for task evaluation, or the input data for processing or output data resulting from task evaluation. The following properties facilitate this communication:

• **InputArguments** — This property of each task contains the input data you specified when creating the task. This data gets passed into the function when the worker performs its evaluation.

• **OutputArguments** — This property of each task contains the results of the function’s evaluation.

• **JobData** — This property of the job object contains data that gets sent to every worker that evaluates tasks for that job. This property works efficiently because the data is passed to a worker only once per job, saving time if that worker is evaluating
more than one task for the job. (Note: Do not confuse this property with the UserData property on any objects in the MATLAB client. Information in UserData is available only in the client, and is not available to the scheduler or workers.)

• **AttachedFiles** — This property of the job object is a cell array in which you manually specify all the folders and files that get sent to the workers. On the worker, the files are installed and the entries specified in the property are added to the search path of the worker session.

**AttachedFiles** contains a list of folders and files that the worker need to access for evaluating a job’s tasks. The value of the property (empty by default) is defined in the cluster profile or in the client session. You set the value for the property as a cell array of character vectors. Each character vector is an absolute or relative pathname to a folder or file. (Note: If these files or folders change while they are being transferred, or if any of the folders are empty, a failure or error can result. If you specify a pathname that does not exist, an error is generated.)

The first time a worker evaluates a task for a particular job, the scheduler passes to the worker the files and folders in the **AttachedFiles** property. On the worker machine, a folder structure is created that is exactly the same as that accessed on the client machine where the property was set. Those entries listed in the property value are added to the top of the command search path in the worker session. (Subfolders of the entries are not added to the path, even though they are included in the folder structure.) To find out where the files are placed on the worker machine, use the function `getAttachedFilesFolder` in code that runs on the worker.

When the worker runs subsequent tasks for the same job, it uses the folder structure already set up by the job’s **AttachedFiles** property for the first task it ran for that job.

When you specify **AttachedFiles** at the time of creating a job, the settings are combined with those specified in the applicable profile. Setting **AttachedFiles** on a job object after it is created does not combine the new setting with the profile settings, but overwrites the existing settings for that job.

The transfer of **AttachedFiles** occurs for each worker running a task for that particular job on a machine, regardless of how many workers run on that machine. Normally, the attached files are deleted from the worker machine when the job is completed, or when the next job begins.

• **AutoAttachFiles** — This property of the job object uses a logical value to specify that you want MATLAB to perform an analysis on the task functions in the job and on
manually attached files to determine which code files are necessary for the workers, and to automatically send those files to the workers. You can set this property value in a cluster profile using the Profile Manager, or you can set it programmatically on a job object at the command line.

c = parcluster();
j = createJob(c);
j.AutoAttachFiles = true;

The supported code file formats for automatic attachment are MATLAB files (.m extension), P-code files (.p), and MEX-files (.mex). Note that AutoAttachFiles does not include data files for your job; use the AttachedFiles property to explicitly transfer these files to the workers.

Use listAutoAttachedFiles to get a listing of the code files that are automatically attached to a job.

If the AutoAttachFiles setting is true for the cluster profile used when starting a parallel pool, MATLAB performs an analysis on spmd blocks, parfor-loops, and other attached files to determine what other code files are necessary for execution, then automatically attaches those files to the parallel pool so that the code is available to the workers.

Note There is a default maximum amount of data that can be sent in a single call for setting properties. This limit applies to the OutputArguments property as well as to data passed into a job as input arguments or AttachedFiles. If the limit is exceeded, you get an error message. For more information about this data transfer size limit, see “Attached Files Size Limitations” on page 6-55.

Pass MATLAB Code for Startup and Finish

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

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

- jobStartup.m automatically executes on a worker when the worker runs its first task of a job.
- `taskStartup.m` automatically executes on a worker each time the worker begins evaluation of a task.
- `poolStartup.m` automatically executes on a worker each time the worker is included in a newly started parallel pool.
- `taskFinish.m` automatically executes on a worker each time the worker completes evaluation of a task.

Empty versions of these files are provided in the folder:

`matlabroot/toolbox/distcomp/user`

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

Alternatively, you can create your own versions of these files and pass them to the job as part of the `AttachedFiles` property, or include the path names to their locations in the `AdditionalPaths` property.

The worker gives precedence to the versions provided in the `AttachedFiles` property, then to those pointed to in the `AdditionalPaths` property. If any of these files is not included in these properties, the worker uses the version of the file in the `toolbox/distcomp/user` folder of the worker’s MATLAB installation.
Integration Scripts for Generic Schedulers

In this section...

“Sample Integration Scripts” on page 7-22
“Writing Custom Integration Scripts” on page 7-23
“Adding User Customization” on page 7-32
“Managing Jobs with Generic Scheduler” on page 7-33
“Submitting from a Remote Host” on page 7-34
“Submitting without a Shared File System” on page 7-35

The generic scheduler interface provides complete flexibility to configure the interaction of the MATLAB client, MATLAB workers, and a third-party scheduler. The integration scripts define how MATLAB interacts with your setup.

The following table lists the supported integration script functions and the stage at which they are evaluated:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>independentSubmitFcn.m</td>
<td>Submitting an independent job</td>
</tr>
<tr>
<td>communicatingSubmitFcn.m</td>
<td>Submitting a communicating job</td>
</tr>
<tr>
<td>getJobStateFcn.m</td>
<td>Querying the state of a job</td>
</tr>
<tr>
<td>cancelJobFcn.m</td>
<td>Canceling a job</td>
</tr>
<tr>
<td>cancelTaskFcn.m</td>
<td>Canceling a task</td>
</tr>
<tr>
<td>deleteJobFcn.m</td>
<td>Deleting a job</td>
</tr>
<tr>
<td>deleteTaskFcn.m</td>
<td>Deleting a task</td>
</tr>
<tr>
<td>postConstructFcn.m</td>
<td>After creating a parallel.cluster.Generic instance</td>
</tr>
</tbody>
</table>

These integration scripts are evaluated only if they have the expected file name and are located in the folder specified by the `IntegrationScriptsLocation` property of the cluster. For more information about how to configure a generic cluster profile, see “Configure Using the Generic Scheduler Interface” (MATLAB Distributed Computing Server).
The independentSubmitFcn.m must exist to submit an independent job, and the communicatingSubmitFcn.m must exist to submit a communicating job.

Sample Integration Scripts

To support usage of the generic scheduler interface, integration scripts are available for the following third-party schedulers:

- IBM Platform LSF
- Grid Engine family
- PBS family
- SLURM

Each installer provides scripts for three possible submission modes:

- Shared - The client can submit directly to the scheduler, and the client and the cluster nodes (or machines) have a shared file system.
- Remote - The client and cluster nodes have a shared file system, but the client machine cannot submit directly to the scheduler, such as when the client utilities of the scheduler are not installed. This mode uses the ssh protocol to submit commands to the scheduler using a remote host.
- Nonshared - The client and cluster nodes do not have a shared file system. This mode uses the ssh protocol to submit commands to the scheduler using a remote host, and it uses the sftp protocol to copy job and task files to the cluster file system.

Each submission mode has its own subfolder within the installation folder. This subfolder contains a README file that provides specific instructions on how to use the scripts. Before using the scripts, decide which submission mode describes your network setup.

To run the installer, download the appropriate support package for your scheduler, and open it in your MATLAB client. The installer includes a wizard to guide you through creating a cluster profile for your cluster configuration.

If your scheduler or cluster configuration is not supported by one of the support packages, it is recommended that you modify the scripts of one of these packages. For more information on how to write a set of integration scripts for generic schedulers, see “Writing Custom Integration Scripts” on page 7-23.
**Wrapper Scripts**

The sample integration scripts use *wrapper* scripts to simplify the implementation of `independentSubmitFcn.m` and `communicatingSubmitFcn.m`. These scripts are not required, however, using them is a good practice to make your code more readable. This table describes these scripts:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>independentJobWrapper.sh</td>
<td>Used in <code>independentSubmitFcn.m</code> to embed a call to the MATLAB executable with the appropriate arguments. It uses environment variables for the location of the executable and its arguments. For an example of its use, see “Sample script for a SLURM scheduler” on page 7-25.</td>
</tr>
<tr>
<td>communicatingJobWrapper.sh</td>
<td>Used in <code>communicatingSubmitFcn.m</code> to distribute a communicating job in your cluster. This script implements the steps in “Submit scheduler job to launch MPI process” on page 7-27. For an example of its use, see “Sample script for a SLURM scheduler” on page 7-28.</td>
</tr>
</tbody>
</table>

**Writing Custom Integration Scripts**

**Note** When writing your own integration scripts, it is a good practice to start by modifying one of the sample integration scripts that most closely matches your setup (see “Sample Integration Scripts” on page 7-22).

**independentSubmitFcn**

When you submit an independent job to a generic cluster, the `independentSubmitFcn.m` function executes in the MATLAB client session.

The declaration line of this function must be:

```matlab
function independentSubmitFcn(cluster, job, environmentProperties)
```
Each task in a MATLAB independent job corresponds to a single job on your scheduler. The purpose of this function is to submit $N$ jobs to your third-party scheduler, where $N$ is the number of tasks in the independent job. Each of these jobs must:

1. Set the five environment variables required by the worker MATLAB to identify the individual task to run. For more information, see “Configure the worker environment” on page 7-24.

2. Call the appropriate MATLAB executable to start the MATLAB worker and run the task. For more information, see “Submit scheduler jobs to run MATLAB workers” on page 7-25.

**Configure the worker environment**

This table identifies the five environment variables and values that must be set on the worker MATLAB to run an individual task:

<table>
<thead>
<tr>
<th>Environment Variable Name</th>
<th>Environment Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDCE_DECODE_FUNCTION</td>
<td>'parallel.cluster.generic.independentDecodeFcn'</td>
</tr>
<tr>
<td>MDCE_STORAGE_CONSTRUCTOR</td>
<td>environmentProperties.StorageConstructor</td>
</tr>
</tbody>
</table>
| MDCE_STORAGE_LOCATION             | • If you have a shared file system between the client and cluster nodes, use environmentProperties.StorageLocation.  
                                         • If you do not have a shared file system between the client and cluster nodes, select a folder visible to all cluster nodes. For instructions on copying job and task files between client and cluster nodes, see “Submitting without a Shared File System” on page 7-35. |
| MDCE_JOB_LOCATION                 | environmentProperties.JobLocation                                                         |
| MDCE_TASK_LOCATION                | environmentProperties.TaskLocation{n} for the $n$th task                                  |

Many schedulers support copying the client environment as part of the submission command. If so, you can set the previous environment variables in the client, so the
scheduler can copy them to the worker environment. If not, you must modify your submission command to forward these variables.

**Submit scheduler jobs to run MATLAB workers**

Once the five required parameters for a given job and task are defined on a worker, the task is run by calling the MATLAB executable with suitable arguments. The MATLAB executable to call is defined in `environmentProperties.MatlabExecutable`. The arguments to pass are defined in `environmentProperties.MatlabArguments`.

**Note** If you cannot submit directly to your scheduler from the client machine, see “Submitting from a Remote Host” on page 7-34 for instructions on how to submit using `ssh`.

**Sample script for a SLURM scheduler**

This script shows a basic submit function for a SLURM scheduler with a shared file system. For a more complete example, see the sample support scripts in “Sample Integration Scripts” on page 7-22.

```matlab
function independentSubmitFcn(cluster,job,environmentProperties)

    % Specify the required environment variables.
    setenv('MDCE_DECODE_FUNCTION', 'parallel.cluster.generic.independentDecodeFcn');
    setenv('MDCE_STORAGE_CONSTRUCTOR', environmentProperties.StorageConstructor);
    setenv('MDCE_STORAGE_LOCATION', environmentProperties.StorageLocation);
    setenv('MDCE_JOB_LOCATION', environmentProperties.JobLocation);

    % Specify the MATLAB executable and arguments to run on the worker.
    % These are used in the independentJobWrapper.sh script.
    setenv('MDCE_MATLAB_EXE', environmentProperties.MatlabExecutable);
    setenv('MDCE_MATLAB_ARGS', environmentProperties.MatlabArguments);

    for ii = 1:environmentProperties.NumberOfTasks
        % Specify the environment variable required to identify which task to run.
        setenv('MDCE_TASK_LOCATION', environmentProperties.TaskLocations{ii});
        % Specify the command to submit the job to the SLURM scheduler.
        % SLURM will automatically copy environment variables to workers.
        commandToRun = 'sbatch --ntasks=1 independentJobWrapper.sh';
        [cmdFailed, cmdOut] = system(commandToRun);
    end
end
```
The previous example submits a simple bash script, *independentJobWrapper.sh*, to the scheduler. The *independentJobWrapper.sh* script embeds the MATLAB executable and arguments using environment variables:

```
#!/bin/sh
# MDCE_MATLAB_EXE - the MATLAB executable to use
# MDCE_MATLAB_ARGS - the MATLAB args to use
exec "${MDCE_MATLAB_EXE}" ${MDCE_MATLAB_ARGS}
```

**communicatingSubmitFcn**

When you submit a communicating job to a generic cluster, the *communicatingSubmitFcn.m* function executes in the MATLAB client session. The declaration line of this function must be:

```
function communicatingSubmitFcn(cluster,job,environmentProperties)
```

The purpose of this function is to submit a single job to your scheduler. This job must:

1. Set the four environment variables required by the MATLAB workers to identify the job to run. For more information, see “Configure the worker environment” on page 7-26.

2. Call MPI to distribute your job to N MATLAB workers. N corresponds to the maximum value specified in the *NumWorkersRange* property of the MATLAB job. For more information, see “Submit scheduler job to launch MPI process” on page 7-27.

**Configure the worker environment**

This table identifies the four environment variables and values that must be set on the worker MATLAB to run a task of a communicating job:

<table>
<thead>
<tr>
<th>Environment Variable Name</th>
<th>Environment Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDCE_DECODE_FUNCTION</td>
<td>'parallel.cluster.generic.communicatingDecodeFcn'</td>
</tr>
<tr>
<td>MDCE_STORAGE_CONSTRUCTOR</td>
<td>environmentProperties.StorageConstructor</td>
</tr>
</tbody>
</table>
If you have a shared file system between the client and cluster nodes, use `environmentProperties.StorageLocation`.

If you do not have a shared file system between the client and cluster nodes, select a folder which exists on all cluster nodes. For instructions on copying job and task files between client and cluster nodes, see “Submitting without a Shared File System” on page 7-35.

| MDCE_JOBLOCATION | environmentProperties.JobLocation |

Many schedulers support copying the client environment as part of the submission command. If so, you can set the previous environment variables in the client, so the scheduler can copy them to the worker environment. If not, you must modify your submission command to forward these variables.

**Submit scheduler job to launch MPI process**

After you define the four required parameters for a given job, run your job by launching \(N\) worker MATLAB processes using `mpiexec`. `mpiexec` is software shipped with the Parallel Computing Toolbox that implements the Message Passing Interface (MPI) standard to allow communication between the worker MATLAB processes.

To use `mpiexec` to distribute a job, the `smpd` service must be running on all nodes that run MATLAB workers.

For more information about `mpiexec` and `smpd`, see the MPICH2 home page at https://www.mcs.anl.gov/research/projects/mpich2. For user’s guides and installation instructions on that page, select **Documentation > User Docs**.

To run your job, you must submit a job to your scheduler, which executes the following steps. Note that `matlabroot` refers to the MATLAB installation location on your worker nodes.

1. Request \(N\) processes from the scheduler. \(N\) corresponds to the maximum value specified in the `NumWorkersRange` property of the MATLAB job.

2. Identify the cluster hosts allocated to your job, and how many processes are allocated per host.
3 Start the `smpd` process on each host allocated to your job. The `smpd` executable is located at `matlabroot/bin/mw_smpd`.

4 Call `mpiexec` to start worker MATLAB processes on the hosts running `smpd`. The number of worker MATLAB processes to start on each host should match the number of processes allocated by your scheduler. The `mpiexec` executable is located at `matlabroot/bin/mw_mpiexec`.

The four environment variables identified in “Configure the worker environment” on page 7-26 must be forwarded through the `mpiexec` command.

5 Once the job has finished, shutdown the `smpd` processes.

**Note** For a complete example of the previous steps, see the `communicatingJobWrapper.sh` script provided with any of the sample integration scripts in “Sample Integration Scripts” on page 7-22. Use this script as a starting point if you need to write your own script.

**Sample script for a SLURM scheduler**

The following script shows a basic submit function for a SLURM scheduler with a shared file system.

The submitted job is contained in a bash script, `communicatingJobWrapper.sh`. This script implements the relevant steps in “Submit scheduler job to launch MPI process” on page 7-27 for a SLURM scheduler. For a more complete example, see the sample support scripts in “Sample Integration Scripts” on page 7-22.

```matlab
function communicatingSubmitFcn(cluster,job,environmentProperties)
    % Specify the four required environment variables.
    setenv('MDCE_DECODE_FUNCTION', 'parallel.cluster.generic.communicatingDecodeFcn');
    setenv('MDCE_STORAGE_CONSTRUCTOR', environmentProperties.StorageConstructor);
    setenv('MDCE_STORAGE_LOCATION', environmentProperties.StorageLocation);
    setenv('MDCE_JOB_LOCATION', environmentProperties.JobLocation);

    % Specify the MATLAB executable and arguments to run on the worker.
    % These are used in the communicatingJobWrapper.sh script.
    setenv('MDCE_MATLAB_EXE', environmentProperties.MatlabExecutable);
    setenv('MDCE_MATLAB_ARGS', environmentProperties.MatlabArguments);
    setenv('MDCE_CMR', cluster.ClusterMatlabRoot);
```

7-28
numberOfTasks = environmentProperties.NumberOfTasks;

% Specify the command to submit a job to the SLURM scheduler which
% requests as many processes as tasks in the job.
% SLURM will automatically copy environment variables to workers.
commandToRun = sprintf('sbatch --ntasks=%d communicatingJobWrapper.sh', numberOfTasks);
[cmdFailed, cmdOut] = system(commandToRun);

getJobStateFcn

When you query the state of a job created with a generic cluster, the getJobStateFcn.m function executes in the MATLAB client session. The declaration line of this function must be:

function state = getJobStateFcn(cluster,job,state)

When using a third-party scheduler, it is possible that the scheduler can have more up-to-date information about your jobs than what is available to the toolbox from the local job storage location. This situation is especially true if using a nonshared file system, where the remote file system could be slow in propagating large data files back to your local data location.

To retrieve that information from the scheduler, add a function called getJobStateFcn.m to the IntegrationScriptsLocation of your cluster.

The state passed into this function is the state derived from the local job storage. The body of this function can then query the scheduler to determine a more accurate state for the job and return it in place of the stored state. The function you write for this purpose must return a valid value for the state of a job object. Allowed values are ‘pending’, ‘queued’, ‘running’, ‘finished’, or ‘failed’.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see “Managing Jobs with Generic Scheduler” on page 7-33.

cancelJobFcn

When you cancel a job created with a generic cluster, the cancelJobFcn.m function executes in the MATLAB client session. The declaration line of this function must be:

function OK = cancelJobFcn(cluster,job)

When you cancel a job created using the generic scheduler interface, by default this action affects only the job data in storage. To cancel the corresponding jobs on your
scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `cancelJobFcn.m` to the `IntegrationScriptsLocation` of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding jobs from the queue. The function must return a logical scalar indicating the success or failure of canceling the jobs on the scheduler.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see “Managing Jobs with Generic Scheduler” on page 7-33.

**cancelTaskFcn**

When you cancel a task created with a generic cluster, the `cancelTaskFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function OK = cancelTaskFcn(cluster,task)
```

When you cancel a task created using the generic scheduler interface, by default, this affects only the task data in storage. To cancel the corresponding job on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `cancelTaskFcn.m` to the `IntegrationScriptsLocation` of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding job from the scheduler queue. The function must return a logical scalar indicating the success or failure of canceling the job on the scheduler.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see “Managing Jobs with Generic Scheduler” on page 7-33.

**deleteJobFcn**

When you delete a job created with a generic cluster, the `deleteJobFcn.m` function executes in the MATLAB client session. The declaration line of this function must be:

```
function deleteTaskFcn(cluster,task)
```

When you delete a job created using the generic scheduler interface, by default, this affects only the job data in storage. To remove the corresponding jobs on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called `deleteJobFcn.m` to the `IntegrationScriptsLocation` of your cluster.
The body of this function can then send a command to the scheduler, for example, to remove the corresponding jobs from the scheduler queue.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see “Managing Jobs with Generic Scheduler” on page 7-33.

deleteTaskFcn

When you delete a task created with a generic cluster, the deleteTaskFcn.m function executes in the MATLAB client session. The declaration line of this function must be:

```matlab
function deleteTaskFcn(cluster,task)
```

When you delete a task created using the generic scheduler interface, by default, this affects only the task data in storage. To remove the corresponding job on your scheduler, you must provide instructions on what to do and when to do it to the scheduler. To achieve this, add a function called deleteTaskFcn.m to the IntegrationScriptsLocation of your cluster.

The body of this function can then send a command to the scheduler, for example, to remove the corresponding job from the scheduler queue.

For instructions on pairing MATLAB tasks with their corresponding scheduler job ID, see “Managing Jobs with Generic Scheduler” on page 7-33.

postConstructFcn

After you create an instance of your cluster in MATLAB, the postConstructFcn.m function executes in the MATLAB client session. For example, the following line of code creates an instance of your cluster and runs the postConstructFcn function associated with the ‘myProfile’ cluster profile:

```matlab
c = parcluster('myProfile');
```

The declaration line of the postConstructFcn function must be:

```matlab
function postConstructFcn(cluster)
```

If you need to perform custom configuration of your cluster before its use, add a function called postConstructFcn.m to the IntegrationScriptsLocation of your cluster. The body of this function can contain any extra setup steps you require.
Adding User Customization

If you need to modify the functionality of your integration scripts at run time, then use the `AdditionalProperties` property of the generic scheduler interface.

As an example, consider the SLURM scheduler. The submit command for SLURM accepts a `--nodelist` argument that allows you to specify the nodes you want to run on. You can change the value of this argument without having to modify your integration scripts. To add this functionality, include the following code pattern in your `independentSubmitFcn.m` and `communicatingSubmitFcn.m` scripts:

```matlab
% Basic SLURM submit command
submitCommand = 'sbatch';

% Check if property is defined
if isprop(cluster.AdditionalProperties, 'NodeList')
    % Add appropriate argument and value to submit string
    submitCommand = [submitCommand ' --nodelist=' cluster.AdditionalProperties.NodeList];
end
```

For an example of how to use this coding pattern, see the nonshared submit functions of the example support scripts in “Sample Integration Scripts” on page 7-22.

Setting AdditionalProperties from the Cluster Profile Manager

With the modification to your scripts in the previous example, you can add an `AdditionalProperties` entry to your generic cluster profile to specify a list of nodes to use. This provides a method of documenting customization added to your integration scripts for anyone you share the cluster profile with.

To add the `NodeList` property to your cluster profile:

1. Start the Cluster Profile Manager from the MATLAB desktop by selecting **Parallel > Manage Cluster Profiles**.
2. Select the profile for your generic cluster, and click **Edit**.
3. Navigate to the `AdditionalProperties` table, and click **Add**.
4. Enter `NodeList` as the **Name**.
5. Set **String** as the **Type**.
6. Set the **Value** to the list of nodes.
Setting AdditionalProperties from the MATLAB Command Line

With the modification to your scripts in “Adding User Customization” on page 7-32, you can edit the list of nodes from the MATLAB command line by setting the appropriate property of the cluster object before submitting a job:

c = parcluster;
c.AdditionalProperties.NodeList = 'gpuNodeName';
j = c.batch('myScript');

Display the AdditionalProperties object to see all currently defined properties and their values:

```
>> c.AdditionalProperties
ans =
    AdditionalProperties with properties:
            ClusterHost: 'myClusterHost'
            NodeList: 'gpuNodeName'
    RemoteJobStorageLocation: '/tmp/jobs'
```

Managing Jobs with Generic Scheduler

The first requirement for job management is to identify the jobs on the scheduler corresponding to a MATLAB job object. When you submit a job to the scheduler, the command that does the submission in your submit function can return some data about the job from the scheduler. This data typically includes a job ID. By storing that scheduler job ID with the MATLAB job object, you can later refer to the scheduler job by this job ID when you send management commands to the scheduler. Similarly, you can store a map of MATLAB task IDs to scheduler job IDs to help manage individual tasks. The toolbox function that stores this cluster data is setJobClusterData.

Save Job Scheduler Data

This example shows how to modify the independentSubmitFcn.m function to parse the output of each command submitted to a SLURM scheduler. You can use regular expressions to extract the scheduler job ID for each task and then store it using setJobClusterData.

```matlab
% Pattern to extract scheduler job ID from SLURM sbatch output
searchPattern = '.*Submitted batch job ([0-9]+).*';

jobIDs = cell(numberOfTasks, 1);
for ii = 1:numberOfTasks
```


setenv('MDCE_TASK_LOCATION', environmentProperties.TaskLocations{ii});
commandToRun = 'sbatch --ntasks=1 independentJobWrapper.sh';
[cmdFailed, cmdOut] = system(commandToRun);
jobIDs{ii} = regexp(cmdOut, searchPattern, 'tokens', 'once');
end

% set the job IDs on the job cluster data
cluster.setJobClusterData(job, struct('ClusterJobIDs', {jobIDs}));

Retrieve Job Scheduler Data

This example modifies the cancelJobFcn.m to cancel the corresponding jobs on the SLURM scheduler. The example uses getJobClusterData to retrieve job scheduler data.

function OK = cancelJobFcn(cluster, job)

% Get the scheduler information for this job
data = cluster.getJobClusterData(job);
jobIDs = data.ClusterJobIDs;

for ii = 1:length(jobIDs)
    % Tell the SLURM scheduler to cancel the job
    commandToRun = sprintf('scancel ''%s''', jobIDs{ii});
    [cmdFailed, cmdOut] = system(commandToRun);
end

OK = true;

Submitting from a Remote Host

If the MATLAB client is unable to submit directly to your scheduler, use parallel.cluster.RemoteClusterAccess to establish a connection and run commands on a remote host.

This object uses the ssh protocol, and hence requires an ssh daemon service running on the remote host. To establish a connection, you must either provide a user name and password for the remote host, or a valid identity file.

The following code executes a command on a remote host, remoteHostname, as the user, user.

% This will prompt for the password of user
access = parallel.cluster.RemoteClusterAccess.getConnectedAccess('remoteHostname', 'user');
% Execute a command on remoteHostname
[cmdFailed, cmdOut] = access.runCommand(commandToRun);

For an example of integration scripts using remote host submission, see the remote folder of the example support scripts in “Sample Integration Scripts” on page 7-22.

**Submitting without a Shared File System**

If the MATLAB client does not have a shared file system with the cluster nodes, use `parallel.cluster.RemoteClusterAccess` to establish a connection and copy job and task files between the client and cluster nodes.

This object uses the ssh protocol, and hence requires an ssh daemon service running on the remote host. To establish a connection, you must either provide a user name and password for the remote host or a valid identity file.

When using nonshared submission, you must specify both a local job storage location to use on the client and a remote job storage location to use on the cluster. The remote job storage location must be available to all nodes of the cluster.

`parallel.cluster.RemoteClusterAccess` uses file mirroring to continuously synchronize the local job and task files with those on the cluster. When file mirroring first starts, local job and task files are uploaded to the remote job storage location. As the job executes, the file mirroring continuously checks the remote job storage location for new files and updates, and copies the files to the local storage on the client. This procedure ensures the MATLAB client always has an up-to-date view of the jobs and tasks executing on the scheduler.

This example connects to the remote host, `remoteHostname`, as the user, `user`, and establishes `/remote/storage` as the remote cluster storage location to synchronize with. It then starts file mirroring for a job, copying the local files of the job to `/remote/storage` on the cluster, and then syncing any changes back to the local machine.

% This will prompt for the password of user
access = parallel.cluster.RemoteClusterAccess.getConnectedAccessWithMirror('remoteHostname', '/remote/storage', 'user');
% Start file mirroring for a job
access.startMirrorForJob(job);

For an example of integration scripts without a shared file system, see the nonshared folder of the example support scripts in “Sample Integration Scripts” on page 7-22.
See Also

Related Examples
- “Configure Using the Generic Scheduler Interface” (MATLAB Distributed Computing Server)
Program Communicating Jobs

- “Program Communicating Jobs” on page 8-2
- “Program Communicating Jobs for a Supported Scheduler” on page 8-4
- “Program Communicating Jobs for a Generic Scheduler” on page 8-7
- “Further Notes on Communicating Jobs” on page 8-10
Program Communicating Jobs

Communicating jobs are those in which the workers can communicate with each other during the evaluation of their tasks. A communicating job consists of only a single task that runs simultaneously on several workers, usually with different data. More specifically, the task is duplicated on each worker, so each worker can perform the task on a different set of data, or on a particular segment of a large data set. The workers can communicate with each other as each executes its task. The function that the task runs can take advantage of a worker’s awareness of how many workers are running the job, which worker this is among those running the job, and the features that allow workers to communicate with each other.

In principle, you create and run communicating jobs similarly to the way you “Program Independent Jobs” on page 7-2:

1 Define and select a cluster profile.
2 Find a cluster.
3 Create a communicating job.
4 Create a task.
5 Submit the job for running. For details about what each worker performs for evaluating a task, see “Submit a Job to the Job Queue” on page 7-12.
6 Retrieve the results.

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

<table>
<thead>
<tr>
<th>Independent Job</th>
<th>Communicating Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB workers perform the tasks but do not communicate with each other.</td>
<td>MATLAB workers can communicate with each other during the running of their tasks.</td>
</tr>
<tr>
<td>You define any number of tasks in a job.</td>
<td>You define only one task in a job. Duplicates of that task run on all workers running the communicating job.</td>
</tr>
</tbody>
</table>
Some of the details of a communicating job and its tasks might depend on the type of scheduler you are using. The following sections discuss different schedulers and explain programming considerations:

- “Program Communicating Jobs for a Supported Scheduler” on page 8-4
- “Program Communicating Jobs for a Generic Scheduler” on page 8-7
- “Further Notes on Communicating Jobs” on page 8-10
Program Communicating Jobs for a Supported Scheduler

**In this section...**

“Schedulers and Conditions” on page 8-4  
“Code the Task Function” on page 8-4  
“Code in the Client” on page 8-5

**Schedulers and Conditions**

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

To use this supported interface for communicating jobs, the following conditions must apply:

- You must have a shared file system between client and cluster machines
- You must be able to submit jobs directly to the scheduler from the client machine

**Note** When using any third-party scheduler for running a communicating job, if all these conditions are not met, you must use the generic scheduler interface. (Communicating jobs also include **pmode**, **parpool**, **spmd**, and **parfor**.) See “Program Communicating Jobs for a Generic Scheduler” on page 8-7.

**Code the Task Function**

In this section a simple example illustrates the basic principles of programming a communicating job with a third-party scheduler. In this example, the worker whose `labindex` value is 1 creates a magic square comprised of a number of rows and columns that is equal to the number of workers running the job (`numlabs`). In this case, four workers run a communicating job with a 4-by-4 magic square. The first worker broadcasts the matrix with `labBroadcast` to all the other workers, each of which calculates the sum of one column of the matrix. All of these column sums are combined with the `gplus` function to calculate the total sum of the elements of the original magic square.

The function for this example is shown below.
function total_sum = colsum
if labindex == 1
    % Send magic square to other workers
    A = labBroadcast(1,magic(numlabs))
else
    % Receive broadcast on other workers
    A = labBroadcast(1)
end

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

% Calculate total sum by combining column sum from all workers
total_sum = gplus(column_sum)

This function is saved as the file colsum.m on the path of the MATLAB client. It will be sent to each worker by the job’s AttachedFiles property.

While this example has one worker create the magic square and broadcast it to the other workers, there are alternative methods of getting data to the workers. Each worker could create the matrix for itself. Alternatively, each worker could read its part of the data from a file on disk, the data could be passed in as an argument to the task function, or the data could be sent in a file contained in the job’s AttachedFiles property. The solution to choose depends on your network configuration and the nature of the data.

**Code in the Client**

As with independent jobs, you choose a profile and create a cluster object in your MATLAB client by using the `parcluster` function. There are slight differences in the profiles, depending on the scheduler you use, but using profiles to define as many properties as possible minimizes coding differences between the scheduler types.

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

```matlab
c = parcluster('MyProfile')
```

where 'MyProfile' is the name of a cluster profile for the type of scheduler you are using. Any required differences for various cluster options are controlled in the profile. You can have one or more separate profiles for each type of scheduler. For complete details, see “Discover Clusters and Use Cluster Profiles” on page 6-15. Create or modify profiles according to the instructions of your system administrator.
When your cluster object is defined, you create the job object with the
createCommunicatingJob function. The job Type property must be set as 'SPMD'
when you create the job.

cjob = createCommunicatingJob(c,'Type','SPMD');

The function file colsum.m (created in “Code the Task Function” on page 8-4) is on the
MATLAB client path, but it has to be made available to the workers. One way to do this is
with the job's AttachedFiles property, which can be set in the profile you used, or by:

cjob.AttachedFiles = {'colsum.m'}

Here you might also set other properties on the job, for example, setting the number of
workers to use. Again, profiles might be useful in your particular situation, especially if
most of your jobs require many of the same property settings. To run this example on four
workers, you can established this in the profile, or by the following client code:

cjob.NumWorkersRange = 4

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

t = createTask(cjob, @colsum, 1, {})

Use submit to run the job.

submit(cjob)

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

wait(cjob)
results = fetchOutputs(cjob)
results =
    [136]
    [136]
    [136]
    [136]
Program Communicating Jobs for a Generic Scheduler

In this section...

“Introduction” on page 8-7
“Code in the Client” on page 8-7

Introduction

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

The principles of using the generic scheduler interface for communicating jobs are the same as those for independent jobs. The overview of the concepts and details of submit and decode functions for independent jobs are discussed fully in “Integration Scripts for Generic Schedulers” on page 7-21. For this purpose, you need to have Parallel Computing Toolbox as well as MATLAB Distributed Computing Server running on your cluster.

The basic steps follow.

Code in the Client

Configure the Scheduler Object

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

1. Create an object representing your cluster with parcluster.
2. Set the appropriate properties on the cluster object if they are not defined in the profile. Because the scheduler itself is often used by many users and applications, it is probably best to use a profile for programming these properties. See “Discover Clusters and Use Cluster Profiles” on page 6-15.

Among the properties required for a communicating job is IntegrationScriptsLocation. This property points to the location of the communicatingSubmitFcn.m function that executes when a communicating job is submitted. You can write your own communicating submit and decode functions, or use those that come with the product for various schedulers and platforms; see the following section, “Troubleshooting and Debugging” on page 6-55.
3 Use `createCommunicatingJob` to create a communicating job object for your cluster.

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

**Support Scripts**

To support usage of the generic scheduler interface, templates and scripts can be installed from the following locations:

- IBM Platform LSF
- Grid Engine family
- PBS family
- SLURM

Each installer provides templates and scripts for the supported submission modes for shared file system, nonshared file system, or remote submission. Each submission mode has its own subfolder within the installation folder, which contains a file named README that provides specific instructions on how to use the scripts. To use these support scripts with your cluster object, set its `IntegrationScriptsLocation` property to the location of this subfolder.

For more information on programming jobs for generic schedulers, see:

- “Integration Scripts for Generic Schedulers” on page 7-21

For each scheduler type, the folder (or configuration subfolder) contains wrappers, submit functions, and other job management scripts for independent and communicating jobs. For example, you can install the following files for use with a PBS scheduler:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>independentSubmitFcn.m</td>
<td>Submit function for an independent job</td>
</tr>
<tr>
<td>communicatingSubmitFcn.m</td>
<td>Submit function for a communicating job</td>
</tr>
<tr>
<td>independentJobWrapper.sh</td>
<td>Script that is submitted to PBS to start workers that evaluate the tasks of an independent job</td>
</tr>
<tr>
<td>communicatingJobWrapper.sh</td>
<td>Script that is submitted to PBS to start workers that evaluate the tasks of a communicating job</td>
</tr>
<tr>
<td>deleteJobFcn.m</td>
<td>Script to delete a job from the scheduler</td>
</tr>
<tr>
<td>Filename</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>extractJobId.m</td>
<td>Script to get the job’s ID from the scheduler</td>
</tr>
<tr>
<td>getJobStateFcn.m</td>
<td>Script to get the job’s state from the scheduler</td>
</tr>
<tr>
<td>getSubmitString.m</td>
<td>Script to get the submission character vector for the scheduler</td>
</tr>
<tr>
<td>postConstructFcn.m</td>
<td>Script that is run following cluster object creation</td>
</tr>
</tbody>
</table>

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

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

The folders for other scheduler types contain similar files.

As more files or solutions for more schedulers might become available at any time, visit the product page at https://www.mathworks.com/products/distriben/. This Web page provides links to updates, supported schedulers, requirements, and contact information in case you have any questions.
Further Notes on Communicating Jobs

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Number of Tasks in a Communicating Job” on page 8-10</td>
</tr>
<tr>
<td>“Avoid Deadlock and Other Dependency Errors” on page 8-10</td>
</tr>
</tbody>
</table>

Number of Tasks in a Communicating Job

Although you create only one task for a communicating job, the system copies this task for each worker that runs the job. For example, if a communicating job runs on four workers, the Tasks property of the job contains four task objects. The first task in the job’s Tasks property corresponds to the task run by the worker whose labindex is 1, and so on, so that the ID property for the task object and labindex for the worker that ran that task have the same value. Therefore, the sequence of results returned by the fetchOutputs function corresponds to the value of labindex and to the order of tasks in the job’s Tasks property.

Avoid Deadlock and Other Dependency Errors

Because code running in one worker for a communicating job can block execution until some corresponding code executes on another worker, the potential for deadlock exists in communicating jobs. This is most likely to occur when transferring data between workers or when making code dependent upon the labindex in an if statement. Some examples illustrate common pitfalls.

Suppose you have a codistributed array D, and you want to use the gather function to assemble the entire array in the workspace of a single worker.

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

The reason this fails is because the gather function requires communication between all the workers across which the array is distributed. When the if statement limits execution to a single worker, the other workers required for execution of the function are not executing the statement. As an alternative, you can use gather itself to collect the data into the workspace of a single worker: `assembled = gather(D, 1).`
In another example, suppose you want to transfer data from every worker to the next worker on the right (defined as the next higher labindex). First you define for each worker what the workers on the left and right are.

\[
\text{from\_lab\_left} = \text{mod}((\text{labindex} - 2, \text{numlabs}) + 1;
\text{to\_lab\_right} = \text{mod}((\text{labindex}, \text{numlabs}) + 1;
\]

Then try to pass data around the ring.

\[
\text{labSend (outdata, to\_lab\_right);}
\text{indata = labReceive(from\_lab\_left);}
\]

The reason this code might fail is because, depending on the size of the data being transferred, the \text{labSend} function can block execution in a worker until the corresponding receiving worker executes its \text{labReceive} function. In this case, all the workers are attempting to send at the same time, and none are attempting to receive while \text{labSend} has them blocked. In other words, none of the workers get to their \text{labReceive} statements because they are all blocked at the \text{labSend} statement. To avoid this particular problem, you can use the \text{labSendReceive} function.
GPU Computing

- “GPU Capabilities and Performance” on page 9-2
- “Establish Arrays on a GPU” on page 9-3
- “Run MATLAB Functions on a GPU” on page 9-8
- “Run Element-wise MATLAB Code on GPU” on page 9-18
- “Identify and Select a GPU Device” on page 9-23
- “Run CUDA or PTX Code on GPU” on page 9-25
- “Run MEX-Functions Containing CUDA Code” on page 9-36
- “Measure and Improve GPU Performance” on page 9-40
- “GPU Support by Release” on page 9-49
GPU Capabilities and Performance

In this section...

“Capabilities” on page 9-2
“Performance Benchmarking” on page 9-2

Capabilities

Parallel Computing Toolbox enables you to program MATLAB to use your computer’s graphics processing unit (GPU) for matrix operations. In many cases, execution in the GPU is faster than in the CPU, so this feature might offer improved performance.

Toolbox capabilities with the GPU let you:

• “Identify and Select a GPU Device” on page 9-23
• “Create GPU Arrays from Existing Data” on page 9-3
• “Run MATLAB Functions on a GPU” on page 9-8
• “Run Element-wise MATLAB Code on GPU” on page 9-18
• “Run CUDA or PTX Code on GPU” on page 9-25
• “Run MEX-Functions Containing CUDA Code” on page 9-36

Performance Benchmarking

You can use gputimeit to measure the execution time of functions that run on the GPU. For more details, see “Measure and Improve GPU Performance” on page 9-40.

The MATLAB Central file exchange offers a function called gpuBench, which measures the execution time for various MATLAB GPU tasks and estimates the peak performance of your GPU. See https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.
A `gpuArray` in MATLAB represents an array that is stored on the GPU. For a complete list of functions that support arrays on the GPU, see “Run MATLAB Functions on a GPU” on page 9-8.

### Create GPU Arrays from Existing Data

#### Send Arrays to the GPU

GPU arrays can be created by transferring existing arrays from the workspace to the GPU. Use the `gpuArray` function to transfer an array from MATLAB to the GPU:

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

You can accomplish this in a single line of code:

```matlab
G = gpuArray(magic(N));
```

`G` is now a MATLAB `gpuArray` object that represents the magic square stored on the GPU. The input provided to `gpuArray` must be numeric (for example: `single`, `double`, `int8`, etc.) or logical. (See also “Work with Complex Numbers on a GPU” on page 9-15.)

#### Retrieve Arrays from the GPU

Use the `gather` function to retrieve arrays from the GPU to the MATLAB workspace. This takes an array that is on the GPU represented by a `gpuArray` object, and transfers it to the MATLAB workspace as a regular MATLAB array. You can use `isequal` to verify that you get the correct values back:

```matlab
G = gpuArray(ones(100,'uint32'));
D = gather(G);
OK = isequal(D,ones(100,'uint32'))
```
Gathering back to the CPU can be costly, and is generally not necessary unless you need to use your result with functions that do not support `gpuArray`.

**Example: Transfer Array to the GPU**

Create a 1000-by-1000 random matrix in MATLAB, and then transfer it to the GPU:

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

**Example: Transfer Array of a Specified Precision**

Create a matrix of double-precision random values in MATLAB, and then transfer the matrix as single-precision from MATLAB to the GPU:

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

**Create GPU Arrays Directly**

A number of methods of the `gpuArray` class allow you to directly construct arrays on the GPU without having to transfer arrays from the MATLAB workspace. These constructors require only array size and data class information, so they can construct an array without any elements from the workspace.

**Example: Construct an Identity Matrix on the GPU**

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

```matlab
II = eye(1024,'int32','gpuArray');
size(II)
1024 1024
```

With one numerical argument, you create a 2-dimensional matrix.

**Example: Construct a Multidimensional Array on the GPU**

To create a 3-dimensional array of ones with data class `double` on the GPU, type

```matlab
G = ones(100,100,50,'gpuArray');
size(G)
100 100 50
```
The default class of the data is `double`, so you do not have to specify it.

**Example: Construct a Vector on the GPU**

To create a 8192-element column vector of zeros on the GPU, type

```matlab
Z = zeros(8192,1,'gpuArray');
size(Z)
```

```
8192     1
```

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

**Control the Random Stream for `gpuArray`**

On the GPU, the random number generator functions `rand`, `randn`, and `randi` use different settings compared to the client MATLAB session on the CPU. The following functions control the random number stream on the GPU:

- `gpurng`
- `parallel.gpu.RandStream`

These functions perform in the same way as `rng` and `RandStream` in MATLAB, but with certain limitations on the GPU. For more information, see `gpurng`, or type `help parallel.gpu.RandStream`

The GPU uses the Combined Multiplicative Recursive generator ('CombRecursive' or 'mrg32k3a') by default to create uniformly distributed random values, and uses the Inversion method to generate normally distributed values.

To set the client MATLAB session on the CPU to use the same settings as the default settings on the GPU, use the following command:

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

In most cases, it does not matter that the default random number generator on the GPU is not the same as the default generator in MATLAB on the CPU. However, if you need to reproduce the same results on both the GPU and CPU, you can set the CPU random number generator accordingly, and use the same seed for both generators:
seed=0; n=4;

cpu_stream = RandStream('CombRecursive','Seed',seed,'NormalTransform','Inversion');
RandStream.setGlobalStream(cpu_stream);

gpurng('CombRecursive',seed);

r = rand(n); % On CPU
R = rand(n,'gpuArray'); % On GPU
OK = isequal(r,R)

1

When running parallel MATLAB code, the default random number generator settings on worker MATLAB sessions are the same as those of the GPU, even if the workers are in a local cluster on the same machine. That is, a MATLAB client and its workers do not have the same default random number generator settings.

There are three supported random generators on the GPU. The Combined Multiplicative Recursive generator is the default because it is a popular and reliable industry standard generator for parallel computing. The following commands select the GPU random number generator, using the default seed of 0:

gpurng(0,'CombRecursive')
gpurng(0,'Philox')
gpurng(0,'Threefry')

For more information about generating random numbers on a GPU, and a comparison between GPU and CPU generation, see “Control Random Number Streams” on page 6-40. For an example that shows performance comparisons for different random generators, see Generating Random Numbers on a GPU.

**Examine gpuArray Characteristics**

There are several functions available for examining the characteristics of a gpuArray object:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classUnderlying</td>
<td>Class of the underlying data in the array</td>
</tr>
<tr>
<td>existsOnGPU</td>
<td>Indication if array exists on the GPU and is accessible</td>
</tr>
<tr>
<td>isreal</td>
<td>Indication if array data is real</td>
</tr>
<tr>
<td>length</td>
<td>Length of vector or largest array dimension</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>ndims</td>
<td>Number of dimensions in the array</td>
</tr>
<tr>
<td>size</td>
<td>Size of array dimensions</td>
</tr>
</tbody>
</table>

For example, to examine the size of the gpuArray object \( G \), type:

\[
G = \text{rand}(100, 'gpuArray');
\]

\[
s = \text{size}(G)
\]

\[
100 \quad 100
\]

**See Also**

“Run MATLAB Functions on a GPU” on page 9-8 | “Identify and Select a GPU Device” on page 9-23
Run MATLAB Functions on a GPU

MATLAB Functions with gpuArray Arguments

Hundreds of functions in MATLAB and other toolboxes run automatically on GPU if you supply a gpuArray argument.

```matlab
A = gpuArray([1 0 1; -1 -2 0; 0 1 -1]);
e = eig(A);
```

Whenever you call any of these functions with at least one gpuArray as a data input argument, the function executes on the GPU. The function generates a gpuArray as the result, unless returning MATLAB data is more appropriate (for example, `size`). You can mix inputs using both gpuArray and MATLAB arrays in the same function call. To learn more about when a function runs on GPU or CPU, see “Special Conditions for gpuArray Inputs” on page 9-16. GPU-enabled functions include the discrete Fourier transform (`fft`), matrix multiplication (`mtimes`), left matrix division (`mldivide`), and hundreds of others.

Check GPU Supported Functions

If a MATLAB function has GPU support, you can consult additional GPU usage information on its function page. See `GPU Arrays` in the `Extended Capabilities` section at the end of the function page.

You can also browse GPU support for functions, and filter by product. On the Help bar, click `Functions` and select a product. In the function list, at the bottom of the left pane, select `GPU Arrays`.

For information about updates to individual GPU-enabled functions, see the release notes.

Check or Select a GPU

If you have a GPU, then MATLAB automatically uses it for GPU computations. You can check your GPU using the `gpuDevice` function. If you have multiple GPUs, then you can use `gpuDevice` to select one of them, or use multiple GPUs with a parallel pool. For an example, see “Identify and Select a GPU” on page 11-169 and “Use Multiple GPUs in a Parallel Pool” on page 11-171. To check if your GPU is supported, see “GPU Support by Release” on page 9-49.
For deep learning, MATLAB provides automatic parallel support for multiple GPUs. See “Deep Learning with MATLAB on Multiple GPUs” (Deep Learning Toolbox).

**Use MATLAB Functions with a GPU**

This example shows how to use GPU-enabled MATLAB functions to operate with gpuArrays. Check your GPU using the `gpuDevice` function.

Create a row vector. To transfer it to the GPU and create a gpuArray, use the `gpuArray` function.

```
range = [-15:15 0 -15:15 0 -15:15];
n = gpuArray(range);
whos n
```

```
    Name      Size            Bytes  Class       Attributes
    n         1x95                4  gpuArray
```

To operate with gpuArrays, use any GPU-enabled MATLAB function. MATLAB automatically runs calculations on the GPU. For more information, see “Run MATLAB Functions on a GPU” on page 9-8. For example, use a combination of `diag`, `expm`, `mod`, `abs`, and `fliplr`.

```
E = expm(diag(n,-1)) * expm(diag(n,1));
M = mod(abs(E),2);
F = M + fliplr(M);
```

Then, plot the results.

```
imagesc(F);
colormap(flip(gray));
```
If you need to transfer the data back from the GPU, use `gather`. Gathering back to the CPU can be costly, and is generally not necessary unless you need to use your result with functions that do not support `gpuArray`.

```matlab
class = 'GPUArray';
result = gather(F);
whos result
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>96x96</td>
<td>73728</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
Sharpen an Image Using the GPU

This example shows how to sharpen an image using gpuArrays and GPU-enabled functions.

Read the image and send it to the GPU using the gpuArray function.

```matlab
image = gpuArray(imread('peppers.png'));
```

Convert the image to doubles, and apply convolutions to obtain the gradient image. Then, using the gradient image, sharpen the image by a factor of amount.

```matlab
dimage = im2double(image);
gradient = convn(dimage,ones(3)./9,'same') - convn(dimage,ones(5)./25,'same');
amount = 5;
sharpened = dimage + amount.*gradient;
```

Resize, plot and compare the original and sharpened images.

```matlab
imshow(imresize([dimage, sharpened],0.7));
title('Original image (left) vs sharpened image (right)');
```

Compute the Mandelbrot Set using GPU-Enabled Functions
This example shows how to use GPU-enabled MATLAB functions to compute a well-known mathematical construction: the Mandelbrot Set. Check your GPU using the `gpuDevice` function.

Define the parameters. The Mandelbrot algorithm iterates over a grid of real and imaginary parts. The following code defines the number of iterations, grid size and grid limits.

```matlab
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862,  0.123640851045266];
```

You can use the `gpuArray` function to transfer data to the GPU and create a `gpuArray`, or you can create an array directly on the GPU. `gpuArray` provides GPU versions of many functions that you can use to create data arrays, such as `linspace`. For more information, see “Create GPU Arrays Directly” on page 9-4.

```matlab
x = gpuArray.linspace(xlim(1),xlim(2),gridSize);
y = gpuArray.linspace(ylim(1),ylim(2),gridSize);
whos a b
```

Many MATLAB functions support `gpuArrays`. When you supply a `gpuArray` argument to any GPU-enabled function, the function runs automatically on the GPU. For more information, see “Run MATLAB Functions on a GPU” on page 9-8. Create a complex grid for the algorithm, and create an array for the results, `count`. To create this array directly on the GPU, use the `ones` function, and specify `'gpuArray'`.

```matlab
[xGrid,yGrid] = meshgrid(x,y);
z0 = complex(xGrid,yGrid);
count = ones(size(z0),'gpuArray');
```

The following code implements the Mandelbrot algorithm using GPU-enabled functions. Because the code uses `gpuArrays`, the calculations happen in the GPU.

```matlab
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end
count = log(count);
```

When computations are done, plot the results.
Work with Sparse Arrays on a GPU

The following functions support sparse gpuArrays.
You can create a sparse gpuArray either by calling `sparse` with a gpuArray input, or by calling `gpuArray` with a sparse input. For example,

\[
x = \begin{bmatrix} 0 & 1 & 0 & 0 & 0; & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
s = \text{sparse}(x)
\]

\[
(1,2) \quad 1 \\
(2,5) \quad 1
\]

\[
g = \text{gpuArray}(s); \quad \% \; g \; \text{is a sparse gpuArray}
gt = \text{transpose}(g); \quad \% \; gt \; \text{is a sparse gpuArray}
f = \text{full}(gt) \quad \% \; f \; \text{is a full gpuArray}
\]

\[
\begin{bmatrix} 0 & 0 \\
1 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 1
\end{bmatrix}
\]
Work with Complex Numbers on a GPU

If the output of a function running on the GPU could potentially be complex, you must explicitly specify its input arguments as complex. This applies to gpuArray or to functions called in code run by arrayfun.

For example, if creating a gpuArray which might have negative elements, use $G = \text{gpuArray}(\text{complex}(p))$, then you can successfully execute $\sqrt{G}$.

Or, within a function passed to arrayfun, if $x$ is a vector of real numbers, and some elements have negative values, $\sqrt{x}$ will generate an error; instead you should call $\sqrt{\text{complex}(x)}$.

If the result is a gpuArray of complex data and all the imaginary parts are zero, these parts are retained and the data remains complex. This could have an impact when using sort, isreal, etc.

The following table lists the functions that might return complex data, along with the input range over which the output remains real.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Range for Real Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(x)</td>
<td>abs(x) &lt;= 1</td>
</tr>
<tr>
<td>acosh(x)</td>
<td>x &gt;= 1</td>
</tr>
<tr>
<td>acoth(x)</td>
<td>abs(x) &gt;= 1</td>
</tr>
<tr>
<td>acsc(x)</td>
<td>abs(x) &gt;= 1</td>
</tr>
<tr>
<td>asec(x)</td>
<td>abs(x) &gt;= 1</td>
</tr>
<tr>
<td>asech(x)</td>
<td>0 &lt;= x &lt;= 1</td>
</tr>
<tr>
<td>asin(x)</td>
<td>abs(x) &lt;= 1</td>
</tr>
<tr>
<td>atanh</td>
<td>abs(x) &lt;= 1</td>
</tr>
<tr>
<td>log(x)</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>log1p(x)</td>
<td>x &gt;= -1</td>
</tr>
<tr>
<td>log10(x)</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>log2(x)</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>power(x,y)</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>reallog(x)</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>Function</td>
<td>Input Range for Real Output</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>realsqrt(x)</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>x &gt;= 0</td>
</tr>
</tbody>
</table>

**Special Conditions for gpuArray Inputs**

GPU-enabled functions run on the GPU only when the data is on the GPU. For example, the following code runs on GPU because the data, the first input, is on the GPU:

```matlab
>> sum(gpuArray(magic(10)),2);
```

However, this code does not run on GPU because the data, the first input, is not on the GPU:

```matlab
>> sum(magic(10),gpuArray(2));
```

If your input argument gpuArrays contain items such as dimensions, scaling factors, or number of iterations, then the function gathers them and computes on the CPU. Functions only run on the GPU when the actual data arguments are gpuArrays.

**Acknowledgments**

MAGMA is a library of linear algebra routines that take advantage of GPU acceleration. Linear algebra functions implemented for gpuArrays in Parallel Computing Toolbox leverage MAGMA to achieve high performance and accuracy.

**See Also**
gpuArray | gpuDevice

**Related Examples**
- “Identify and Select a GPU Device” on page 9-23
- “Establish Arrays on a GPU” on page 9-3

**More About**
- “GPU Support by Release” on page 9-49
• “GPU Capabilities and Performance” on page 9-2
• MAGMA
Run Element-wise MATLAB Code on GPU

MATLAB Code vs. gpuArray Objects

You have options for performing MATLAB calculations on the GPU:

- You can transfer or create data on the GPU, and use the resulting gpuArray as input to enhanced built-in functions that support them. For more information and a list of functions that support gpuArray as inputs, see “Run MATLAB Functions on a GPU” on page 9-8.

- You can run your own MATLAB function of element-wise operations on a GPU.

Your decision on which solution to adopt depends on whether the functions you require are enhanced to support gpuArray, and the performance impact of transferring data to/from the GPU.

Run Your MATLAB Functions on a GPU

To execute your MATLAB function on a GPU, call arrayfun or bsxfun with a function handle to the MATLAB function as the first input argument:

\[ \text{result} = \text{arrayfun}(@\text{myFunction}, \text{arg1}, \text{arg2}); \]

Subsequent arguments provide inputs to the MATLAB function. These input arguments can be workspace data or gpuArray. If any of the input arguments is a gpuArray, the function executes on the GPU and returns a gpuArray. (If none of the inputs is a gpuArray, then arrayfun and bsxfun execute in the CPU.)

**Note** arrayfun and bsxfun support only element-wise operations on a GPU.
See the `arrayfun` and `bsxfun` reference pages for descriptions of their available options.

**Example: Run Your MATLAB Code**

In this example, a small function applies correction data to an array of measurement data. The function defined in the file `myCal.m` is:

```matlab
function c = myCal(rawdata, gain, offst)
    c = (rawdata .* gain) + offst;
```

The function performs only element-wise operations when applying a gain factor and offset to each element of the `rawdata` array.

Create some nominal measurement:

```matlab
meas = ones(1000)*3; % 1000-by-1000 matrix
```

The function allows the gain and offset to be arrays of the same size as `rawdata`, so that unique corrections can be applied to individual measurements. In a typical situation, you might keep the correction data on the GPU so that you do not have to transfer it for each application:

```matlab
gn   = rand(1000,'gpuArray')/100 + 0.995;
offs = rand(1000,'gpuArray')/50  - 0.01;
```

Run your calibration function on the GPU:

```matlab
corrected = arrayfun(@myCal,meas,gn,offs);
```

This runs on the GPU because the input arguments `gn` and `offs` are already in GPU memory.

Retrieve the corrected results from the GPU to the MATLAB workspace:

```matlab
results = gather(corrected);
```

**Supported MATLAB Code**

The function you pass into `arrayfun` or `bsxfun` can contain the following built-in MATLAB functions and operators:
### Generate Random Numbers on a GPU

The function you pass to `arrayfun` or `bsxfun` for execution on a GPU can contain the random number generator functions `rand`, `randi`, and `randn`. However, the GPU does not support the complete functionality that MATLAB does.

`arrayfun` and `bsxfun` support the following functions for random matrix generation on the GPU:

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rand</code></td>
<td><code>double</code></td>
<td></td>
</tr>
<tr>
<td><code>randi</code></td>
<td><code>int8</code></td>
<td></td>
</tr>
<tr>
<td><code>randn</code></td>
<td><code>single</code></td>
<td></td>
</tr>
</tbody>
</table>

Scalar expansion versions of the following:

- `*`
- `/`
- `\`
- `^`

Branching instructions:

- `break`
- `continue`
- `else`
- `elseif`
- `for`
- `if`
- `return`
- `while`
You do not specify the array size for random generation. Instead, the number of generated random values is determined by the sizes of the input variables to your function. In effect, there will be enough random number elements to satisfy the needs of any input or output variables.

For example, suppose your function `myfun.m` contains the following code that includes generating and using the random matrix `R`:

```matlab
function Y = myfun(X)
    R = rand();
    Y = R.*X;
end
```

If you use `arrayfun` to run this function with an input variable that is a `gpuArray`, the function runs on the GPU, where the number of random elements for `R` is determined by the size of `X`, so you do not need to specify it. The following code passes the `gpuArray` matrix `G` to `myfun` on the GPU.

```matlab
G = 2*ones(4,4, 'gpuArray')
H = arrayfun(@myfun, G)
```

Because `G` is a 4-by-4 `gpuArray`, `myfun` generates 16 random value scalar elements for `R`, one for each calculation with an element of `G`.

Random number generation by `arrayfun` and `bsxfun` on the GPU uses the same global stream as `gpuArray` random generation as described in “Control the Random Stream for `gpuArray`” on page 9-5. For more information about generating random numbers on a GPU, and a comparison between GPU and CPU generation, see “Control Random Number Streams” on page 6-40. For an example that shows performance comparisons for different random generators, see Generating Random Numbers on a GPU.
Tips and Restrictions

The following limitations apply to the code within the function that arrayfun or bsxfun is evaluating on a GPU.

- Like arrayfun in MATLAB, matrix exponential power, multiplication, and division (^, *, /, \) perform element-wise calculations only.
- Operations that change the size or shape of the input or output arrays (cat, reshape, etc.), are not supported.
- When generating random matrices with rand, randi, or randn, you do not need to specify the matrix size, and each element of the matrix has its own random stream. See “Generate Random Numbers on a GPU” on page 9-20.
- arrayfun and bsxfun support read-only indexing (subsref) and access to variables of the parent (outer) function workspace from within nested functions, i.e., those variables that exist in the function before the arrayfun/bsxfun evaluation on the GPU. Assignment or subsasgn indexing of these variables from within the nested function is not supported. For an example of the supported usage see Stencil Operations on a GPU
- Anonymous functions do not have access to their parent function workspace.
- Overloading the supported functions is not allowed.
- The code cannot call scripts.
- There is no ans variable to hold unassigned computation results. Make sure to explicitly assign to variables the results of all calculations that you need to access.
- The following language features are not supported: persistent or global variables, parfor, spmd, switch, and try/catch.
- P-code files cannot contain a call to arrayfun or bsxfun with gpuArray data.
Identify and Select a GPU Device

This example shows how to identify and select a GPU for your computations.

To determine how many GPU devices are available in your computer, use the `gpuDeviceCount` function.

```matlab
gpuDeviceCount
2
```

When there are multiple devices, the first is the default. You can examine its properties with the `gpuDevice` function to determine if that is the one you want to use.

```matlab
d = gpuDevice
d =
CUDADevice with properties:
  Name: 'GeForce GTX 1080'
  Index: 1
  ComputeCapability: '6.1'
  SupportsDouble: 1
  ToolkitVersion: 9
  MaxThreadsPerBlock: 1024
  MaxShmemPerBlock: 49152
  MaxThreadBlockSize: [1024 1024 64]
  MaxGridSize: [2.1475e+09 65535 65535]
  SIMDWidth: 32
  TotalMemory: 8.5899e+09
  AvailableMemory: 7.0053e+09
  MultiprocessorCount: 20
  ClockRateKHz: 1733500
  ComputeMode: 'Default'
  GPUOverlapsTransfers: 1
  KernelExecutionTimeout: 1
  CanMapHostMemory: 1
  DeviceSupported: 1
  DeviceSelected: 1
```

If this is the device you want to use, you can proceed. To run computations on the GPU, use `gpuArray` enabled functions. For more information, see “Run MATLAB Functions on a GPU” on page 9-8.
To use another device, call `gpuDevice` with the index of the other device.

gpuDevice(2)

**See Also**

“Establish Arrays on a GPU” on page 9-3 | “Measure and Improve GPU Performance” on page 9-40 | “Run MATLAB Functions on a GPU” on page 9-8 | “GPU Capabilities and Performance” on page 9-2

**More About**

- “GPU Support by Release” on page 9-49
Run CUDA or PTX Code on GPU

Overview

This topic explains how to create an executable kernel from CU or PTX (parallel thread execution) files, and run that kernel on a GPU from MATLAB. The kernel is represented in MATLAB by a CUDAKernel object, which can operate on MATLAB array or gpuArray variables.

The following steps describe the CUDAKernel general workflow:

1. Use compiled PTX code to create a CUDAKernel object, which contains the GPU executable code.
2. Set properties on the CUDAKernel object to control its execution on the GPU.
3. Call feval on the CUDAKernel with the required inputs, to run the kernel on the GPU.

MATLAB code that follows these steps might look something like this:

```matlab
% 1. Create CUDAKernel object.
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu','entryPt1');

% 2. Set object properties.
k.GridSize = [8 1];
k.ThreadBlockSize = [16 1];

% 3. Call feval with defined inputs.
g1 = gpuArray(in1); % Input gpuArray.
g2 = gpuArray(in2); % Input gpuArray.
result = feval(k,g1,g2);
```

The following sections provide details of these commands and workflow steps.
Create a CUDAKernel Object

- “Compile a PTX File from a CU File” on page 9-26
- “Construct CUDAKernel Object with CU File Input” on page 9-26
- “Construct CUDAKernel Object with C Prototype Input” on page 9-26
- “Supported Data Types” on page 9-27
- “Argument Restrictions” on page 9-28
- “CUDAKernel Object Properties” on page 9-29
- “Specify Entry Points” on page 9-29
- “Specify Number of Threads” on page 9-31

Compile a PTX File from a CU File

If you have a CU file you want to execute on the GPU, you must first compile it to create a PTX file. One way to do this is with the `nvcc` compiler in the NVIDIA® CUDA® Toolkit. For example, if your CU file is called `myfun.cu`, you can create a compiled PTX file with the shell command:

```
nvcc -ptx myfun.cu
```

This generates the file named `myfun.ptx`.

Construct CUDAKernel Object with CU File Input

With a `.cu` file and a `.ptx` file you can create a CUDAKernel object in MATLAB that you can then use to evaluate the kernel:

```
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu');
```

**Note** You cannot save or load CUDAKernel objects.

Construct CUDAKernel Object with C Prototype Input

If you do not have the CU file corresponding to your PTX file, you can specify the C prototype for your C kernel instead of the CU file. For example:

```
k = parallel.gpu.CUDAKernel('myfun.ptx','float *, const float *, float');
```
Another use for C prototype input is when your source code uses an unrecognized renaming of a supported data type. (See the supported types below.) Suppose your kernel comprises the following code.

```c
typedef float ArgType;
__global__ void add3( ArgType * v1, const ArgType * v2 )
{
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

ArgType itself is not recognized as a supported data type, so the CU file that includes it cannot be directly used as input when creating the CUDAKernel object in MATLAB. However, the supported input types to the add3 kernel can be specified as C prototype input to the CUDAKernel constructor. For example:

```matlab
k = parallel.gpu.CUDAKernel('test.ptx','float *, const float *','add3');
```

**Supported Data Types**

The supported C/C++ standard data types are listed in the following table.

<table>
<thead>
<tr>
<th>Float Types</th>
<th>Integer Types</th>
<th>Boolean and Character Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>double, double2</td>
<td>short, unsigned short, short2, ushort2</td>
<td>bool</td>
</tr>
<tr>
<td>float, float2</td>
<td>int, unsigned int, int2, uint2</td>
<td>char, unsigned char, char2, uchar2</td>
</tr>
<tr>
<td></td>
<td>long, unsigned long, long2, ulong2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>long long, unsigned long long, longlong2, ulonglong2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ptrdiff_t, size_t</td>
<td></td>
</tr>
</tbody>
</table>

Also, the following integer types are supported when you include the `tmwtypes.h` header file in your program.
### Integer Types

<table>
<thead>
<tr>
<th>int8_T, int16_T, int32_T, int64_T</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_T, uint16_T, uint32_T, uint64_T</td>
</tr>
</tbody>
</table>

The header file is shipped as `matlabroot/extern/include/tmwtypes.h`. You include the file in your program with the line:

```c
#include "tmwtypes.h"
```

### Argument Restrictions

All inputs can be scalars or pointers, and can be labeled `const`.

The C declaration of a kernel is always of the form:

```c
__global__ void aKernel(inputs ...)
```

- The kernel must return nothing, and operate only on its input arguments (scalars or pointers).
- A kernel is unable to allocate any form of memory, so all outputs must be pre-allocated before the kernel is executed. Therefore, the sizes of all outputs must be known before you run the kernel.
- In principle, all pointers passed into the kernel that are not `const` could contain output data, since the many threads of the kernel could modify that data.

When translating the definition of a kernel in C into MATLAB:

- All scalar inputs in C (`double, float, int, etc.`) must be scalars in MATLAB, or scalar (i.e., single-element) gpuArray variables.
- All `const` pointer inputs in C (`const double *`, etc.) can be scalars or matrices in MATLAB. They are cast to the correct type, copied onto the device, and a pointer to the first element is passed to the kernel. No information about the original size is passed to the kernel. It is as though the kernel has directly received the result of `mxGetData` on an `mxArray`.
- All nonconstant pointer inputs in C are transferred to the kernel exactly as nonconstant pointers. However, because a nonconstant pointer could be changed by the kernel, this will be considered as an output from the kernel.
• Inputs from MATLAB workspace scalars and arrays are cast into the requested type and then passed to the kernel. However, gpuArray inputs are not automatically cast, so their type and complexity must exactly match those expected.

These rules have some implications. The most notable is that every output from a kernel must necessarily also be an input to the kernel, since the input allows the user to define the size of the output (which follows from being unable to allocate memory on the GPU).

**CUDAKernel Object Properties**

When you create a kernel object without a terminating semicolon, or when you type the object variable at the command line, MATLAB displays the kernel object properties. For example:

```matlab
k = parallel.gpu.CUDAKernel('conv.ptx','conv.cu')
```

```
k =
   parallel.gpu.CUDAKernel handle
   Package: parallel.gpu

   Properties:
     ThreadBlockSize: [1 1 1]
     MaxThreadsPerBlock: 512
     GridSize: [1 1 1]
     SharedMemorySize: 0
     EntryPoint: '_Z8theEntryPf'
     MaxNumLHSArguments: 1
     NumRHSArguments: 2
     ArgumentTypes: {'in single vector' 'inout single vector'}
```

The properties of a kernel object control some of its execution behavior. Use dot notation to alter those properties that can be changed.

For a descriptions of the object properties, see the CUDAKernel object reference page. A typical reason for modifying the settable properties is to specify the number of threads, as described below.

**Specify Entry Points**

If your PTX file contains multiple entry points, you can identify the particular kernel in myfun.ptx that you want the kernel object k to refer to:

```matlab
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu','myKernel1');
```
A single PTX file can contain multiple entry points to different kernels. Each of these entry points has a unique name. These names are generally mangled (as in C++ mangling). However, when generated by nvcc the PTX name always contains the original function name from the CU file. For example, if the CU file defines the kernel function as

```c
__global__ void simplestKernelEver( float * x, float val )
```

then the PTX code contains an entry that might be called

```
_Z18simplestKernelEverPff.
```

When you have multiple entry points, specify the entry name for the particular kernel when calling CUDAKernel to generate your kernel.

**Note** The CUDAKernel function searches for your entry name in the PTX file, and matches on any substring occurrences. Therefore, you should not name any of your entries as substrings of any others.

You might not have control over the original entry names, in which case you must be aware of the unique mangled derived for each. For example, consider the following function template.

```c
template <typename T>
__global__ void add4( T * v1, const T * v2 )
{
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

When the template is expanded out for float and double, it results in two entry points, both of which contain the substring add4.

```c
template __global__ void add4<float>(float *, const float *);
template __global__ void add4<double>(double *, const double *);
```

The PTX has corresponding entries:

```
_Z4add4IfEvPT_PKs0
_Z4add4IdEvPT_PKs0
```

Use entry point add4If for the float version, and add4Id for the double version.

```c
k = parallel.gpu.CUDAKernel('test.ptx','double *, const double *', 'add4Id');
```
Specify Number of Threads

You specify the number of computational threads for your CUDAKernel by setting two of its object properties:

- **GridSize** — A vector of three elements, the product of which determines the number of blocks.
- **ThreadBlockSize** — A vector of three elements, the product of which determines the number of threads per block. (Note that the product cannot exceed the value of the property MaxThreadsPerBlock.)

The default value for both of these properties is \([1\ 1\ 1]\), but suppose you want to use 500 threads to run element-wise operations on vectors of 500 elements in parallel. A simple way to achieve this is to create your CUDAKernel and set its properties accordingly:

```matlab
k = parallel.gpu.CUDAKernel('myfun.ptx','myfun.cu');
k.ThreadBlockSize = [500,1,1];
```

Generally, you set the grid and thread block sizes based on the sizes of your inputs. For information on thread hierarchy, and multiple-dimension grids and blocks, see the NVIDIA CUDA C Programming Guide.

Run a CUDAKernel

- “Use Workspace Variables” on page 9-31
- “Use gpuArray Variables” on page 9-32
- “Determine Input and Output Correspondence” on page 9-32

Use the `feval` function to evaluate a CUDAKernel on the GPU. The following examples show how to execute a kernel using MATLAB workspace variables and gpuArray variables.

**Use Workspace Variables**

Assume that you have already written some kernels in a native language and want to use them in MATLAB to execute on the GPU. You have a kernel that does a convolution on two vectors; load and run it with two random input vectors:

```matlab
k = parallel.gpu.CUDAKernel('conv.ptx','conv.cu');
result = feval(k(rand(100,1),rand(100,1)));
Even if the inputs are constants or variables for MATLAB workspace data, the output is `gpuArray`.

**Use `gpuArray` Variables**

It might be more efficient to use `gpuArray` objects as input when running a kernel:

```matlab
k = parallel.gpu.CUDAKernel('conv.ptx','conv.cu');

i1 = gpuArray(rand(100,1,'single'));
i2 = gpuArray(rand(100,1,'single'));

result1 = feval(k,i1,i2);
```

Because the output is a `gpuArray`, you can now perform other operations using this input or output data without further transfers between the MATLAB workspace and the GPU. When all your GPU computations are complete, gather your final result data into the MATLAB workspace:

```matlab
result2 = feval(k,i1,i2);

r1 = gather(result1);
r2 = gather(result2);
```

**Determine Input and Output Correspondence**

When calling `[out1, out2] = feval(kernel, in1, in2, in3)`, the inputs `in1`, `in2`, and `in3` correspond to each of the input arguments to the C function within your CU file. The outputs `out1` and `out2` store the values of the first and second non-const pointer input arguments to the C function after the C kernel has been executed.

For example, if the C kernel within a CU file has the following signature:

```c
void reallySimple( float * pInOut, float c )
```

the corresponding kernel object (`k`) in MATLAB has the following properties:

- `MaxNumLHSArguments`: 1
- `NumRHSArguments`: 2
- `ArgumentTypes`: {'inout single vector'  'in single scalar'}

Therefore, to use the kernel object from this code with `feval`, you need to provide `feval` two input arguments (in addition to the kernel object), and you can use one output argument:
y = feval(k,x1,x2)

The input values x1 and x2 correspond to pIn0ut and c in the C function prototype. The output argument y corresponds to the value of pIn0ut in the C function prototype after the C kernel has executed.

The following is a slightly more complicated example that shows a combination of const and non-const pointers:

void moreComplicated( const float * pIn, float * pIn0ut1, float * pIn0ut2 )

The corresponding kernel object in MATLAB then has the properties:

MaxNumLHSArguments: 2
NumRHSArguments: 3
ArgumentTypes: {'in single vector' 'inout single vector' 'inout single vector'}

You can use feval on this code’s kernel (k) with the syntax:

[y1,y2] = feval(k,x1,x2,x3)

The three input arguments x1, x2, and x3, correspond to the three arguments that are passed into the C function. The output arguments y1 and y2, correspond to the values of pIn0ut1 and pIn0ut2 after the C kernel has executed.

Complete Kernel Workflow

- “Add Two Numbers” on page 9-33
- “Add Two Vectors” on page 9-34
- “Example with CU and PTX Files” on page 9-35

Add Two Numbers

This example adds two doubles together in the GPU. You should have the NVIDIA CUDA Toolkit installed, and have CUDA-capable drivers for your device.

1 The CU code to do this is as follows.

    __global__ void add1( double * pi, double c )
    {
        *pi += c;
    }
The directive __global__ indicates that this is an entry point to a kernel. The code uses a pointer to send out the result in pi, which is both an input and an output. Put this code in a file called test.cu in the current directory.

2 Compile the CU code at the shell command line to generate a PTX file called test.ptx.

    nvcc -ptx test.cu

3 Create the kernel in MATLAB. Currently this PTX file only has one entry so you do not need to specify it. If you were to put more kernels in, you would specify add1 as the entry.

    k = parallel.gpu.CUDAKernel('test.ptx','test.cu');

4 Run the kernel with two numeric inputs. By default, a kernel runs on one thread.

    result = feval(k,2,3)

    result =
    5

**Add Two Vectors**

This example extends the previous one to add two vectors together. For simplicity, assume that there are exactly the same number of threads as elements in the vectors and that there is only one thread block.

1 The CU code is slightly different from the last example. Both inputs are pointers, and one is constant because you are not changing it. Each thread will simply add the elements at its thread index. The thread index must work out which element this thread should add. (Getting these thread- and block-specific values is a very common pattern in CUDA programming.)

    __global__ void add2( double * v1, const double * v2 )
    {
      int idx = threadIdx.x;
      v1[idx] += v2[idx];
    }

Save this code in the file test.cu.

2 Compile as before using nvcc.

    nvcc -ptx test.cu
If this code was put in the same CU file along with the code of the first example, you need to specify the entry point name this time to distinguish it.

```matlab
k = parallel.gpu.CUDAKernel('test.ptx','test.cu','add2');
```

Before you run the kernel, set the number of threads correctly for the vectors you want to add.

```matlab
N = 128;
k.ThreadBlockSize = N;
in1 = ones(N,1,'gpuArray');
in2 = ones(N,1,'gpuArray');
result = feval(k,in1,in2);
```

**Example with CU and PTX Files**

For an example that shows how to work with CUDA, and provides CU and PTX files for you to experiment with, see Illustrating Three Approaches to GPU Computing: The Mandelbrot Set.
Run MEX-Functions Containing CUDA Code

**In this section...**

- “Write a MEX-File Containing CUDA Code” on page 9-36
- “Run the Resulting MEX-Functions” on page 9-37
- “Comparison to a CUDA Kernel” on page 9-37
- “Access Complex Data” on page 9-38
- “Compile a GPU MEX-File” on page 9-38

**Write a MEX-File Containing CUDA Code**

As with any MEX-files, those containing CUDA code have a single entry point, known as `mexFunction`. The MEX-function contains the host-side code that interacts with `gpuArray` objects from MATLAB and launches the CUDA code. The CUDA code in the MEX-file must conform to the CUDA runtime API.

You should call the function `mxInitGPU` at the entry to your MEX-file. This ensures that the GPU device is properly initialized and known to MATLAB.

The interface you use to write a MEX-file for `gpuArray` objects is different from the MEX interface for standard MATLAB arrays.

You can see an example of a MEX-file containing CUDA code at:

`matlabroot/toolbox/distcomp gpu/extern/src/mex/mexGPUExample.cu`

This file contains the following CUDA device function:

```c
void __global__ TimesTwo(double const * const A,
                         double * const B,
                         int const N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        B[i] = 2.0 * A[i];
}
```

It contains the following lines to determine the array size and launch a grid of the proper size:
N = (int)(mxGPUGetNumberOfElements(A));
blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
TimesTwo<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, N);

**Run the Resulting MEX-Functions**

The MEX-function in this example multiplies every element in the input array by 2 to get the values in the output array. To test it, start with a gpuArray in which every element is 1:

```matlab
x = ones(4,4,'gpuArray');
y = mexGPUExample(x)
```

```
y =

2  2  2  2
2  2  2  2
2  2  2  2
2  2  2  2
```

Both the input and output arrays are gpuArray objects:

```matlab
disp(['class(x) = ',class(x),', class(y) = ',class(y)])
class(x) = gpuArray, class(y) = gpuArray
```

**Comparison to a CUDA Kernel**

Parallel Computing Toolbox also supports CUDAKernel objects that can be used to integrate CUDA code with MATLAB. Consider the following when choosing the MEX-file approach versus the CUDAKernel approach:

- MEX-files can interact with host-side libraries, such as the NVIDIA Performance Primitives (NPP) or CUFFT libraries, and can also contain calls from the host to functions in the CUDA runtime library.
- MEX-files can analyze the size of the input and allocate memory of a different size, or launch grids of a different size, from C or C++ code. In comparison, MATLAB code that calls CUDAKernel objects must preallocate output memory and determine the grid size.
**Access Complex Data**

Complex data on a GPU device is stored in interleaved complex format. That is, for a complex gpuArray `A`, the real and imaginary parts of element `i` are stored in consecutive addresses. MATLAB uses CUDA built-in vector types to store complex data on the device (see the NVIDIA CUDA C Programming Guide).

Depending on the needs of your kernel, you can cast the pointer to complex data either as the real type or as the built-in vector type. For example, in MATLAB, suppose you create a matrix:

```matlab
a = complex(ones(4,'gpuArray'),ones(4,'gpuArray'));
```

If you pass a gpuArray to a MEX-function as the first argument (prhs[0]), then you can get a pointer to the complex data by using the calls:

```matlab
mxGPUArray const * A = mxGPUCreateFromMxArray(prhs[0]);
mwSize numel_complex = mxGPUGetNumberOfElements(A);
double2 * d_A = (double2 const *)(mxGPUGetDataReadOnly(A));
```

To treat the array as a real double-precision array of twice the length, you could do it this way:

```matlab
mxGPUArray const * A = mxGPUCreateFromMxArray(prhs[0]);
mwSize numel_real = 2*mxGPUGetNumberOfElements(A);
double * d_A = (double const *)(mxGPUGetDataReadOnly(A));
```

Various functions exist to convert data between complex and real formats on the GPU. These operations require a copy to interleave the data. The function `mxGPUCreateComplexGPUArray` takes two real mxGPUArrays and interleaves their elements to produce a single complex mxGPUArray of the same length. The functions `mxGPUCopyReal` and `mxGPUCopyImag` each copy either the real or the imaginary elements into a new real mxGPUArray. (There is no equivalent of the `mxGetImagData` function for mxGPUArray objects.)

**Compile a GPU MEX-File**

To compile CUDA code you must have installed the CUDA toolkit version consistent with the `ToolkitVersion` property of the GPUDevice object.

Use the `mexcuda` command in MATLAB to compile a MEX-file containing the CUDA code. You can compile the example file using the command:
mexcuda mexGPUExample.cu

If `mexcuda` has trouble locating the NVIDIA compiler (`nvcc`), it might be installed in a non-default location. You can specify the location of `nvcc` on your system by storing it in the environment variable `MW_NVCC_PATH`. You can set this variable using the MATLAB `setenv` command. For example,

```
setenv('MW_NVCC_PATH','/usr/local/CUDA/bin')
```

Only a subset of Visual Studio® compilers is supported for `mexcuda`. For details, consult the NVIDIA toolkit documentation.

**See Also**

**Related Examples**

- “Accessing Advanced CUDA Features Using MEX”
Measure and Improve GPU Performance

Getting Started with GPU Benchmarking

You can use various benchmark tests in MATLAB to measure the performance of your GPU:

- Use gpuBench in MATLAB Central File Exchange to do various tests, including both memory and compute intensive tasks in both single and double precision. Compare the performance of a display card with a compute card. For more information, see https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.
- Use the paralleldemo_gpu_bench script in “Measuring GPU Performance” to obtain information on your PCI bus speed, GPU memory read/write and peak calculation performances for double precision matrix calculations.

Improve Performance Using Single Precision Calculations

You can improve the performance of your GPU by doing your calculations in single precision instead of double precision. In CPU computations, on the other hand, you do not get this improvement when switching from double to single precision. The reason is that most GPU cards are designed for graphic display, demanding high single precision performance.

Typical examples of calculations suitable for single-precision computation on the GPU include image processing and machine learning, see e.g. https://www.mathworks.com/deep-learning-in-the-cloud-with-matlab-white-paper. However, other types of calculations, such as linear algebra problems, typically require double precision processing.

You can get a performance improvement of up to a factor of 50 for single compared to double precision calculations, depending on the GPU card and total number of cores. High end compute cards typically show a smaller improvement. You can determine the performance improvement of your particular GPU by using gpuBench, see https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench.

For a comprehensive performance overview of NVIDIA GPU cards, see https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units. You can calculate the performance improvement factor between single precision and double precision as follows:
• Find the GPU on the wiki page above.
• Get the stated single and double precision performance values from the table. If there is no double precision GFLOPS value, assume the ratio is 24-32x slower for double precision.
• Divide the stated single precision GFLOPS value by the double precision GFLOPS value.

**Note** If you have a mobile graphics card in your laptop, you can use this card for GPU computing. However, the laptop GPU is likely to be much less powerful than the desktop machine equivalent and so performance is reduced.

**Basic Workflow for Improving Performance**

The purpose of GPU computing in MATLAB is to speed up your applications. This topic discusses fundamental concepts and practices that can help you achieve better performance on the GPU, such as the configuration of the GPU hardware and best practices within your code. It discusses the trade-off between implementation difficulty and performance, and describes the criteria you might use to choose between using gpuArray functions, arrayfun, MEX-files, or CUDA kernels. Finally, it describes how to accurately measure performance on the GPU.

When converting MATLAB code to run on the GPU, it is best to start with MATLAB code that already performs well. While the GPU and CPU have different performance characteristics, the general guidelines for writing good MATLAB code also help you write good MATLAB code for the GPU. The first step is almost always to profile your CPU code. The lines of code that the profiler shows taking the most time on the CPU will likely be ones that you must concentrate on when you code for the GPU.

It is easiest to start converting your code using MATLAB built-in functions that support gpuArray data. These functions take gpuArray inputs, perform calculations on the GPU, and return gpuArray outputs. A list of the MATLAB functions that support gpuArray data is found in “Run MATLAB Functions on a GPU” on page 9-8. In general, these functions support the same arguments and data types as standard MATLAB functions that are calculated on the CPU.

If all the functions that you want to use are supported on the GPU, running code on the GPU may be as simple as calling gpuArray to transfer input data to the GPU, and calling gather to retrieve the output data from the GPU when finished. In many cases, you
might need to vectorize your code, replacing looped scalar operations with MATLAB matrix and vector operations. While vectorizing is generally a good practice on the CPU, it is usually critical for achieving high performance on the GPU. For more information, see “Vectorize for Improved GPU Performance” on page 9-45.

**Advanced Tools for Improving Performance**

It is possible that even after converting inputs to gpuArrays and vectorizing your code, there are operations in your algorithm that are either not built-in functions, or that are not fast enough to meet your application’s requirements. In such situations you have three main options: use `arrayfun` to precompile element-wise parts of your application, make use of GPU library functions, or write a custom CUDA kernel.

If you have a purely element-wise function, you can improve its performance by calling it with `arrayfun`. The `arrayfun` function on the GPU turns an element-wise MATLAB function into a custom CUDA kernel, thus reducing the overhead of performing the operation. Often, there is a subset of your application that can be used with `arrayfun` even if the entire application cannot be. The example Improve Performance of Element-wise MATLAB Functions on the GPU using ARRAYFUN shows the basic concepts of this approach; and the example Using ARRAYFUN for Monte-Carlo Simulations shows how this can be done in simulations for a finance application.

MATLAB provides an extensive library of GPU-enabled functions in Parallel Computing Toolbox, Image Processing Toolbox™, Signal Processing Toolbox™, and other products. However, there are many libraries of additional functions that do not have direct built-in analogs in MATLAB’s GPU support. Examples include the NVIDIA Performance Primitives library and the CURAND library, which are included in the CUDA toolkit that ships with MATLAB. If you need to call a function in one of these libraries, you can do so using the GPU MEX interface. This interface allows you to extract the pointers to the device data from MATLAB gpuArrays so that you can pass these pointers to GPU functions. You can convert the returned values into gpuArrays for return to MATLAB. For more information see “Run MEX-Functions Containing CUDA Code” on page 9-36.

Finally, you have the option of writing a custom CUDA kernel for the operation that you need. Such kernels can be directly integrated into MATLAB using the CUDAKernel object.

The example Illustrating Three Approaches to GPU Computing: The Mandelbrot Set shows how to implement a simple calculation using three of the approaches mentioned in this section. This example begins with MATLAB code that is easily converted to run on the GPU, rewrites the code to use `arrayfun` for element-wise operations, and finally shows how to integrate a custom CUDA kernel for the same operation.
Alternately, you can write a CUDA kernel as part of a MEX-file and call it using the CUDA Runtime API inside the MEX-file. Either of these approaches might let you work with low-level features of the GPU, such as shared memory and texture memory, that are not directly available in MATLAB code. For more details, see the example Accessing Advanced CUDA Features Using MEX.

**Best Practices for Improving Performance**

**Hardware Configuration**

In general you can achieve the best performance when your GPU is dedicated to computing. It is usually not practical to use the same GPU device for both computations and graphics, because of the amount of memory taken up for problems of reasonable size and the constant use of the device by the system for graphics. If possible, obtain a separate device for graphics. Details of configuring your device for compute or graphics depend on the operating system and driver version.

On Windows systems, a GPU device can be in one of two modes: Windows Display Driver Model (WDDM) or Tesla Compute Cluster (TCC) mode. For best performance, any devices used for computing should be in TCC mode. Consult NVIDIA documentation for more details.

NVIDIA’s highest-performance compute devices, the Tesla line, support error correcting codes (ECC) when reading and writing GPU memory. The purpose of ECC is to correct for occasional bit-errors that occur normally when reading or writing dynamic memory. One technique to improve performance is to turn off ECC to increase the achievable memory bandwidth. While the hardware can be configured this way, MathWorks does not recommend this practice. The potential loss of accuracy due to silent errors can be more harmful than the performance benefit.

**MATLAB Coding Practices**

This topic describes general techniques that help you achieve better performance on the GPU. Some of these tips apply when writing MATLAB code for the CPU as well.

Data in MATLAB arrays is stored in column-major order. Therefore, it is beneficial to operate along the first or column dimension of your array. If one dimension of your data is significantly longer than others, you might achieve better performance if you make that the first dimension. Similarly, if you frequently operate along a particular dimension, it is usually best to have it as the first dimension. In some cases, if consecutive operations
target different dimensions of an array, it might be beneficial to transpose or permute the array between these operations.

GPUs achieve high performance by calculating many results in parallel. Thus, matrix and higher-dimensional array operations typically perform much better than operations on vectors or scalars. You can achieve better performance by rewriting your loops to make use of higher-dimensional operations. The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called vectorization. For more details, see “Using Vectorization” (MATLAB).

By default, all operations in MATLAB are performed in double-precision floating-point arithmetic. However, most operations support a variety of data types, including integer and single-precision floating-point. Today’s GPUs and CPUs typically have much higher throughput when performing single-precision operations, and single-precision floating-point data occupies less memory. If your application’s accuracy requirements allow the use of single-precision floating-point, it can greatly improve the performance of your MATLAB code.

The GPU sits at the end of a data transfer mechanism known as the PCI bus. While this bus is an efficient, high-bandwidth way to transfer data from the PC host memory to various extension cards, it is still much slower than the overall bandwidth to the global memory of the GPU device or of the CPU (for more details, see the example Measuring GPU Performance). In addition, transfers from the GPU device to MATLAB host memory cause MATLAB to wait for all pending operations on the device to complete before executing any other statements. This can significantly hurt the performance of your application. In general, you should limit the number of times you transfer data between the MATLAB workspace and the GPU. If you can transfer data to the GPU once at the start of your application, perform all the calculations you can on the GPU, and then transfer the results back into MATLAB at the end, that generally results in the best performance. Similarly, when possible it helps to create arrays directly on the GPU, using either the 'gpuArray' or the 'like' option for functions such as zeros (e.g., \( Z = \text{zeros}(__,'\text{gpuArray}') \) or \( Z = \text{zeros}(N,'\text{like}',g) \) for existing gpuArray \( g \)).

**Measure Performance on the GPU**

The best way to measure performance on the GPU is to use `gputimeit`. This function takes as input a function handle with no input arguments, and returns the measured execution time of that function. It takes care of such benchmarking considerations as repeating the timed operation to get better resolution, executing the function before measurement to avoid initialization overhead, and subtracting out the overhead of the
timing function. Also, `gputimeit` ensures that all operations on the GPU have completed before the final timing.

For example, consider measuring the time taken to compute the `lu` factorization of a random matrix `A` of size `N`-by-`N`. You can do this by defining a function that does the `lu` factorization and passing the function handle to `gputimeit`:

```matlab
A = rand(N,'gpuArray');
fh = @( ) lu(A);
gputimeit(fh,2); % 2nd arg indicates number of outputs
```

You can also measure performance with `tic` and `toc`. However, to get accurate timing on the GPU, you must wait for operations to complete before calling `toc`. There are two ways to do this. You can call `gather` on the final GPU output before calling `toc`: this forces all computations to complete before the time measurement is taken. Alternately, you can use the `wait` function with a `GPUDevice` object as its input. For example, if you wanted to measure the time taken to compute the `lu` factorization of matrix `A` using `tic`, `toc`, and `wait`, you can do it as follows:

```matlab
gd = gpuDevice();
tic();
[l,u] = lu(A);
wait(gd);
tLU = toc();
```

You can also use the MATLAB profiler to show how computation time is distributed in your GPU code. Note, that to accomplish timing measurements, the profiler runs each line of code independently, so it cannot account for overlapping (asynchronous) execution such as might occur during normal operation. For timing whole algorithms, you should use `tic` and `toc`, or `gputimeit`, as described above. Also, the profile might not yield correct results for user-defined MEX functions if they run asynchronously.

### Vectorize for Improved GPU Performance

This example shows you how to improve performance by running a function on the GPU instead of the CPU, and by vectorizing the calculations.

Consider a function that performs fast convolution on the columns of a matrix. Fast convolution, which is a common operation in signal processing applications, transforms each column of data from the time domain to the frequency domain, multiplies it by the transform of a filter vector, transforms back to the time domain, and stores the result in an output matrix.
function y = fastConvolution(data,filter)
    [m,n] = size(data);
    % Zero-pad filter to the column length of data, and transform
    filter_f = fft(filter,m);
    
    % Create an array of zeros of the same size and class as data
    y = zeros(m,n,'like',data);
    
    % Transform each column of data
    for ix = 1:n
        af = fft(data(:,ix));
        y(:,ix) = ifft(af .* filter_f);
    end
end

Execute this function in the CPU on data of a particular size, and measure the execution
time using the MATLAB timeit function. The timeit function takes care of common
benchmarking considerations, such as accounting for startup and overhead.

a = complex(randn(4096,100),randn(4096,100));  % Data input
b = randn(16,1);                               % Filter input
c = fastConvolution(a,b);                      % Calculate output
c = timeit(@()fastConvolution(a,b));          % Measure CPU time
disp(['Execution time on CPU = ',num2str(ctime)]);

On a sample machine, this code displays the output:

Execution time on CPU = 0.019335

Now execute this function on the GPU. You can do this easily by changing the input data
to be gpuArrays rather than normal MATLAB arrays. The 'like' syntax used when
creating the output inside the function ensures that y will be a gpuArray if data is a
gpuArray.

ga = gpuArray(a);                              % Move array to GPU
gb = gpuArray(b);                              % Move filter to GPU
gc = fastConvolution(ga,gb);                   % Calculate on GPU
gc = gputimeit(@()fastConvolution(a,b));       % Measure GPU time
gc = max(abs(gather(gc)-c));                   % Calculate error
disp(['Execution time on GPU = ',num2str(gctime)]);
disp(['Maximum absolute error = ',num2str(gerr)]);

On the same machine, this code displays the output:
Unfortunately, the GPU is slower than the CPU for this problem. The reason is that the for-loop is executing the FFT, multiplication, and inverse FFT operations on individual columns of length 4096. The best way to increase the performance is to vectorize the code, so that a single MATLAB function call performs more calculation. The FFT and IFFT operations are easy to vectorize: `fft(A)` computes the FFT of each column of a matrix A. You can perform a multiply of the filter with every column in a matrix at once using the MATLAB binary scalar expansion function `bsxfun`. The vectorized function looks like this:

```matlab
function y = fastConvolution_v2(data, filter)
    m = size(data,1);
    % Zero-pad filter to the length of data, and transform
    filter_f = fft(filter, m);

    % Transform each column of the input
    af = fft(data);

    % Multiply each column by filter and compute inverse transform
    y = ifft(bsxfun(@times, af, filter_f));
end
```

Perform the same experiment using the vectorized function:

```matlab
a = complex(randn(4096,100),randn(4096,100));   % Data input
b = randn(16,1);                                % Filter input
c = fastConvolution_v2(a, b);                   % Calculate output
c.timeit(@()fastConvolution_v2(a, b));          % Measure CPU time
disp(['Execution time on CPU = ', num2str(c.time)]);

ga = gpuArray(a);                               % Move data to GPU
gb = gpuArray(b);                                % Move filter to GPU
gc = fastConvolution_v2(ga, gb);                % Calculate on GPU
g.timeit(@()fastConvolution_v2(ga, gb));        % Measure GPU time
gerr = max(max(abs(gather(gc)-c)));             % Calculate error
disp(['Execution time on GPU = ', num2str(g.time)]);
disp(['Maximum absolute error = ', num2str(gerr)]);
```

Execution time on CPU = 0.010393
Execution time on GPU = 0.0020537
Maximum absolute error = 1.1374e-14
In conclusion, vectorizing the code helps both the CPU and GPU versions to run faster. However, vectorization helps the GPU version much more than the CPU. The improved CPU version is nearly twice as fast as the original; the improved GPU version is 13 times faster than the original. The GPU code went from being 40% slower than the CPU in the original version, to about five times faster in the revised version.

**Troubleshooting GPUs**

If you only have one GPU in your machine, then it is likely that your graphics card is also acting as your display card. In this case, your GPU is probably subject to timeout imposed by the operating system (OS). You can examine this for your GPU as follows:

```matlab
gpuDevice
ans =
...

KernelExecutionTimeout: 1
```

If `KernelExecutionTimeout = 1`, then your GPU is subject to timeout imposed by the OS, ensuring that the OS is always able to print updates to the screen. If your GPU calculation takes too much time, then the operation is killed. In this case, you must restart MATLAB to resume GPU calculations successfully.

**See Also**

“GPU Capabilities and Performance” on page 9-2 | “Establish Arrays on a GPU” on page 9-3 | “Run MATLAB Functions on a GPU” on page 9-8 | “Identify and Select a GPU Device” on page 9-23 | `gpuDevice`
GPU Support by Release

Consult the following table to see support for NVIDIA GPU architectures by MATLAB release.

• ✔ - Built-in binary support.

• ✔️ - Supported by NVIDIA's *forward compatibility* (requires recompilation). The MATLAB release was built before this GPU architecture was available. The CUDA driver must recompile the GPU libraries because your device is more recent than the libraries. The first time you access the GPU from MATLAB, the compilation can take several minutes. Increase the CUDA cache size to prevent a recurrence of this delay. For instructions, see “Increase the CUDA Cache Size” on page 9-50.

The *cc* numbers show the compute capability of the GPU architecture. To check your GPU compute capability, see `ComputeCapability` in the output of the `gpuDevice` function. Alternatively, see CUDA GPUs (NVIDIA). The CUDA Toolkit version is the version of the CUDA libraries used. Each CUDA Toolkit requires a CUDA driver of the same version or later. If you want to create your own CUDA enabled MEX-functions or CUDAKernel objects, install this version of the CUDA Toolkit. For more information, see “Run MEX-Functions Containing CUDA Code” on page 9-36 and “Run CUDA or PTX Code on GPU” on page 9-25.

<table>
<thead>
<tr>
<th>MATLAB Release</th>
<th>CUDA Toolkit</th>
<th>Tesla (cc1.3)</th>
<th>Fermi (cc2.x)</th>
<th>Kepler (cc3.x)</th>
<th>Maxwell (cc5.x)</th>
<th>Pascal (cc6.x)</th>
<th>Volta (cc7.x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2010b</td>
<td>3.1</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2011a</td>
<td>3.2</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2011b</td>
<td>4.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2012a</td>
<td>4.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2012b</td>
<td>4.2</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2013a</td>
<td>5.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>MATLAB Release</td>
<td>CUDA Toolkit</td>
<td>Tesla (cc1.3)</td>
<td>Fermi (cc2.x)</td>
<td>Kepler (cc3.x)</td>
<td>Maxwell (cc5.x)</td>
<td>Pascal (cc6.x)</td>
<td>Volta (cc7.x)</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>----------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>R2013b</td>
<td>5.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2014a</td>
<td>5.5</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2014b</td>
<td>6.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2015a</td>
<td>6.5</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2015b</td>
<td>7.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2016a</td>
<td>7.5</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2016b</td>
<td>7.5</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2017a</td>
<td>8.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2017b</td>
<td>8.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2018a</td>
<td>9.0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>R2018b</td>
<td>9.1</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Increase the CUDA Cache Size**

If your GPU architecture does not have built-in binary support in your MATLAB release, the graphics driver must compile and cache the GPU libraries. This process can take a few minutes the first time you access the GPU from MATLAB. To increase the CUDA cache size to prevent a recurrence of this delay, set the environment variable `CUDA_CACHE_MAXSIZE` to a minimum of 536870912 (512MB). For help setting an environment variable, see this example: “Set the MATLABPATH Environment Variable” (MATLAB).
See Also

Related Examples
• “Identify and Select a GPU Device” on page 9-23

External Websites
• Deep Learning with GPUs and MATLAB
Objects — Alphabetical List
**codistributed**

Access elements of arrays distributed among workers in parallel pool

**Constructor**

codistributed, codistributed.build

You can also create a codistributed array explicitly from spmd code or a communicating job task with any of several MATLAB functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eye(___,'codistributed')</td>
<td>rand(___,'codistributed')</td>
</tr>
<tr>
<td>false(___,'codistributed')</td>
<td>randi(___,'codistributed')</td>
</tr>
<tr>
<td>Inf(___,'codistributed')</td>
<td>randn(___,'codistributed')</td>
</tr>
<tr>
<td>NaN(___,'codistributed')</td>
<td>codistributed.cell</td>
</tr>
<tr>
<td>ones(___,'codistributed')</td>
<td>codistributed.spalloc</td>
</tr>
<tr>
<td>true(___,'codistributed')</td>
<td>codistributed.speye</td>
</tr>
<tr>
<td>zeros(___,'codistributed')</td>
<td>codistributed.sprand</td>
</tr>
<tr>
<td></td>
<td>codistributed.sprandn</td>
</tr>
</tbody>
</table>

**Description**

Arrays partitioned among the workers in a pool, are accessible from the workers as codistributed array objects.

Codistributed arrays on workers that you create inside spmd statements or from within task functions of communicating jobs can be accessed as distributed arrays on the client.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classUnderlying</td>
<td>Class of elements within gpuArray or distributed array</td>
</tr>
<tr>
<td>codistributed.cell</td>
<td>Create codistributed cell array</td>
</tr>
<tr>
<td>codistributed.colon</td>
<td>Distributed colon operation</td>
</tr>
<tr>
<td>codistributed.spalloc</td>
<td>Allocate space for sparse codistributed matrix</td>
</tr>
<tr>
<td>codistributed.speye</td>
<td>Create codistributed sparse identity matrix</td>
</tr>
<tr>
<td>codistributed.sprand</td>
<td>Create codistributed sparse array of uniformly distributed pseudo-random values</td>
</tr>
<tr>
<td>codistributed.sprandn</td>
<td>Create codistributed sparse array of uniformly distributed pseudo-random values</td>
</tr>
<tr>
<td>eye</td>
<td>Identity matrix</td>
</tr>
<tr>
<td>false</td>
<td>Array of logical 0 (false)</td>
</tr>
<tr>
<td>gather</td>
<td>Transfer distributed array or gpuArray to local workspace</td>
</tr>
<tr>
<td>getCodistributor</td>
<td>Codistributor object for existing codistributed array</td>
</tr>
<tr>
<td>getLocalPart</td>
<td>Local portion of codistributed array</td>
</tr>
<tr>
<td>globalIndices</td>
<td>Global indices for local part of codistributed array</td>
</tr>
<tr>
<td>Inf</td>
<td>Array of infinity</td>
</tr>
<tr>
<td>isaUnderlying</td>
<td>True if distributed array's underlying elements are of specified class</td>
</tr>
<tr>
<td>isc codistributed</td>
<td>True for codistributed array</td>
</tr>
<tr>
<td>NaN</td>
<td>Array of Not-a-Numbers</td>
</tr>
<tr>
<td>ones</td>
<td>Array of ones</td>
</tr>
<tr>
<td>rand</td>
<td>Array of rand values</td>
</tr>
<tr>
<td>randi</td>
<td>Array of random integers</td>
</tr>
<tr>
<td>randn</td>
<td>Array of randn values</td>
</tr>
<tr>
<td>redistribute</td>
<td>Redistribute codistributed array with another distribution scheme</td>
</tr>
<tr>
<td>sparse</td>
<td>Create sparse distributed or codistributed matrix</td>
</tr>
<tr>
<td>true</td>
<td>Array of logical 1 (true)</td>
</tr>
<tr>
<td>zeros</td>
<td>Array of zeros</td>
</tr>
</tbody>
</table>
The methods for codistributed arrays are too numerous to list here. Most resemble and behave the same as built-in MATLAB functions. See “Run MATLAB Functions with Distributed Arrays” on page 5-24.

Also among the methods there are several for examining the characteristics of the array itself. Most behave like the MATLAB functions of the same name:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classUnderlying</code></td>
<td>Class (data type) of the elements in the array</td>
</tr>
<tr>
<td><code>iscodistributed</code></td>
<td>Indication if array is codistributed</td>
</tr>
<tr>
<td><code>isreal</code></td>
<td>Indication if array elements are real</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Length of vector or largest array dimension</td>
</tr>
<tr>
<td><code>ndims</code></td>
<td>Number of dimensions in the array</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Size of array dimensions</td>
</tr>
</tbody>
</table>

**Introduced in R2008b**
codistributor1d

1-D distribution scheme for codistributed array

Constructor
codistributor1d

Description
A codistributor1d object defines the 1-D distribution scheme for a codistributed array. The 1-D codistributor distributes arrays along a single specified dimension, the distribution dimension, in a noncyclic, partitioned manner.

For help on codistributor1d, including a list of links to individual help for its methods and properties, type

help codistributor1d

Methods
codistributor1d.defaultPartition
Default partition for codistributed array
globalIndices
Global indices for local part of codistributed array
isComplete
True if codistributor object is complete

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension</td>
<td>Distributed dimension of codistributor1d object</td>
</tr>
</tbody>
</table>

10-5
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>Partition scheme of codistributor1d object</td>
</tr>
</tbody>
</table>

Introduced in R2009b
**codistributor2dbc**

2-D block-cyclic distribution scheme for codistributed array

**Constructor**

codistributor2dbc

**Description**

A codistributor2dbc object defines the 2-D block-cyclic distribution scheme for a codistributed array. The 2-D block-cyclic codistributor can only distribute two-dimensional matrices. It distributes matrices along two subscripts over a rectangular computational grid of labs in a blocked, cyclic manner. The parallel matrix computation software library called ScaLAPACK uses the 2-D block-cyclic codistributor.

For help on codistributor2dbc, including a list of links to individual help for its methods and properties, type

help codistributor2dbc

**Methods**

- **codistributor2dbc.defaultLabGrid**
  
  Default computational grid for 2-D block-cyclic distributed arrays

- **globalIndices**
  
  Global indices for local part of codistributed array

- **isComplete**
  
  True if codistributor object is complete
## Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockSize</td>
<td>Block size of codistributor2dbc object</td>
</tr>
<tr>
<td>LabGrid</td>
<td>Lab grid of codistributor2dbc object</td>
</tr>
<tr>
<td>Orientation</td>
<td>Orientation of codistributor2dbc object</td>
</tr>
</tbody>
</table>

**Introduced in R2009b**
Composite

Access nondistributed variables on multiple workers from client

Constructor

Composite

Description

Variables that exist on the workers running an `spmd` statement are accessible on the client as a Composite object. A Composite resembles a cell array with one element for each worker. So for Composite C:

\[
\begin{align*}
C\{1\} & \text{ represents value of } C \text{ on worker1} \\
C\{2\} & \text{ represents value of } C \text{ on worker2} \\
\text{etc.}
\end{align*}
\]

`spmd` statements create Composites automatically, which you can access after the statement completes. You can also create a Composite explicitly with the `Composite` function.

Methods

exist \hspace{1cm} \text{Check whether Composite is defined on workers}
subsasgn \hspace{1cm} \text{Subscripted assignment for Composite}
subsref \hspace{1cm} \text{Subscripted reference for Composite}

Other methods of a Composite object behave similarly to these MATLAB array functions:

disp, display \hspace{1cm} \text{Display Composite}
end \hspace{1cm} \text{Indicate last Composite index}
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isempty</td>
<td>Determine whether Composite is empty</td>
</tr>
<tr>
<td>length</td>
<td>Length of Composite</td>
</tr>
<tr>
<td>ndims</td>
<td>Number of Composite dimensions</td>
</tr>
<tr>
<td>numel</td>
<td>Number of elements in Composite</td>
</tr>
<tr>
<td>size</td>
<td>Composite dimensions</td>
</tr>
</tbody>
</table>

**Introduced in R2008b**
**CUDAKernel**

Kernel executable on GPU

**Constructor**

`parallel.gpu.CUDAKernel`

**Description**

A `CUDAKernel` object represents a CUDA kernel, that can execute on a GPU. You create the kernel when you compile PTX or CU code, as described in “Run CUDA or PTX Code on GPU” on page 9-25.

**Methods**

- `existsOnGPU` Determine if gpuArray or CUDAKernel is available on GPU
- `feval` Evaluate kernel on GPU
- `setConstantMemory` Set some constant memory on GPU

**Properties**

A `CUDAKernel` object has the following properties:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadBlockSize</td>
<td>Size of block of threads on the kernel. This can be an integer vector of length 1, 2, or 3 (since thread blocks can be up to 3-dimensional). The product of the elements of <code>ThreadBlockSize</code> must not exceed the <code>MaxThreadsPerBlock</code> for this kernel, and no element of <code>ThreadBlockSize</code> can exceed the corresponding element of the <code>GPUDevice</code> property <code>MaxThreadBlockSize</code>.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MaxThreadsPerBlock</td>
<td>Maximum number of threads permissible in a single block for this CUDA kernel. The product of the elements of ThreadBlockSize must not exceed this value.</td>
</tr>
<tr>
<td>GridSize</td>
<td>Size of grid (effectively the number of thread blocks that will be launched independently by the GPU). This is an integer vector of length 3. None of the elements of this vector can exceed the corresponding element in the vector of the MaxGridSize property of the GPUDevice object.</td>
</tr>
<tr>
<td>SharedMemorySize</td>
<td>The amount of dynamic shared memory (in bytes) that each thread block can use. Each thread block has an available shared memory region. The size of this region is limited in current cards to ~16 kB, and is shared with registers on the multiprocessors. As with all memory, this needs to be allocated before the kernel is launched. It is also common for the size of this shared memory region to be tied to the size of the thread block. Setting this value on the kernel ensures that each thread in a block can access this available shared memory region.</td>
</tr>
<tr>
<td>EntryPoint</td>
<td>(read-only) A character vector containing the actual entry point name in the PTX code that this kernel is going to call. An example might look like '_Z13returnPointerPKfPy'.</td>
</tr>
<tr>
<td>MaxNumLHSArguments</td>
<td>(read-only) The maximum number of left hand side arguments that this kernel supports. It cannot be greater than the number of right hand side arguments, and if any inputs are constant or scalar it will be less.</td>
</tr>
<tr>
<td>NumRHSArguments</td>
<td>(read-only) The required number of right hand side arguments needed to call this kernel. All inputs need to define either the scalar value of an input, the elements for a vector input/output, or the size of an output argument.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ArgumentTypes</td>
<td>(read-only) Cell array of character vectors, the same length as NumRHSArguments. Each of the character vectors indicates what the expected MATLAB type for that input is (a numeric type such as uint8, single, or double followed by the word scalar or vector to indicate if we are passing by reference or value). In addition, if that argument is only an input to the kernel, it is prefixed by in; and if it is an input/output, it is prefixed by inout. This allows you to decide how to efficiently call the kernel with both MATLAB arrays and gpuArray, and to see which of the kernel inputs are being treated as outputs.</td>
</tr>
</tbody>
</table>

**See Also**

gpuArray, GPUDevice

**Introduced in R2011b**
distributed

Access elements of distributed arrays from client

Constructor

distributed

You can also create a distributed array explicitly from the client with any of several MATLAB functions, shown in the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eye(____,'distributed')</td>
<td>true(____,'distributed')</td>
</tr>
<tr>
<td>false(____,'distributed')</td>
<td>zeros(____,'distributed')</td>
</tr>
<tr>
<td>Inf(____,'distributed')</td>
<td>distributed.cell</td>
</tr>
<tr>
<td>NaN(____,'distributed')</td>
<td>distributed.spalloc</td>
</tr>
<tr>
<td>ones(____,'distributed')</td>
<td>distributed.speye</td>
</tr>
<tr>
<td>rand(____,'distributed')</td>
<td>distributed.sprand</td>
</tr>
<tr>
<td>randi(____,'distributed')</td>
<td>distributed.sprandn</td>
</tr>
<tr>
<td>randn(____,'distributed')</td>
<td></td>
</tr>
</tbody>
</table>

Description

Distributed arrays represent those arrays which are partitioned out among the workers in a parallel pool. A distributed array resembles a normal MATLAB array in the way you index and manipulate its elements, but none of its elements exists on the client.

Codistributed arrays that you create inside `spmd` statements are accessible as distributed arrays from the client.

Use the `gather` function to retrieve distributed arrays into the client work space.
Methods

classUnderlying: Class of elements within gpuArray or distributed array
distributed.cell: Create distributed cell array
distributed.spalloc: Allocate space for sparse distributed matrix
distributed.speye: Create distributed sparse identity matrix
distributed.sprand: Create distributed sparse array of uniformly distributed pseudo-random values
distributed.sprandn: Create distributed sparse array of normally distributed pseudo-random values
distributed.sprandn: Create distributed sparse array of normally distributed pseudo-random values
eye: Identity matrix
false: Array of logical 0 (false)
gather: Transfer distributed array or gpuArray to local workspace
Inf: Array of infinity
isaUnderlying: True if distributed array's underlying elements are of specified class
isdistributed: True for distributed array
NaN: Array of Not-a-Numbers
ones: Array of ones
rand: Array of rand values
randi: Array of random integers
randn: Array of randn values
sparse: Create sparse distributed or codistributed matrix
true: Array of logical 1 (true)
write: Write distributed data to an output location
zeros: Array of zeros

The methods for distributed arrays are too numerous to list here. Most resemble and behave the same as built-in MATLAB functions. See “Run MATLAB Functions with Distributed Arrays” on page 5-24.

Also among the methods are several for examining the characteristics of the array itself. Most behave like the MATLAB functions of the same name:
<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>classUnderlying</td>
<td>Class (data type) of the elements in the array</td>
</tr>
<tr>
<td>isdistributed</td>
<td>Indication if array is distributed</td>
</tr>
<tr>
<td>isreal</td>
<td>Indication if array elements are real</td>
</tr>
<tr>
<td>length</td>
<td>Length of vector or largest array dimension</td>
</tr>
<tr>
<td>ndims</td>
<td>Number of dimensions in the array</td>
</tr>
<tr>
<td>size</td>
<td>Size of array dimensions</td>
</tr>
</tbody>
</table>

**Introduced in R2008a**
gpuArray

Array stored on GPU

Constructor

gpuArray converts an array in the MATLAB workspace into a gpuArray with elements stored on the GPU device.

Also, the following create gpuArrays:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eye(__,'gpuArray')</td>
<td>converts an array into a gpuArray</td>
</tr>
<tr>
<td>false(__,'gpuArray')</td>
<td>converts an array into a gpuArray</td>
</tr>
<tr>
<td>Inf(__,'gpuArray')</td>
<td>converts an array into a gpuArray</td>
</tr>
<tr>
<td>NaN(__,'gpuArray')</td>
<td>converts an array into a gpuArray</td>
</tr>
<tr>
<td>ones(__,'gpuArray')</td>
<td>initializes a gpuArray with ones</td>
</tr>
<tr>
<td>true(__,'gpuArray')</td>
<td>initializes a gpuArray with true</td>
</tr>
<tr>
<td>zeros(__,'gpuArray')</td>
<td>initializes a gpuArray with zeros</td>
</tr>
<tr>
<td>rand(__,'gpuArray')</td>
<td>initializes a gpuArray with random numbers</td>
</tr>
<tr>
<td>randi(__,'gpuArray')</td>
<td>initializes a gpuArray with random integers</td>
</tr>
<tr>
<td>randn(__,'gpuArray')</td>
<td>initializes a gpuArray with random numbers</td>
</tr>
</tbody>
</table>

For class specific help on the three methods with the gpuArray prefix, type

help gpuArray.methodname

where methodname is the name of the method. For example, to get help on colon, type

help gpuArray.colon

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

- gpurng
- parallel.gpu.RandStream
Description

A gpuArray object represents an array stored on the GPU. You can use the array for direct calculations, or in CUDA kernels that execute on the GPU. You can return the array to the MATLAB workspace with the gather function.

Methods

arrayfun  Apply function to each element of array on GPU
bsxfun    Binary singleton expansion function for gpuArray
classUnderlying  Class of elements within gpuArray or distributed array
existsOnGPU  Determine if gpuArray or CUDAKernel is available on GPU
eye       Identity matrix
false      Array of logical 0 (false)
gather    Transfer distributed array or gpuArray to local workspace
Inf       Array of infinity
NaN        Array of Not-a-Numbers
ones      Array of ones
pagefun   Apply function to each page of array on GPU
rand      Array of rand values
randi     Array of random integers
randn     Array of randn values
true      Array of logical 1 (true)
zeros     Array of zeros

Other methods for a gpuArray object are too numerous to list here. Most resemble and behave the same as built-in MATLAB functions. See “Run MATLAB Functions on a GPU” on page 9-8.

Among the gpuArray methods there are several for examining the characteristics of a gpuArray object. Most behave like the MATLAB functions of the same name:
<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>classUnderlying</td>
<td>Class (data type) of the elements in the array</td>
</tr>
<tr>
<td>existsOnGPU</td>
<td>Indication if array exists on the GPU and is accessible</td>
</tr>
<tr>
<td>isreal</td>
<td>Indication if array elements are real</td>
</tr>
<tr>
<td>length</td>
<td>Length of vector or largest array dimension</td>
</tr>
<tr>
<td>ndims</td>
<td>Number of dimensions in the array</td>
</tr>
<tr>
<td>size</td>
<td>Size of array dimensions</td>
</tr>
</tbody>
</table>

**See Also**

CUDAKernel, GPUDevice

**Introduced in R2012b**
**GPUDevice**

Graphics processing unit (GPU)

**Constructor**

gpuDevice

**Description**

A GPUDevice object represents a graphic processing unit (GPU) in your computer. You can use the GPU to execute CUDA kernels or MATLAB code.

**Methods**

The following functions let you identify, select, reset, or wait for a GPU device:

- `gpuDevice` Query or select GPU device
- `gpuDeviceCount` Number of GPU devices present
- `reset` Reset GPU device and clear its memory
- `wait (GPUDevice)` Wait for GPU calculation to complete

Methods of the class include the following:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parallel.gpu.GPUDevice.isAvailable(idx)</code></td>
<td>True if the GPU specified by index <code>idx</code> is supported and capable of being selected. <code>idx</code> can be an integer or a vector of integers; the default index is the current device.</td>
</tr>
<tr>
<td><code>parallel.gpu.GPUDevice.getDevice(idx)</code></td>
<td>Returns a GPUDevice object without selecting it.</td>
</tr>
</tbody>
</table>
For the complete list, use the methods function on the GPUDevice class:

```matlab
methods('parallel.gpu.GPUDevice')
```

You can get help on any of the class methods with the command

```matlab
help parallel.gpu.GPUDevice.methodname
```

where `methodname` is the name of the method. For example, to get help on `isAvailable`, type

```matlab
help parallel.gpu.GPUDevice.isAvailable
```

## Properties

A GPUDevice object has the following read-only properties:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the CUDA device.</td>
</tr>
<tr>
<td>Index</td>
<td>Index by which you can select the device.</td>
</tr>
<tr>
<td>ComputeCapability</td>
<td>Computational capability of the CUDA device. Must meet required specification.</td>
</tr>
<tr>
<td>SupportsDouble</td>
<td>Indicates if this device can support double precision operations.</td>
</tr>
<tr>
<td>DriverVersion</td>
<td>The CUDA device driver version currently in use. Must meet required specification.</td>
</tr>
<tr>
<td>ToolkitVersion</td>
<td>Version of the CUDA toolkit used by the current release of MATLAB.</td>
</tr>
<tr>
<td>MaxThreadsPerBlock</td>
<td>Maximum supported number of threads per block during CUDAKernel execution.</td>
</tr>
<tr>
<td>MaxShmemPerBlock</td>
<td>Maximum supported amount of shared memory that can be used by a thread block during CUDAKernel execution.</td>
</tr>
<tr>
<td>MaxThreadBlockSize</td>
<td>Maximum size in each dimension for thread block. Each dimension of a thread block must not exceed these dimensions. Also, the product of the thread block size must not exceed MaxThreadsPerBlock.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MaxGridSize</td>
<td>Maximum size of grid of thread blocks.</td>
</tr>
<tr>
<td>SIMDWidth</td>
<td>Number of simultaneously executing threads.</td>
</tr>
<tr>
<td>TotalMemory</td>
<td>Total memory (in bytes) on the device.</td>
</tr>
<tr>
<td>AvailableMemory</td>
<td>Total amount of memory (in bytes) available for data. This property is available only for the currently selected device.</td>
</tr>
<tr>
<td>MultiprocessorCount</td>
<td>The number of vector processors present on the device.</td>
</tr>
<tr>
<td>ClockRateKHz</td>
<td>Peak clock rate of the GPU in kHz.</td>
</tr>
<tr>
<td>ComputeMode</td>
<td>The compute mode of the device, according to the following values: 'Default' — The device is not restricted and can be used by multiple applications simultaneously. MATLAB can share the device with other applications, including other MATLAB sessions or workers. 'Exclusive thread' or 'Exclusive process' — The device can be used by only one application at a time. While the device is selected in MATLAB, it cannot be used by other applications, including other MATLAB sessions or workers. 'Prohibited' — The device cannot be used.</td>
</tr>
<tr>
<td>GPUOverlapsTransfers</td>
<td>Indicates if the device supports overlapped transfers.</td>
</tr>
<tr>
<td>KernelExecutionTimeout</td>
<td>Indicates if the device can abort long-running kernels. If true, the operating system places an upper bound on the time allowed for the CUDA kernel to execute, after which the CUDA driver times out the kernel and returns an error.</td>
</tr>
<tr>
<td>CanMapHostMemory</td>
<td>Indicates if the device supports mapping host memory into the CUDA address space.</td>
</tr>
<tr>
<td>DeviceSupported</td>
<td>Indicates if toolbox can use this device. Not all devices are supported; for example, if their ComputeCapability is insufficient, the toolbox cannot use them.</td>
</tr>
<tr>
<td>DeviceSelected</td>
<td>Indicates if this is the currently selected device.</td>
</tr>
</tbody>
</table>

**See Also**

CUDAKernel, gpuArray
Introduced in R2010b
**GPUDeviceManager**

Manager for GPU Devices

**Constructor**

`parallel.gpu.GPUDeviceManager.instance`

**Description**

`parallel.gpu.GPUDeviceManager` provides events that indicate when a GPU device has been selected or deselected. These events also fire when a GPU device is reset. There is only a single instance of the `parallel.gpu.GPUDeviceManager` available in a given MATLAB session, and it is obtained using the method `parallel.gpu.GPUDeviceManager.instance`.

**Events**

Events of the class include the following:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceSelected</td>
<td>Fired after a GPU device is selected.</td>
</tr>
<tr>
<td>DeviceDeselecting</td>
<td>Fired just before a GPU device is deselected.</td>
</tr>
</tbody>
</table>

**Properties**

A `GPUDeviceManager` object has one property:
### Property Name

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectedDevice</td>
<td>Contains the currently selected GPU Device.</td>
</tr>
</tbody>
</table>

### Methods

Methods of the class include the following:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDeviceCount</td>
<td>Returns the number of GPU devices available.</td>
</tr>
<tr>
<td>selectDevice</td>
<td>Selects a GPU device.</td>
</tr>
</tbody>
</table>

### See Also

gpuDevice, gpuDeviceCount, gpuArray

*Introduced in R2016a*
mxGPUArray

Type for MATLAB gpuArray

Description

mxGPUArray is an opaque C language type that allows a MEX function access to the elements in a MATLAB gpuArray. Using the mxGPU API, you can perform calculations on a MATLAB gpuArray, and return gpuArray results to MATLAB.

All MEX functions receive inputs and pass outputs as mxArrays. A gpuArray in MATLAB is a special kind of mxArray that represents an array stored on the GPU. In your MEX function, you use mxGPUArray objects to access an array stored on the GPU: these objects correspond to MATLAB gpuArrays.

The mxGPU API contains functions that manipulate mxGPUArray objects. These functions allow you to extract mxGPUArrays from input mxArrays, to wrap output mxGPUArrays as mxArrays for return to MATLAB, to determine the characteristics of the arrays, and to get pointers to the underlying elements. You can perform calculations by passing the pointers to CUDA functions that you write or that are available in external libraries.

The basic structure of a GPU MEX function is:

1. Call `mxInitGPU` to initialize MathWorks GPU library.
2. Determine which mxArray inputs contain GPU data.
3. Create mxGPUArray objects from the input mxArray arguments, and get pointers to the input elements on the device.
4. Create mxGPUArray objects to hold the outputs, and get the pointers to the output elements on the device.
5. Call a CUDA function, passing it the device pointers.
6. Wrap the output mxGPUArray as an mxArray for return to MATLAB.
7. Destroy the mxGPUArray objects you created.

The header file that contains this type is `mxGPUArray.h`. You include it with the line:

```c
#include "gpu/mxGPUArray.h"
```
See Also

gpuArray, mxArray

Introduced in R2013a
**parallel.Cluster**

Access cluster properties and behaviors

**Constructors**

parcluster
gGetCurrentCluster (in the workspace of the MATLAB worker)

**Container Hierarchy**

<table>
<thead>
<tr>
<th>Parent</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children</td>
<td>parallel.Job, parallel.Pool</td>
</tr>
</tbody>
</table>

**Description**

A parallel.Cluster object provides access to a cluster, which controls the job queue, and distributes tasks to workers for execution.

**Types**

The two categories of clusters are the MATLAB job scheduler (MJS) and common job scheduler (CJS). The MJS is available in the MATLAB Distributed Computing Server. The CJS clusters encompass all other types, including the local, generic, and third-party schedulers.

Use MJSCluster objects to interact with MATLAB Distributed Computing Server for Amazon EC2® clusters.

The following table describes the available types of cluster objects.
<table>
<thead>
<tr>
<th>Cluster Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel.cluster.MJS</td>
<td>Interact with MATLAB job scheduler (MJS) cluster on-premises</td>
</tr>
<tr>
<td>parallel.cluster.MJSComputeCloud</td>
<td>Interact with MATLAB Distributed Computing Server for Amazon EC2 cluster</td>
</tr>
<tr>
<td>parallel.cluster.Local</td>
<td>Interact with CJS cluster running locally on client machine</td>
</tr>
<tr>
<td>parallel.cluster.HPCServer</td>
<td>Interact with CJS cluster running Windows Microsoft HPC Server</td>
</tr>
<tr>
<td>parallel.cluster.LSF</td>
<td>Interact with CJS cluster running Platform LSF</td>
</tr>
<tr>
<td>parallel.cluster.PBSPro</td>
<td>Interact with CJS cluster running Altair PBS Pro</td>
</tr>
<tr>
<td>parallel.cluster.Torque</td>
<td>Interact with CJS cluster running TORQUE</td>
</tr>
<tr>
<td>parallel.cluster.Slurm</td>
<td>Interact with CJS cluster running Slurm</td>
</tr>
<tr>
<td>parallel.cluster.Generic</td>
<td>Interact with CJS cluster using the generic interface</td>
</tr>
</tbody>
</table>

**Methods**

**Common to All Cluster Types**

- **batch**: Run MATLAB script or function on worker
- **createCommunicatingJob**: Create communicating job on cluster
- **createJob**: Create independent job on cluster
- **findJob**: Find job objects stored in cluster
- **isequal**: True if clusters have same property values
- **parpool**: Create parallel pool on cluster
- **saveAsProfile**: Save cluster properties to specified profile
- **saveProfile**: Save modified cluster properties to its current profile
MJS

changePassword  Prompt user to change MJS password
demote          Demote job in cluster queue
logout          Log out of MJS cluster
pause           Pause MATLAB job scheduler queue
promote         Promote job in MJS cluster queue
resume          Resume processing queue in MATLAB job scheduler

MJSComputeCloud

shutdown        Shut down cloud cluster
start           Start cloud cluster
wait (cluster)  Wait for cloud cluster to change state

HPC Server, PBS Pro, LSF, TORQUE, Slurm, and Local Clusters

getDebugLog     Read output messages from job run in CJS cluster

Generic

getDebugLog     Read output messages from job run in CJS cluster
getJobClusterData Get specific user data for job on generic cluster
getJobFolder    Folder on client where jobs are stored
getJobFolderOnCluster Folder on cluster where jobs are stored
getLogLocation  Log location for job or task
setJobClusterData Set specific user data for job on generic cluster
Properties

Common to All Cluster Types

The following properties are common to all cluster object types.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterMatlabRoot</td>
<td>Specifies path to MATLAB for workers to use</td>
</tr>
<tr>
<td>Host</td>
<td>Host name of the cluster head node</td>
</tr>
<tr>
<td>JobStorageLocation</td>
<td>Location where cluster stores job and task information</td>
</tr>
<tr>
<td>Jobs</td>
<td>List of jobs contained in this cluster</td>
</tr>
<tr>
<td>LicenseNumber</td>
<td>License number to use when running jobs with this cluster</td>
</tr>
<tr>
<td>Modified</td>
<td>True if any properties in this cluster have been modified</td>
</tr>
<tr>
<td>NumThreads</td>
<td>Number of computational threads for workers</td>
</tr>
<tr>
<td>NumWorkers</td>
<td>Number of workers available for this cluster</td>
</tr>
<tr>
<td>OperatingSystem</td>
<td>Operating system of nodes used by cluster</td>
</tr>
<tr>
<td>Profile</td>
<td>Profile used to build this cluster</td>
</tr>
<tr>
<td>RequiresMathWorksHostedLicensing</td>
<td>True if the cluster is using the MathWorks Hosted License Manager</td>
</tr>
<tr>
<td>Type</td>
<td>Type of this cluster</td>
</tr>
<tr>
<td>UserData</td>
<td>Information associated with cluster object within client session</td>
</tr>
</tbody>
</table>

MJS

MJS cluster objects have the following properties in addition to the common properties:
<table>
<thead>
<tr>
<th><strong>Property</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AllHostAddresses</td>
<td>IP addresses of the cluster host</td>
</tr>
<tr>
<td>BusyWorkers</td>
<td>Workers currently running tasks</td>
</tr>
<tr>
<td>IdleWorkers</td>
<td>Workers currently available for running tasks</td>
</tr>
<tr>
<td>HasSecureCommunication</td>
<td>True if cluster is using secure communication</td>
</tr>
<tr>
<td>Name</td>
<td>Name of this cluster</td>
</tr>
<tr>
<td>NumBusyWorkers</td>
<td>Number of workers currently running tasks</td>
</tr>
<tr>
<td>NumIdleWorkers</td>
<td>Number of workers available for running tasks</td>
</tr>
<tr>
<td>PromptForPassword</td>
<td>True if system should prompt for password when authenticating user</td>
</tr>
<tr>
<td>SecurityLevel</td>
<td>Degree of security applied to cluster and its jobs. For descriptions of security levels, see “Set MJS Cluster Security” (MATLAB Distributed Computing Server).</td>
</tr>
<tr>
<td>State</td>
<td>Current state of cluster</td>
</tr>
<tr>
<td>Username</td>
<td>User accessing cluster</td>
</tr>
</tbody>
</table>

**MJSComputeCloud**

MJSComputeCloud cluster objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th><strong>Property</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyWorkers</td>
<td>Workers currently running tasks</td>
</tr>
<tr>
<td>Certificate</td>
<td>Cluster SSL certificate</td>
</tr>
<tr>
<td>HasSecureCommunication</td>
<td>True if cluster is using secure communication</td>
</tr>
<tr>
<td>Identifier</td>
<td>Unique cluster identifier</td>
</tr>
<tr>
<td>IdleWorkers</td>
<td>Workers currently available for running tasks</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MatlabVersion</td>
<td>Version of MATLAB running on the workers</td>
</tr>
<tr>
<td>Name</td>
<td>Name of this cluster</td>
</tr>
<tr>
<td>NumBusyWorkers</td>
<td>Number of workers currently running tasks</td>
</tr>
<tr>
<td>NumIdleWorkers</td>
<td>Number of workers available for running tasks</td>
</tr>
<tr>
<td>ShutdownAt</td>
<td>Shutdown time or event</td>
</tr>
<tr>
<td>State</td>
<td>Current state of cluster</td>
</tr>
<tr>
<td>Username</td>
<td>User accessing cluster</td>
</tr>
</tbody>
</table>

**Local**

Local cluster objects have no editable properties beyond the properties common to all clusters.

**HPC Server**

HPC Server cluster objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterVersion</td>
<td>Version of Microsoft Windows HPC Server running on the cluster</td>
</tr>
<tr>
<td>HasSharedFilesystem</td>
<td>Specify whether client and cluster nodes share JobStorageLocation</td>
</tr>
<tr>
<td>JobDescriptionFile</td>
<td>Name of XML job description file to use when creating jobs</td>
</tr>
<tr>
<td>JobTemplate</td>
<td>Name of job template to use for jobs submitted to HPC Server</td>
</tr>
<tr>
<td>Name</td>
<td>Name of this cluster</td>
</tr>
<tr>
<td>UseSOAJobSubmission</td>
<td>Allow service-oriented architecture (SOA) submission on HPC Server</td>
</tr>
</tbody>
</table>
PBS Pro and TORQUE

PBS Pro and TORQUE cluster objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommunicatingJobWrapper</td>
<td>Script that cluster runs to start workers</td>
</tr>
<tr>
<td>RcpCommand</td>
<td>Command to copy files to and from client</td>
</tr>
<tr>
<td>ResourceTemplate</td>
<td>Define resources to request for communicating jobs</td>
</tr>
<tr>
<td>RshCommand</td>
<td>Remote execution command used on worker nodes during communicating job</td>
</tr>
<tr>
<td>HasSharedFilesystem</td>
<td>Specify whether client and cluster nodes share JobStorageLocation</td>
</tr>
<tr>
<td>SubmitArguments</td>
<td>Specify additional arguments to use when submitting jobs</td>
</tr>
</tbody>
</table>

LSF

LSF cluster objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterName</td>
<td>Name of Platform LSF cluster</td>
</tr>
<tr>
<td>CommunicatingJobWrapper</td>
<td>Script that the cluster runs to start workers</td>
</tr>
<tr>
<td>HasSharedFilesystem</td>
<td>Specify whether client and cluster nodes share JobStorageLocation</td>
</tr>
<tr>
<td>SubmitArguments</td>
<td>Specify additional arguments to use when submitting jobs</td>
</tr>
</tbody>
</table>

Slurm

Slurm cluster objects have the following properties in addition to the common properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterName</td>
<td>Name of the Slurm cluster</td>
</tr>
<tr>
<td>CommunicatingJobWrapper</td>
<td>Script that the cluster runs to start workers</td>
</tr>
<tr>
<td>SubmitArguments</td>
<td>Specify additional arguments to use when submitting jobs</td>
</tr>
</tbody>
</table>

**Generic**

If you create a generic cluster object from an R2017a or later profile, you have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdditionalProperties</td>
<td>Additional properties for integration scripts</td>
</tr>
<tr>
<td>HasSharedFilesystem</td>
<td>Specify whether client and cluster nodes share JobStorageLocation</td>
</tr>
<tr>
<td>IntegrationScriptsLocation</td>
<td>Folder containing scheduler integration scripts</td>
</tr>
</tbody>
</table>

If you create a generic cluster object from an R2016b or earlier profile, you have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CancelJobFcn</td>
<td>Function to run when cancelling job</td>
</tr>
<tr>
<td>CancelTaskFcn</td>
<td>Function to run when cancelling task</td>
</tr>
<tr>
<td>CommunicatingSubmitFcn</td>
<td>Function to run when submitting communicating job</td>
</tr>
<tr>
<td>DeleteJobFcn</td>
<td>Function to run when deleting job</td>
</tr>
<tr>
<td>DeleteTaskFcn</td>
<td>Function to run when deleting task</td>
</tr>
<tr>
<td>GetJobStateFcn</td>
<td>Function to run when querying job state</td>
</tr>
<tr>
<td>HasSharedFilesystem</td>
<td>Specify whether client and cluster nodes share JobStorageLocation</td>
</tr>
<tr>
<td>IndependentSubmitFcn</td>
<td>Function to run when submitting independent job</td>
</tr>
</tbody>
</table>
Help

For further help on cluster objects, including links to help for specific cluster types and object properties, type:

help parallel.Cluster

See Also


Introduced in R2012a
**parallel.cluster.Hadoop**

Hadoop cluster for mapreduce, mapreduce and tall arrays

**Constructors**

`parallel.cluster.Hadoop`

**Description**

A `parallel.cluster.Hadoop` object provides access to a cluster for configuring mapreduce, mapreduce, and tall arrays.

**Properties**

A `parallel.cluster.Hadoop` object has the following properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdditionalPaths</td>
<td>Paths to be added to MATLAB command search path on workers</td>
</tr>
<tr>
<td>AttachedFiles</td>
<td>Files transferred to the workers during a mapreduce call</td>
</tr>
<tr>
<td>AutoAttachFiles</td>
<td>Specifies whether automatically attach files</td>
</tr>
<tr>
<td>ClusterMatlabRoot</td>
<td>Specifies path to MATLAB for workers to use</td>
</tr>
<tr>
<td>HadoopConfigurationFile</td>
<td>Application configuration file to be given to Hadoop</td>
</tr>
<tr>
<td>HadoopInstallFolder</td>
<td>Installation location of Hadoop on the local machine</td>
</tr>
<tr>
<td>HadoopProperties</td>
<td>Map of name-value property pairs to be given to Hadoop</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LicenseNumber</td>
<td>License number to use with MathWorks hosted licensing</td>
</tr>
<tr>
<td>RequiresMathWorksHostedLicensing</td>
<td>Specify whether cluster uses MathWorks hosted licensing</td>
</tr>
<tr>
<td>SparkInstallFolder</td>
<td>Installation location of Spark on the local machine</td>
</tr>
<tr>
<td>SparkProperties</td>
<td>Map of name-value property pairs to be given to Spark</td>
</tr>
</tbody>
</table>

HadoopProperties allows you to override configuration properties for Hadoop. See the list of properties in the Hadoop documentation.

The SparkInstallFolder is by default set to the SPARK_HOME environment variable. This is required for tall array evaluation on Hadoop (but not for mapreduce). For a correctly configured cluster, you only need to set the installation folder.

SparkProperties allows you to override configuration properties for Spark. See the list of properties in the Spark documentation.

**Help**

For further help, type:

```bash
help parallel.cluster.Hadoop
```

**Specify Memory Properties**

Spark enabled Hadoop clusters place limits on how much memory is available. You must adjust these limits to support your workflow.

**Size of Data to Gather**

The amount of data gathered to the client is limited by the Spark properties:

- spark.driver.memory
• `spark.executor.memory`

The amount of data to gather from a single Spark task must fit in these properties. A single Spark task processes one block of data from HDFS, which is 128 MB of data by default. If you gather a tall array containing most of the original data, you must ensure these properties are set to fit.

If these properties are set too small, you see an error like the following.

```
Error using tall/gather (line 50)
Out of memory; unable to gather a partition of size 300m from Spark.
Adjust the values of the Spark properties `spark.driver.memory` and `spark.executor.memory` to fit this partition.
```

The error message also specifies the property settings you need.

Adjust the properties either in the default settings of the cluster or directly in MATLAB. To adjust the properties in MATLAB, add name-value pairs to the `SparkProperties` property of the cluster. For example:

```
cluster = parallel.cluster.Hadoop;
cluster.SparkProperties('spark.driver.memory') = '2048m';
cluster.SparkProperties('spark.executor.memory') = '2048m';
mapreducer(cluster);
```

**Specify Working Memory Size for a MATLAB Worker**

The amount of working memory for a MATLAB Worker is limited by the Spark property:

• `spark.yarn.executor.memoryOverhead`

By default, this is set to 2.5 GB. You typically need to increase this if you use `arrayfun`, `cellfun`, or custom datastores to generate large amounts of data in one go. It is advisable to increase this if you come across lost or crashed Spark Executor processes.

You can adjust these properties either in the default settings of the cluster or directly in MATLAB. To adjust the properties in MATLAB, add name-value pairs to the `SparkProperties` property of the cluster. For example:

```
cluster = parallel.cluster.Hadoop;
cluster.SparkProperties('spark.yarn.executor.memoryOverhead') = '4096m';
mapreducer(cluster);
```
See Also

parallel.Cluster, parallel.Pool

See Also

Topics
  “Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
  “Run mapreduce on a Hadoop Cluster” on page 6-75

Introduced in R2014b
Future

Request function execution on parallel pool workers or MATLAB client

Introduced in R2013b
**parallel.Job**

Access job properties and behaviors

**Constructors**

createCommunicatingJob, createJob, findJob, recreate

getCurrentJob (in the workspace of the MATLAB worker)

**Container Hierarchy**

Parent parallel.Cluster

Children parallel.Task

**Description**

A parallel.Job object provides access to a job, which you create, define, and submit for execution.

**Types**

The following table describes the available types of job objects. The job type is determined by the type of cluster, and whether the tasks must communicate with each other during execution.

<table>
<thead>
<tr>
<th>Job Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel.job.MJSIndependentJob</td>
<td>Job of independent tasks on MJS cluster</td>
</tr>
<tr>
<td>parallel.job.MJSCommunicatingJob</td>
<td>Job of communicating tasks on MJS cluster</td>
</tr>
<tr>
<td>parallel.job.CJSIndependentJob</td>
<td>Job of independent tasks on CJS cluster</td>
</tr>
<tr>
<td>parallel.job.CJSCommunicatingJob</td>
<td>Job of communicating tasks on CJS cluster</td>
</tr>
</tbody>
</table>
Methods

All job type objects have the same methods, described in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>Cancel job or task</td>
</tr>
<tr>
<td>createTask</td>
<td>Create new task in job</td>
</tr>
<tr>
<td>delete</td>
<td>Remove job or task object from cluster and memory</td>
</tr>
<tr>
<td>diary</td>
<td>Display or save Command Window text of batch job</td>
</tr>
<tr>
<td>fetchOutputs (Job)</td>
<td>Retrieve output arguments from all tasks in job</td>
</tr>
<tr>
<td>findTask</td>
<td>Task objects belonging to job object</td>
</tr>
<tr>
<td>listAutoAttachedFiles</td>
<td>List of files automatically attached to job, task, or parallel pool</td>
</tr>
<tr>
<td>load</td>
<td>Load workspace variables from batch job</td>
</tr>
<tr>
<td>recreate</td>
<td>Create new job from existing job</td>
</tr>
<tr>
<td>submit</td>
<td>Queue job in scheduler</td>
</tr>
<tr>
<td>wait</td>
<td>Wait for job to change state</td>
</tr>
</tbody>
</table>

Properties

Common to All Job Types

The following properties are common to all job object types.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdditionalPaths</td>
<td>Folders to add to MATLAB search path of workers</td>
</tr>
<tr>
<td>AttachedFiles</td>
<td>Files and folders that are sent to workers</td>
</tr>
<tr>
<td>AutoAddClientPath</td>
<td>Specifies whether user-added-entries on the client’s path are automatically added to each worker’s path</td>
</tr>
<tr>
<td>AutoAttachFiles</td>
<td>Specifies if dependent code files are automatically sent to workers</td>
</tr>
<tr>
<td>CreateDateTime</td>
<td>Date and time when the job is created</td>
</tr>
</tbody>
</table>
## Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnvironmentVariables</td>
<td>Names of environment variables that are sent to the workers</td>
</tr>
<tr>
<td>FinishDateTime</td>
<td>Date and time when the job finishes running</td>
</tr>
<tr>
<td>ID</td>
<td>Job’s numeric identifier</td>
</tr>
<tr>
<td>JobData</td>
<td>Information made available to all workers for job’s tasks</td>
</tr>
<tr>
<td>Name</td>
<td>Name of job</td>
</tr>
<tr>
<td>Parent</td>
<td>Cluster object containing this job</td>
</tr>
<tr>
<td>StartDateTime</td>
<td>Date and time when the job starts running</td>
</tr>
<tr>
<td>State</td>
<td>State of job: 'pending', 'queued', 'running', 'finished', or 'failed'</td>
</tr>
<tr>
<td>SubmitDateTime</td>
<td>Date and time when the job is submitted to the queue</td>
</tr>
<tr>
<td>Tag</td>
<td>Label associated with job</td>
</tr>
<tr>
<td>Tasks</td>
<td>Array of task objects contained in job</td>
</tr>
<tr>
<td>Type</td>
<td>Job type: 'independent', 'pool', or 'spmd'</td>
</tr>
<tr>
<td>UserData</td>
<td>Information associated with job object</td>
</tr>
<tr>
<td>Username</td>
<td>Name of user who owns job</td>
</tr>
</tbody>
</table>

### MJS Jobs

MJS independent job objects and MJS communicating job objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuthorizedUsers</td>
<td>Users authorized to access job</td>
</tr>
<tr>
<td>FinishedFcn</td>
<td>Callback function executed on client when this job finishes</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>NumWorkersRange</td>
<td>Minimum and maximum limits for number of workers to run job</td>
</tr>
<tr>
<td>QueuedFcn</td>
<td>Callback function executed on client when this job is submitted to queue</td>
</tr>
<tr>
<td>RestartWorker</td>
<td>True if workers are restarted before evaluating first task for this job</td>
</tr>
<tr>
<td>RunningFcn</td>
<td>Callback function executed on client when this job starts running</td>
</tr>
<tr>
<td>Timeout</td>
<td>Time limit, in seconds, to complete job</td>
</tr>
</tbody>
</table>

**CJS Jobs**

CJS independent job objects do not have any properties beyond the properties common to all job types.

CJS communicating job objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumWorkersRange</td>
<td>Minimum and maximum limits for number of workers to run job</td>
</tr>
</tbody>
</table>

**Help**

To get further help on a particular type of parallel.Job object, including a list of links to help for its properties, type `help parallel.job.<job-type>`. For example:

```
help parallel.job.MJSIndependentJob
```

**See Also**

`parallel.Cluster`, `parallel.Task`, `parallel.Worker`
Introduced in R2012a
**parallel.Pool**

Access parallel pool

**Constructors**

parpool, gcp

**Description**

A parallel.Pool object provides access to a parallel pool running on a cluster.

**Methods**

A parallel pool object has the following methods.

- `addAttachedFiles`       Attach files or folders to parallel pool
- `delete (Pool)`         Shut down parallel pool
- `listAutoAttachedFiles`  List of files automatically attached to job, task, or parallel pool
- `parfeval`               Execute function asynchronously on parallel pool worker
- `parfevalOnAll`          Execute function asynchronously on all workers in parallel pool
- `ticBytes`               Start counting bytes transferred within parallel pool
- `tocBytes`               Read how many bytes have been transferred since calling `ticBytes`
- `updateAttachedFiles`    Update attached files or folders on parallel pool

**Properties**

A parallel pool object has the following properties.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttachedFiles</td>
<td>Files and folders that are sent to workers</td>
</tr>
<tr>
<td>Cluster</td>
<td>Cluster on which the parallel pool is running</td>
</tr>
<tr>
<td>Connected</td>
<td>False if the parallel pool has shut down</td>
</tr>
<tr>
<td>FevalQueue</td>
<td>Queue of FevalFutures to run on the parallel pool</td>
</tr>
<tr>
<td>IdleTimeout</td>
<td>Time duration in minutes before idle parallel pool will shut down</td>
</tr>
<tr>
<td>NumWorkers</td>
<td>Number of workers comprising the parallel pool</td>
</tr>
<tr>
<td>SpmdEnabled</td>
<td>Indication if pool can run SPMD code</td>
</tr>
</tbody>
</table>

**Help**

To get further help on parallel.Pool objects, including a list of links to help for specific properties, type:

help parallel.Pool

**See Also**

parallel.Cluster, parallel.Future

**See Also**

**Topics**

“Run Code on Parallel Pools” on page 2-67

**Introduced in R2013b**
DataQueue

Class that enables sending and listening for data between client and workers

Properties

QueueLength — Number of items currently held on the queue
zero or positive integer

Dependent property on the queue that indicates how many items of data are waiting to be removed from the queue.

q = parallel.pool.DataQueue;
% No messages in queue because nothing has been sent.
q.QueueLength

ans =

0

q.send('sending a message')
% Now QueueLength = 1 because one message has been sent.
q.QueueLength

ans =

1

% Add a callback to process the queue.
listener = q.afterEach(@disp);

sending a message

% Now QueueLength = 0 because there are no more pending messages.
q.QueueLength

ans =

0
% Data sent now is immediately processed by the callback so that QueueLength remains 0.
q.send('sending message 2')
q.QueueLength

sending message 2
ans =

0

% Deleting all callback listeners causes messages to build up in the queue again.
delete(listener)
q.send('sending message 3')
q.QueueLength

ans =

1

Examples

Send a Message in a parfor-Loop, and Dispatch the Message on the Queue

Construct a DataQueue, and call afterEach.

q = parallel.pool.DataQueue;
afterEach(q, @disp);

Start a parfor-loop, and send a message. The pending message is passed to the
afterEach function, in this example @disp.

parfor i = 1:3
    send(q, i);
end;
For more details on listening for data using a DataQueue, see `afterEach`.

**Construct a Simple `parfor` Wait Bar Using a Data Queue**

Create a DataQueue, and use `afterEach` to specify the function to execute each time the queue receives data. This example calls a subfunction that updates the wait bar.

Create a `parfor`-loop to carry out a computationally demanding task in MATLAB. Use `send` to send some dummy data on each iteration of the `parfor`-loop. When the queue receives the data, `afterEach` calls `nUpdateWaitbar` in the client MATLAB, and you can observe the wait bar progress.

```matlab
function a = parforWaitbar

D = parallel.pool.DataQueue;
h = waitbar(0, 'Please wait ...');
afterEach(D, @nUpdateWaitbar);

N = 200;
p = 1;

parfor i = 1:N
    a(i) = max(abs(eig(rand(400))));
    send(D, i);
end

function nUpdateWaitbar(~)
    waitbar(p/N, h);
    p = p + 1;
end
end
```

![Wait Bar Example](image-url)
Call afterEach to Dispatch Data on a Queue

If you call afterEach and there are items on the queue waiting to be dispatched, these items are immediately dispatched to the function handle specified by afterEach. Call afterEach before sending data to the queue, to ensure that on send, the function handle @disp is called.

Construct a DataQueue and call afterEach.

```matlab
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```

If you then send messages to the queue, each message is passed to the function handle specified by afterEach immediately.

```matlab
parfor i = 1:3
    send(q, i);
end
send(q, 0);
```

If you send the data to the queue and then call afterEach, each of the pending messages are passed to the function handle specified by afterEach.

```matlab
q = parallel.pool.DataQueue;
parfor i = 1:3
    send(q, i);
end
afterEach(q, @disp);
```
See Also
afterEach | gcp | labReceive | labSend | parallel.pool.PollableDataQueue | parfor | poll | send | spmd

Topics
Class Attributes (MATLAB)
Property Attributes (MATLAB)
“Plot During Parameter Sweep with parfor”

Introduced in R2017a
parallel.pool.PollableDataQueue

Class that enables sending and polling for data between client and workers

Description

parallel.pool.PollableDataQueue enables sending and polling for data or messages between workers and client in a parallel pool while a computation is being carried out. You can get intermediate values and progress of the computation.

To send data from a parallel pool worker back to the client, first construct a PollableDataQueue in the client. Pass this PollableDataQueue into a parfor-loop or other parallel language construct, such as parfeval. From the workers, call send to send data back to the client. At the client, use poll to retrieve the result of a message or data sent from a worker.

- You can call send from the process that calls the constructor, if required.
- You can construct the queue on the workers and send it back to the client to enable communication in the reverse direction. However, you cannot send a queue from one worker to another. Use spmd, labSend, or labReceive instead.
- Unlike all other handle objects, PollableDataQueue instances do remain connected when they are sent to workers.

Construction

`p = parallel.pool.PollableDataQueue`

The constructor for a PollableDataQueue takes no arguments and returns an object that can be used to send and poll for messages (or data) from different workers. You call the constructor only in the process where you want to receive the data. In the usual workflow, the workers should not be calling the constructor, but should be handed an existing PollableDataQueue instance instead.
Properties

QueueLength — Number of items currently held on the queue
zero or positive integer

Read-only property that indicates how many items of data are waiting to be removed from the queue.

Example:

```matlab
pollableQ = parallel.pool.PollableDataQueue;
% No messages in queue because nothing has been sent yet.
pollableQ.QueueLength

ans =
    0

pollableQ.send('A message')
% Now QueueLength = 1 because one message is sent and not yet polled.
pollableQ.QueueLength

ans =
    1

msg = pollableQ.poll();
disp(msg)
A message

pollableQ.QueueLength
% Now QueueLength = 0 because there are no more pending messages.
ans =
    0
```

Methods

A parallel.pool.PollableDataQueue object has the following methods.
Retrieve data sent from a worker

Send data from worker to client using a data queue

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Send a Message in a parfor-loop, and Poll for the Result

Construct a PollableDataQueue.

```matlab
p = parallel.pool.PollableDataQueue;
```

Start a parfor-loop, and send a message, such as data with the value 1.

```matlab
parfor i = 1
    send(p, i);
end
```

Poll for the result.

```matlab
poll(p)
```

1

For more details on polling for data using a PollableDataQueue, see poll.

See Also

gcp | labReceive | labSend | parallel.pool.DataQueue | parfeval | parfevalOnAll | parfor | poll | send

Topics

Class Attributes (MATLAB)
Property Attributes (MATLAB)

**Introduced in R2017a**
parallel.Task

Access task properties and behaviors

Constructors

createTask, findTask

gGetCurrentTask (in the workspace of the MATLAB worker)

Container Hierarchy

Parent       parallel.Job
Children     none

Description

A parallel.Task object provides access to a task, which executes on a worker as part of a job.

Types

The following table describes the available types of task objects, determined by the type of cluster.

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel.task.MJSTask</td>
<td>Task on MJS cluster</td>
</tr>
<tr>
<td>parallel.task.CJSTask</td>
<td>Task on CJS cluster</td>
</tr>
</tbody>
</table>
Methods

All task type objects have the same methods, described in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>Cancel job or task</td>
</tr>
<tr>
<td>delete</td>
<td>Remove job or task object from cluster and memory</td>
</tr>
<tr>
<td>listAutoAttachedFiles</td>
<td>List of files automatically attached to job, task, or parallel pool</td>
</tr>
<tr>
<td>wait</td>
<td>Wait for job to change state</td>
</tr>
</tbody>
</table>

Properties

Common to All Task Types

The following properties are common to all task object types.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaptureDiary</td>
<td>Specify whether to return diary output</td>
</tr>
<tr>
<td>CreateDateTime</td>
<td>Date and time when the task is created</td>
</tr>
<tr>
<td>Diary</td>
<td>Text produced by execution of task object’s function</td>
</tr>
<tr>
<td>Error</td>
<td>Task error information</td>
</tr>
<tr>
<td>ErrorIdentifier</td>
<td>Task error identifier</td>
</tr>
<tr>
<td>ErrorMessage</td>
<td>Message from task error</td>
</tr>
<tr>
<td>FinishDateTime</td>
<td>Date and time when the task is finished</td>
</tr>
<tr>
<td>Function</td>
<td>Function called when evaluating task</td>
</tr>
<tr>
<td>ID</td>
<td>Task’s numeric identifier</td>
</tr>
<tr>
<td>InputArguments</td>
<td>Input arguments to task function</td>
</tr>
<tr>
<td>Name</td>
<td>Name of this task</td>
</tr>
<tr>
<td>NumOutputArguments</td>
<td>Number of arguments returned by task function</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutputArguments</td>
<td>Output arguments from running task function on worker</td>
</tr>
<tr>
<td>Parent</td>
<td>Job object containing this task</td>
</tr>
<tr>
<td>StartDateTime</td>
<td>Date and time when the task is started</td>
</tr>
<tr>
<td>State</td>
<td>Current state of task</td>
</tr>
<tr>
<td>UserData</td>
<td>Information associated with this task object</td>
</tr>
<tr>
<td>Warnings</td>
<td>Warning information issued during execution of the task, captured in a struct array with the fields message, identifier, and stack</td>
</tr>
<tr>
<td>Worker</td>
<td>Object representing worker that ran this task</td>
</tr>
</tbody>
</table>

**MJS Tasks**

MJS task objects have the following properties in addition to the common properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FailureInfo</td>
<td>Information returned from failed task</td>
</tr>
<tr>
<td>FinishedFcn</td>
<td>Callback executed in client when task finishes</td>
</tr>
<tr>
<td>MaximumRetries</td>
<td>Maximum number of times to rerun failed task</td>
</tr>
<tr>
<td>NumFailures</td>
<td>Number of times tasked failed</td>
</tr>
<tr>
<td>RunningFcn</td>
<td>Callback executed in client when task starts running</td>
</tr>
<tr>
<td>Timeout</td>
<td>Time limit, in seconds, to complete task</td>
</tr>
</tbody>
</table>

**CJS Tasks**

CJS task objects have no properties beyond the properties common to all clusters.
Help

To get further help on either type of parallel.Task object, including a list of links to help for its properties, type:

help parallel.task.MJSTask
help parallel.task.CJSTask

See Also


Introduced in R2012a
**parallel.Worker**

Access worker that ran task

**Constructors**

getcCurrentWorker in the workspace of the MATLAB worker.

In the client workspace, a parallel.Worker object is available from the Worker property of a parallel.Task object.

**Container Hierarchy**

Parent: parallel.cluster.MJS

Children: none

**Description**

A parallel.Worker object provides access to the MATLAB worker session that executed a task as part of a job.

**Types**

<table>
<thead>
<tr>
<th>Worker Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel.cluster.MJSWorker</td>
<td>MATLAB worker on MJS cluster</td>
</tr>
<tr>
<td>parallel.cluster.CJSWorker</td>
<td>MATLAB worker on CJS cluster</td>
</tr>
</tbody>
</table>

**Methods**

There are no methods for a parallel.Worker object other than generic methods for any objects in the workspace, such as `delete`, etc.
Properties

MJS Worker

The following table describes the properties of an MJS worker.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllHostAddresses</td>
<td>IP addresses of worker host</td>
</tr>
<tr>
<td>Name</td>
<td>Name of worker, set when worker session started</td>
</tr>
<tr>
<td>Parent</td>
<td>MJS cluster to which this worker belongs</td>
</tr>
</tbody>
</table>

CJS Worker

The following table describes the properties of an CJS worker.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComputerType</td>
<td>Type of computer on which worker ran; the value of the MATLAB function computer executed on the worker</td>
</tr>
<tr>
<td>Host</td>
<td>Host name where worker executed task</td>
</tr>
<tr>
<td>ProcessId</td>
<td>Process identifier for worker</td>
</tr>
</tbody>
</table>

Help

To get further help on either type of parallel.Worker object, including a list of links to help for its properties, type:

help parallel.cluster.MJSWorker
help parallel.cluster.CJSWorker

See Also

Introduced in R2012a
RemoteClusterAccess

Connect to schedulers when client utilities are not available locally

Constructor

parallel.cluster.RemoteClusterAccess

r = parallel.cluster.RemoteClusterAccess(username)

r = parallel.cluster.RemoteClusterAccess(username,P1,V1,...,Pn,Vn)

Description

parallel.cluster.RemoteClusterAccess allows you to establish a connection and run commands on a remote host. This class is intended for use with the generic scheduler interface when using remote submission of jobs or on nonshared file systems.

r = parallel.cluster.RemoteClusterAccess(username) uses the supplied username when connecting to the remote host, and returns a RemoteClusterAccess object r. You will be prompted for a password when establishing the connection.

r = parallel.cluster.RemoteClusterAccess(username,P1,V1,...,Pn,Vn) allows additional parameter-value pairs that modify the behavior of the connection. The accepted parameters are:

- 'IdentityFilename' — A character vector containing the full path to the identity file to use when connecting to a remote host. If 'IdentityFilename' is not specified, you are prompted for a password when establishing the connection.
- 'IdentityFileHasPassphrase' — A logical indicating whether or not the identity file requires a passphrase. If true, you are prompted for a password when establishing a connection. If an identity file is not supplied, this property is ignored. This value is false by default.

For more information and detailed examples, see the integration scripts provided in matlabroot/toolbox/distcomp/examples/integration. For example, the scripts for PBS in a nonshared file system are in
### Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect</td>
<td><code>connect(r,clusterHost)</code> establishes a connection to the specified host using the user credential options supplied in the constructor. File mirroring is not supported.</td>
</tr>
<tr>
<td></td>
<td><code>connect(r,clusterHost,remoteDataLocation)</code> establishes a connection to the specified host using the user credential options supplied in the constructor. <code>remoteDataLocation</code> identifies a folder on the <code>clusterHost</code> that is used for file mirroring. The user credentials supplied in the constructor must have write access to this folder.</td>
</tr>
<tr>
<td>disconnect</td>
<td><code>disconnect(r)</code> disconnects the existing remote connection. The <code>connect</code> method must have already been called.</td>
</tr>
<tr>
<td>doLastMirrorForJob</td>
<td><code>doLastMirrorForJob(r,job)</code> performs a final copy of changed files from the remote <code>DataLocation</code> to the local <code>DataLocation</code> for the supplied job. Any running mirrors for the job also stop and the job files are removed from the remote <code>DataLocation</code>. The <code>startMirrorForJob</code> or <code>resumeMirrorForJob</code> method must have already been called.</td>
</tr>
<tr>
<td>getRemoteJobLocation</td>
<td><code>getRemoteJobLocation(r,jobID,remoteOS)</code> returns the full path to the remote job location for the supplied <code>jobID</code>. Valid values for <code>remoteOS</code> are 'pc' and 'unix'.</td>
</tr>
<tr>
<td>isJobUsingConnection</td>
<td><code>isJobUsingConnection(r,jobID)</code> returns true if the job is currently being mirrored.</td>
</tr>
<tr>
<td>Method Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>resumeMirrorForJob</td>
<td>resumeMirrorForJob(r, job) resumes the mirroring of files from the remote DataLocation to the local DataLocation for the supplied job. This is similar to the startMirrorForJob method, but does not first copy the files from the local DataLocation to the remote DataLocation. The connect method must have already been called. This is useful if the original client MATLAB session has ended, and you are accessing the same files from a new client session.</td>
</tr>
<tr>
<td>runCommand</td>
<td>[status, result] = runCommand(r, command) runs the supplied command on the remote host and returns the resulting status and standard output. The connect method must have already been called.</td>
</tr>
<tr>
<td>startMirrorForJob</td>
<td>startMirrorForJob(r, job) copies all the job files from the local DataLocation to the remote DataLocation, and starts mirroring files so that any changes to the files in the remote DataLocation are copied back to the local DataLocation. The connect method must have already been called.</td>
</tr>
<tr>
<td>stopMirrorForJob</td>
<td>stopMirrorForJob(r, job) immediately stops the mirroring of files from the remote DataLocation to the local DataLocation for the specified job. The startMirrorForJob or resumeMirrorForJob method must have already been called. This cancels the running mirror and removes the files for the job from the remote location. This is similar to doLastMirrorForJob, except that stopMirrorForJob makes no attempt to ensure that the local job files are up to date. For normal mirror stoppage, use doLastMirrorForJob.</td>
</tr>
<tr>
<td>connectedAccess</td>
<td>getConnectedAccess(host, username) returns a RemoteClusterAccess object that is connected to the supplied host. This function may return a previously constructed RemoteClusterAccess object if one exists.</td>
</tr>
<tr>
<td></td>
<td>getConnectedAccess(..., P1, V1, ...Pn, Vn) passes the additional parameters to the RemoteClusterAccess constructor.</td>
</tr>
</tbody>
</table>
Method Name | Description
--- | ---
getConnectedAccessWithMirror | getConnectedAccessWithMirror(host, location, username) returns a RemoteClusterAccess object that is connected to the supplied host, using the location as the mirror location. This function may return a previously constructed RemoteClusterAccess object if one exists.

getConnectedAccessWithMirror(..., P1,V1,...Pn,Vn) passes the additional parameters to the RemoteClusterAccess constructor.

Properties

A RemoteClusterAccess object has the following read-only properties. Their values are set when you construct the object or call its connect method.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hostname</td>
<td>Name of the remote host to access.</td>
</tr>
<tr>
<td>IdentityFileHasPassphrase</td>
<td>Indicates if the identity file requires a passphrase.</td>
</tr>
<tr>
<td>IdentityFilename</td>
<td>Full path to the identity file used when connecting to the remote host.</td>
</tr>
<tr>
<td>IsConnected</td>
<td>Indicates if there is an active connection to the remote host.</td>
</tr>
<tr>
<td>IsFileMirrorSupported</td>
<td>Indicates if file mirroring is supported for this connection. This is false if no remote DataLocation is supplied to the connect() method.</td>
</tr>
<tr>
<td>JobStorageLocation</td>
<td>Location on the remote host for files that are being mirrored.</td>
</tr>
<tr>
<td>UseIdentityFile</td>
<td>Indicates if an identity file should be used when connecting to the remote host.</td>
</tr>
<tr>
<td>Username</td>
<td>User name for connecting to the remote host.</td>
</tr>
</tbody>
</table>

Examples

Mirror files from the remote data location. Assume the object job represents a job on your generic scheduler.
remoteConnection = parallel.cluster.RemoteClusterAccess('testname');
connect(remoteConnection,'headnode1','/tmp/filemirror');
startMirrorForJob(remoteConnection,job);
submit(job)
% Wait for the job to finish
wait(job);

% Ensure that all the local files are up to date, and remove the
% remote files
doLastMirrorForJob(remoteConnection,job);

% Get the output arguments for the job
results = fetchOutputs(job)

For more detailed examples, see the integration scripts provided in `matlabroot/toolbox/distcomp/examples/integration`. For example, the scripts for PBS in a nonshared file system are in

`matlabroot/toolbox/distcomp/examples/integration/pbs/nonshared`

**See Also**

**Topics**

“Integration Scripts for Generic Schedulers” on page 7-21

**Introduced in R2011a**
Functions — Alphabetical List
addAttachedFiles

Attach files or folders to parallel pool

Syntax

addAttachedFiles(poolobj,files)

Description

addAttachedFiles(poolobj,files) adds extra attached files to the specified parallel pool. These files are transferred to each worker and are treated exactly the same as if they had been set at the time the pool was opened — specified by the parallel profile or the 'AttachedFiles' argument of the parpool function.

Examples

Add Attached Files to Current Parallel Pool

Add two attached files to the current parallel pool.

```
poolobj = gcp;
poolobj{AttachedFiles} = 'myFun1.m','myFun2.m';
addAttachedFiles(poolobj,{'myFun1.m','myFun2.m'})
```

Input Arguments

poolobj — Pool to which files attach

pool object

Pool to which files attach, specified as a pool object.

Example: poolobj = gcp;
files — Files or folders to attach
character vector | cell array

Files or folders to attach, specified as a character vector or cell array of character vectors. Each character vector can specify either an absolute or relative path to a file or folder.

Example: {'myFun1.m','myFun2.m'}

See Also
gcp | getAttachedFilesFolder | listAutoAttachedFiles | parpool | updateAttachedFiles

Topics
“Add and Modify Cluster Profiles” on page 6-20

Introduced in R2013b
afterAll

Specify a function to invoke after all parallel.Futures complete

Syntax

outputFuture = afterAll(futures,funtocall,nout)
outputFuture = afterAll(futures,funtocall,nout,'PassFuture',
passFuture)

Description

outputFuture = afterAll(futures,funtocall,nout) automatically evaluates
funtocall on the output arguments of all the futures in futures when they are all
complete, and returns outputFuture to hold the result. afterAll evaluates
funtocall on the vertical concatenation of the output arguments of all futures. If the
number of output arguments of the elements in futures differ, afterAll uses the
minimum and disregards the ending output arguments. afterAll invokes funtocall
with nout output arguments.

A useful application for afterAll is to update user interfaces such as plots and apps
during parallel computations using parfeval. For example, you can send several
computations to workers using parfeval and update your user interface when all of
them finish using afterAll.

outputFuture = afterAll(futures,funtocall,nout,'PassFuture',
passFuture) behaves the same if passFuture is false. If passFuture is true,
afterAll invokes funtocall on the array of futures futures and not on their output
arguments. This happens even if elements of futures encountered errors.

Examples
Call `afterAll` on `parfeval` Computations

You can use `afterAll` to automatically invoke functions on all of the combined outputs of your `parfeval` computations.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```matlab
for idx = 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

Display the maximum element among all of those vectors after they are created. `afterAll` executes the function handle on the combined output of all the futures when they all become ready.

```matlab
afterAll(f, @(r) disp(max(r)), 0);

0.9998
```

Combine `afterEach` and `afterAll`

You can combine `afterEach` and `afterAll` to automatically invoke more functions on the results of futures. Both `afterEach` and `afterAll` generate future variables that can be used again in `afterEach` and `afterAll`.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```matlab
for idx = 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

Starting parallel pool (parpool) using the 'local' profile ...
connected to 8 workers.

Compute the largest element in each of those vectors when they become ready. `afterEach` executes the function handle on the output of each future when they become ready and creates another future to hold the results.

```matlab
maxFuture = afterEach(f, @(r) max(r), 1);
```
To compute the minimum value among them, call `afterAll` on this new future. `afterAll` executes a function on the combined output arguments of all the futures after they all complete. In this case, `afterAll` executes the function `min` on the outputs of `maxFuture` after completing and creates another future to hold the result.

```matlab
minFuture = afterAll(maxFuture, @(r) min(r), 1);
```

You can fetch the result using `fetchOutputs`. `fetchOutput` waits until the future completes to gather the results.

```matlab
fetchOutputs(minFuture)
ans = 0.9973
```

You can check the result of `afterEach` by calling `fetchOutputs` on its future variable.

```matlab
fetchOutputs(maxFuture)
ans = 10x1
    0.9996
    0.9989
    0.9994
    0.9973
    1.0000
    1.0000
    0.9989
    0.9994
    0.9998
    0.9999
```

**Update a User Interface Asynchronously Using `afterEach` and `afterAll`**

You can perform asynchronous computations on workers using `parfeval` and leave the user interface responsive. Use `afterEach` to update the user interface when intermediate computations are ready. Use `afterAll` to update the user interface when all the computations are ready.

Create a simple user interface using a `waitbar`.

```matlab
h = waitbar(0, 'Waiting...');
```
Use `parfeval` to carry out time-consuming computations in the workers, for example, eigenvalues of random matrices. The computations happen asynchronously and the user interface updates during computation. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```matlab
for idx = 1:100
    f(idx) = parfeval(@(n) real(eig(randn(n))), 1, 5e2);
end
```

Compute the largest value in each of the computations when they become ready using `afterEach`. Update the proportion of finished futures in the waitbar when each of them completes using `afterEach`.

```matlab
maxFuture = afterEach(f, @max, 1);
updateWaitbarFuture = afterEach(f, @(~) waitbar(sum(strcmp('finished', {f.State}))/numel(f), h), 1);
```

Close the waitbar when all the computations are done. Use `afterAll` on `updateWaitbarFuture` to continue automatically with a close operation. `afterAll` obtains the figure handle from `updateWaitbarFuture` and executes its function on it.

```matlab
closeWaitbarFuture = afterAll(updateWaitbarFuture, @(h) delete(h), 0);
```

Show a histogram after all the maximum values are computed. Use `afterAll` on `maxFuture` to continue the operation automatically. `afterAll` obtains the maximum values from `maxFuture` and calls `histogram` on them.

```matlab
showsHistogramFuture = afterAll(maxFuture, @histogram, 0);
```

**Handle Errors in Future Variables**

When computations for future variables result in an error, by default, `afterAll` does not evaluate its function. If you want to handle any errors, for example, you have a user
interface that you want to update, you can use the name-value pair PassFuture. When set to true, the future variable is passed to the callback function. You can call `fetchOutputs` on it, process the outputs, and handle any possible errors.

Send computations to the workers using `parfeval`. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created. If your `parfeval` computations result in an error, the future variable errors, and its Error property reflects it.

```matlab
errorFuture = parfeval( @(n) randn(n), 0, 0.5);
```

Starting parallel pool (parpool) using the 'Local' profile ... connected to 4 workers.

```matlab
wait(errorFuture);
errorFuture.Error
```

```matlab
ans =
    ParallelException with properties:
        identifier: 'MATLAB:NonIntegerInput'
        message: 'Size inputs must be integers.'
        cause: {}
        remotecause: {[1x1 MException]}
        stack: [1x1 struct]
```

If you use `afterAll` on that future, the callback function is not evaluated. In the code below, the `msgbox` is not executed because the future errors.

```matlab
afterAll(errorFuture, @() msgbox('Operation completed'), 0);
```

To handle futures that result in an error, use the name-value pair PassFuture when calling `afterAll`. The future variable is passed to the callback function instead of its outputs. Call `fetchOutputs` on it, and process its outputs. If the future results in an error, `fetchOutputs` throws an error that you can catch and handle. For example, the following code shows an error dialog box.

```matlab
afterAll(errorFuture, @handleError, 0, 'PassFuture',true);
```
function handleError (f)
try
    output = fetchOutputs(f);
    % Do something with the output
catch
    errordlg('Operation failed');
end
end

**Input Arguments**

**futures** — Futures
parallel.Future

Futures, specified as an array of parallel.Future. `funtocall` is invoked on the combined outputs of all its elements when they all become ready. You can use `parfeval` to create futures.

If any element of `futures` encounters an error, `funtocall` is not invoked, and `outputFuture` completes with an error. To see if the future completes with an error, you can check the `Error` property of `outputFuture`. If you cancel an element of `futures`, this results in the same behavior as if the element encountered an error.

Example: `future = parfeval(@rand,1,1000,1); afterAll(future,@max,1);`

Data Types: `parallel.Future`

**funtocall** — Function to execute

function handle
Function to execute, specified as a function to call on the combined output arguments of all the futures in `futures` when they all become ready. `funtocall` is evaluated on the MATLAB client, not on the parallel pool workers.

Example: `funtocall = @max`

Data Types: `function handle`

**nout — Number of outputs**

integer

Number of outputs, specified as an integer, expected from `funtocall`.

Example: `afterAll(futures,@max,1)`

Data Types: `scalar`

**passFuture — Type of input arguments to `funtocall`**

logical scalar

Indicator, specified as a logical scalar that determines the type of input arguments to `funtocall`. If set to `true`, the future array `futures` is passed to `funtocall`. Otherwise, the output arguments of all futures in `futures` are passed to `funtocall`. This argument is optional and is `false` by default.

You can use this approach if you want to handle any errors. Set `passFuture` to `true` so that `afterAll` invokes `funtocall` on the outputs of `futures`, even if they encountered errors. You must call `fetchOutputs` on the input argument to `funtocall` to extract the results. If the future results in an error, `fetchOutputs` throws an error that you can catch and handle.

Example: `afterAll(futures,@(f) disp(fetchOutputs(f)), 0,'PassFuture',true)`

Data Types: `logical scalar`

**Output Arguments**

**outputFuture — Future**

`parallel.Future`
Future, returned as a `parallel.Future` to hold the results of evaluating `funtocall` on the combined output arguments of all the futures in `futures` when they all become ready.

To extract the results, call `fetchOutputs` on `outputFuture`:

```matlab
outputFutures = afterAll(futures, funtocall, nout);
[out1, out2, ..., outM] = fetchOutputs(outputFutures);
```

Note that this is equivalent to calling `funtocall` after fetching the outputs in `futures`, except that `afterAll` calls automatically `funtocall` when all elements in `futures` complete:

```matlab
[tmp1, tmp2, ..., tmpN] = fetchOutputs(futures);
[out1, out2, ..., outM] = funtocall(tmp1, tmp2, ..., tmpN)
```

`N` is the number of outputs from `futures` and `M` is the number of outputs specified in `afterAll` with the input argument `nout`.

**Tips**

- Use `afterAll` on any of the futures returned from `parfeval`, `parfevalOnAll`, `afterEach`, `afterAll`, or an array containing a combination of them. For example, use `afterAll` to automatically invoke more functions on the results of another `afterEach` or `afterAll`. You can invoke `afterAll` on futures before and after they finish.
- Use `cancel` on a future returned from `afterAll` to cancel its execution. If you invoke `afterAll` on a canceled future, this results in the same behavior as if the future encountered an error.

**See Also**

`afterEach` | `parallel.Future` | `parfeval`

**Introduced in R2018a**
**afterEach**

Define a function to call when new data is received on a DataQueue

**Syntax**

```matlab
listener = afterEach(queue, funtocall)
```

**Description**

`listener = afterEach(queue, funtocall)` specifies a function `funtocall` to execute each time the queue receives new data. You can specify multiple different functions to call, because each call to `afterEach` creates a new listener on the queue. If you want to specify another function, call `afterEach` again. To remove the registration of the function with the queue, delete the returned `listener` object.

You must call `afterEach` in the same process where you created the data queue, otherwise an error occurs. After calling `afterEach`, any current data in the queue is dispatched immediately to the supplied function.

**Examples**

**Call afterEach to Dispatch Data on a Queue**

If you call `afterEach` and there are items on the queue waiting to be dispatched, these items are immediately dispatched to the `afterEach` function. Call `afterEach` before sending data to the queue, to ensure that on `send`, the function handle specified by `afterEach` is called.

Construct a `DataQueue` and call `afterEach`.

```matlab
define a DataQueue
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```
If you then send messages to the queue, each message is passed to the function handle specified by `afterEach` immediately.

```matlab
parfor i = 1
    send(q, 2);
end

send(q, 3)
```

You can also first send various messages to the queue. When you call `afterEach`, the pending messages are passed to the `afterEach` function, in this example to the function handle `@disp`.

```matlab
q = parallel.pool.DataQueue;
parfor i = 1
    send(q, 2);
end
send(q, 3)

afterEach(q, @disp);
```

### Remove a Callback by Deleting the Listener

Construct a `DataQueue` and create a listener.

```matlab
D = parallel.pool.DataQueue;
listener = D.afterEach(@disp);
```

Send some data with the value 1.

```matlab
D.send(1)
```

Delete the listener.
delete(listener)
D.send(1)

No data is returned because you have removed the callback by deleting the listener.

**Input Arguments**

*queue*  —  Data queue
`parallel.pool.DataQueue`

Data queue, specified as a `parallel.pool.DataQueue` object.
Example: `q = parallel.pool.DataQueue;`

*funtocall*  —  Function handle
`function handle`

Function handle, specifying the function added to the list of functions to call when a piece of new data is received from *queue*.
Example: `listener = afterEach(queue, funtocall)`

All callback functions must accept *data* as single argument.

*afterEach(queue, @foo)* expects a function handle `@foo` to a function of the form

```
function foo(data)
end
```

When `send(queue, someData)` is called on the worker, *someData* is serialized and sent back to the client. *someData* is deserialized on the client and passed as the input to `foo(data)`.

**Output Arguments**

*listener*  —  listener
`event.listener`

Listener object created by `afterEach`, returned as the handle to an `event.listener` object.
See Also

event.listener | parallel.pool.DataQueue | parallel.pool.PollableDataQueue | parfor | poll | send

Introduced in R2017a
**afterEach**

Specify a function to invoke after each parallel.Future completes

**Syntax**

```matlab
outputFuture = afterEach(futures,funtocall,nout)
outputFuture = afterEach(futures,funtocall,nout,'PassFuture',
passFuture)
```

**Description**

`outputFuture = afterEach(futures,funtocall,nout)` automatically evaluates `funtocall` on the output arguments of each of the elements in `futures` as they become ready. `afterEach` calls `funtocall` with `nout` output arguments and produces `outputFuture` to hold the outputs.

A useful application for `afterEach` is to update user interfaces such as plots and apps during parallel computations using `parfeval`. For example, you can send several computations to workers using `parfeval` and update your user interface when each of those finishes using `afterEach`.

`outputFuture = afterEach(futures,funtocall,nout,'PassFuture',
passFuture)` behaves the same if `passFuture` is false. If `passFuture` is true, `afterEach` invokes `funtocall` on each element in `futures`, and not on their output arguments. This happens even if elements of `futures` encountered errors.

**Examples**

**Call afterEach on parfeval Computations**

You can use `afterEach` to automatically invoke functions on each of the results of `parfeval` computations.
Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```matlab
for idx = 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end
```

Display the maximum element in each of those vectors after they are created. `afterEach` executes the function handle on the output of each future when they become ready.

```matlab
afterEach(f, @(r) disp(max(r)), 0);
```

```
0.9975
0.9990
0.9982
0.9991
0.9982
0.9998
0.9999
0.9986
0.9996
0.9990
```

**Combine `afterEach` and `afterAll`**

You can combine `afterEach` and `afterAll` to automatically invoke more functions on the results of futures. Both `afterEach` and `afterAll` generate future variables that can be used again in `afterEach` and `afterAll`.

Use `parfeval` to compute random vectors in the workers. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.
for idx= 1:10
    f(idx) = parfeval(@rand, 1, 1000, 1);
end

Starting parallel pool (parpool) using the 'local' profile ... connected to 8 workers.

Compute the largest element in each of those vectors when they become ready. 
**afterEach** executes the function handle on the output of each future when they become ready and creates another future to hold the results.

maxFuture = afterEach(f, @(r) max(r), 1);

To compute the minimum value among them, call **afterAll** on this new future. 
**afterAll** executes a function on the combined output arguments of all the futures after they all complete. In this case, **afterAll** executes the function **min** on the outputs of **maxFuture** after completing and creates another future to hold the result.

minFuture = afterAll(maxFuture, @(r) min(r), 1);

You can fetch the result using **fetchOutputs**. **fetchOutput** waits until the future completes to gather the results.

fetchOutputs(minFuture)

ans = 0.9973

You can check the result of **afterEach** by calling **fetchOutputs** on its future variable.

fetchOutputs(maxFuture)

ans = 10×1

0.9996
0.9989
0.9994
0.9973
1.0000
1.0000
0.9989
0.9994
0.9998
0.9999
Update a User Interface Asynchronously Using afterEach and afterAll

You can perform asynchronous computations on workers using `parfeval` and leave the user interface responsive. Use `afterEach` to update the user interface when intermediate computations are ready. Use `afterAll` to update the user interface when all the computations are ready.

Create a simple user interface using a `waitbar`.

```matlab
h = waitbar(0, 'Waiting...');
```

Use `parfeval` to carry out time-consuming computations in the workers, for example, eigenvalues of random matrices. The computations happen asynchronously and the user interface updates during computation. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created.

```matlab
for idx = 1:100
    f(idx) = parfeval(@(n) real(eig(randn(n))), 1, 5e2);
end
```

Compute the largest value in each of the computations when they become ready using `afterEach`. Update the proportion of finished futures in the waitbar when each of them completes using `afterEach`.

```matlab
maxFuture = afterEach(f, @max, 1);
updateWaitbarFuture = afterEach(f, @(~) waitbar(sum(strcmp('finished', {f.State}))/numel(f), h), 1);
```

Close the waitbar when all the computations are done. Use `afterAll` on `updateWaitbarFuture` to continue automatically with a close operation. `afterAll` obtains the figure handle from `updateWaitbarFuture` and executes its function on it.

```matlab
closeWaitbarFuture = afterAll(updateWaitbarFuture, @(h) delete(h), 0);
```
Show a histogram after all the maximum values are computed. Use `afterAll` on `maxFuture` to continue the operation automatically. `afterAll` obtains the maximum values from `maxFuture` and calls `histogram` on them.

showsHistogramFuture = afterAll(maxFuture, @histogram, 0);

Handle Errors in Future Variables

When computations for future variables result in an error, by default, `afterEach` does not evaluate its function on the elements that failed. If you want to handle any errors, for example, you have a user interface that you want to update, you can use the name-value pair `PassFuture`. When set to `true`, the future variable is passed to the callback function. You can call `fetchOutputs` on it, process the outputs, and handle any possible errors.

Send computations to the workers using `parfeval`. With default preferences, `parfeval` creates a `parpool` automatically if there is not one already created. If your `parfeval` computations result in an error, the future variable errors, and its `Error` property reflects it.

errorFuture = parfeval( @(n) randn(n), 0, 0.5);
wait(errorFuture);
errorFuture.Error

ans =
ParallelException with properties:

    identifier: 'MATLAB:NonIntegerInput'
    message: 'Size inputs must be integers.'
    cause: {}
    remotecause: {[1x1 MException]}
    stack: [1x1 struct]

If you use `afterEach` on that future, the callback function is not evaluated on those elements in the future that errored. In the code below, the `msgbox` is not executed because the future errors.

afterEach(errorFuture, @() msgbox('Operation completed'), 0);

To handle futures that result in errors, use the name-value pair `PassFuture` when calling `afterEach`. The future variable is passed to the callback function instead of its outputs.
Call `fetchOutputs` on the future variable, and process its outputs. If the future results in an error, `fetchOutputs` throws an error that you can catch and handle. The following code shows an error dialog box.

```matlab
afterEach(errorFuture, @handleError, 0, 'PassFuture', true);
```

![Error Dialog](image)

```matlab
function handleError (f)
try
    output = fetchOutputs(f);
    % Do something with the output
catch
    errordlg('Operation failed');
end
end
```

**Input Arguments**

- **futures** — Futures
  `parallel.Future`

Futures, specified as an array of `parallel.Future`. The function `funtocall` is invoked on each of its elements when they become ready. You can use `parfeval` to create a future.

If an element of `futures` encounters an error, `funtocall` is not evaluated for that element of `futures`, however, it is evaluated for other elements of `futures` that do not encounter errors. To see if there are any futures with errors, you can check the `Error` property of `outputFuture`. This property is an empty cell array if there are no errors. If there are errors, it is a cell array that contains as many cells as futures in `futures`. A cell contains an error if the corresponding element of `futures` encountered an error, and is otherwise empty. If you cancel an element of `futures`, this results in the same behavior as if the element encountered an error.
Example: `future = parfeval(@rand,1,1000,1); afterEach(future,@max,1);`
Data Types: `parallel.Future`

**funtocall — Function to execute**

function handle

Function to execute, specified as a function to call on the output arguments of each of the futures in `futures` when they become ready. `funtocall` is evaluated on the MATLAB client and not on the parallel pool workers.

Example: `funtocall = @max`
Data Types: `function handle`

**nout — Number of outputs**

integer

Number of outputs, specified as an integer, expected from `funtocall`.

Example: `afterEach(futures,@max,1)`
Data Types: `scalar`

**passFuture — Type of input arguments to funtocall**

logical scalar

Indicator, specified as a logical scalar that determines the type of input arguments to `funtocall`. If set to `true`, each future in `futures` is passed to `funtocall`. Otherwise, the output arguments of each future in `futures` are passed to `funtocall`. This argument is optional and is `false` by default.

You can use this approach if you want to handle any errors. Set `passFuture` to `true` so that `afterEach` invokes `funtocall` on each element in `futures`, even if they encountered errors. You must call `fetchOutputs` on the input argument to `funtocall` to extract the results. If there are any futures with errors, `fetchOutputs` throws an error that you can catch and handle.

Example: `afterEach(futures,@(f) disp(fetchOutputs(f)), 0,'PassFuture',true)`
Data Types: `logical scalar`
Output Arguments

outputFuture — Future
parallel.Future

Future, returned as a `parallel.Future` to hold the results of evaluating `funtocall` on each of the futures in `futures` when they become ready. To extract the results, call `fetchOutputs` on `outputFuture`.

Tips

- Use `afterEach` on any of the futures returned from `parfeval`, `parfevalOnAll`, `afterAll`, `afterEach`, or an array containing a combination of them. For example, use `afterEach` to automatically invoke more functions on the results of another `afterAll` or `afterEach`. You can invoke `afterEach` on futures before and after they finish.
- Use `cancel` on a future returned from `afterEach` to cancel its execution. If you invoke `afterEach` on a canceled future, `afterEach` behaves the same way as if the future had an error.

See Also

`afterAll` | `parallel.Future` | `parfeval`

Introduced in R2018a
arrayfun

Apply function to each element of array on GPU

Syntax

\[
A = \text{arrayfun}(\text{FUN}, B)
\]
\[
A = \text{arrayfun}(\text{FUN}, B, C, \ldots)
\]
\[
[A, B, \ldots] = \text{arrayfun}(\text{FUN}, C, \ldots)
\]

Description

This method of a gpuArray object is very similar in behavior to the MATLAB function \text{arrayfun}, except that the actual evaluation of the function happens on the GPU, not on the CPU. Thus, any required data not already on the GPU is moved to GPU memory, the MATLAB function passed in for evaluation is compiled for the GPU, and then executed on the GPU. All the output arguments return as gpuArray objects, whose data you can retrieve with the \text{gather} method.

\[
A = \text{arrayfun}(\text{FUN}, B)
\]

applies the function specified by FUN to each element of the gpuArray B, and returns the results in gpuArray A. A is the same size as B, and \(A(i,j,\ldots)\) is equal to \(\text{FUN}(B(i,j,\ldots))\). FUN is a function handle to a function that takes one input argument and returns a scalar value. FUN must return values of the same class each time it is called. The input data must be an array of one of the following types: numeric, logical, or gpuArray. The order in which \text{arrayfun} computes elements of A is not specified and should not be relied on.

FUN must be a handle to a function that is written in the MATLAB language (i.e., not a MEX-function).

For more detailed information, see “Run Element-wise MATLAB Code on GPU” on page 9-18. For the subset of the MATLAB language that is currently supported by \text{arrayfun} on the GPU, see “Supported MATLAB Code” on page 9-19.

\[
A = \text{arrayfun}(\text{FUN}, B, C, \ldots)
\]

evaluates FUN using elements of arrays B, C, ... as input arguments with singleton expansion enabled. The resulting gpuArray element \(A(i,j,\ldots)\) is equal to \(\text{FUN}(B(i,j,\ldots), C(i,j,\ldots), \ldots)\). The inputs B, C, ... must
all have the same size or be scalar. Any scalar inputs are scalar expanded before being input to the function FUN.

One or more of the inputs B, C, ... must be a gpuArray; any of the others can reside in CPU memory. Each array that is held in CPU memory is converted to a gpuArray before calling the function on the GPU. If you plan to use an array in several different `arrayfun` calls, it is more efficient to convert that array to a gpuArray before making the series of calls to `arrayfun`.

```
[A,B,...] = arrayfun(FUN,C,...),
```

where `FUN` is a function handle to a function that returns multiple outputs, returns gpuArrays A, B, ..., each corresponding to one of the output arguments of `FUN`. `arrayfun` calls `FUN` each time with as many outputs as there are in the call to `arrayfun`. `FUN` can return output arguments having different classes, but the class of each output must be the same each time `FUN` is called. This means that all elements of A must be the same class; B can be a different class from A, but all elements of B must be of the same class, etc.

Although the MATLAB `arrayfun` function allows you to specify optional parameter name/value pairs, the `gpuArray arrayfun` method does not support these options.

**Tips**

- The first time you call `arrayfun` to run a particular function on the GPU, there is some overhead time to set up the function for GPU execution. Subsequent calls of `arrayfun` with the same function can run significantly faster.

- Nonsingleton dimensions of input arrays must match each other. In other words, the corresponding dimensions of arguments B, C, etc., must be equal to each other, or equal to one. Whenever a dimension of an input array is singleton (equal to 1), `arrayfun` uses singleton expansion to virtually replicate the array along that dimension to match the largest of the other arrays in that dimension. In the case where a dimension of an input array is singleton and the corresponding dimension in another argument array is zero, `arrayfun` virtually diminishes the singleton dimension to 0.

The size of the output array A is such that each dimension is the largest of the input arrays in that dimension for nonzero size, or zero otherwise. Notice in the following code how dimensions of size 1 are scaled up or down to match the size of the corresponding dimension in the other argument:

```matlab
R1 = rand(2,5,4,'gpuArray');
R2 = rand(2,1,4,3,'gpuArray');
```
R3 = rand(1,5,4,3,'gpuArray');
R = arrayfun(@(x,y,z)(x+y.*z),R1,R2,R3);
size(R)

2     5     4     3

R1 = rand(2,2,0,4,'gpuArray');
R2 = rand(2,1,1,4,'gpuArray');
R = arrayfun(@plus,R1,R2);
size(R)

2     2     0     4

• Because the operations supported by arrayfun are strictly element-wise, and each element’s computation is performed independently of the others, certain restrictions are imposed:
  • Input and output arrays cannot change shape or size.
  • Functions like rand do not support size specifications. Arrays of random numbers have independent streams for each element.

For more limitations and details, see “Tips and Restrictions” on page 9-22.

Examples

If you define a MATLAB function as follows:

```matlab
function [o1,o2] = aGpuFunction(a,b,c)
o1 = a + b;
o2 = o1 .* c + 2;
```

You can evaluate this on the GPU.

```matlab
s1 = gpuArray(rand(400));
s2 = gpuArray(rand(400));
s3 = gpuArray(rand(400));
[o1,o2] = arrayfun(@aGpuFunction,s1,s2,s3);
whos
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>400x400</td>
<td>108</td>
<td>gpuArray</td>
</tr>
<tr>
<td>o2</td>
<td>400x400</td>
<td>108</td>
<td>gpuArray</td>
</tr>
</tbody>
</table>
Use `gather` to retrieve the data from the GPU to the MATLAB workspace.

d = gather(o2);

**See Also**
bsxfun | gather | gpuArray | pagefun

*Introduced in R2010b*
batch

Run MATLAB script or function on worker

Syntax

\[
j = \text{batch('aScript')}
\]
\[
j = \text{batch(myCluster,'aScript')}
\]
\[
j = \text{batch(fcn,N,{x1, ..., xn})}
\]
\[
j = \text{batch(myCluster,fcn,N,{x1,...,xn})}
\]
\[
j = \text{batch(...,'p1',v1,'p2',v2,...)}
\]

Arguments

- \textit{j} 
  - The batch job object.
- \textit{'aScript'} 
  - The script of MATLAB code to be evaluated by the worker.
- \textit{myCluster} 
  - Cluster object representing cluster compute resources.
- \textit{fcn} 
  - Function handle or function name to be evaluated by the worker.
- \textit{N} 
  - The number of output arguments from the evaluated function.
- \textit{{x1, ..., xn}} 
  - Cell array of input arguments to the function.
- \textit{p1, p2} 
  - Object properties or other arguments to control job behavior.
- \textit{v1, v2} 
  - Initial values for corresponding object properties or arguments.

Description

\texttt{j = batch('aScript')} runs the script code of the file \texttt{aScript.m} on a worker in the cluster specified by the default cluster profile. (Note: Do not include the .m file extension with the script name argument.) The function returns \texttt{j}, a handle to the job object that runs the script. The script file \texttt{aScript.m} is copied to the worker.

\texttt{j = batch(myCluster,'aScript')} is identical to \texttt{batch('aScript')} except that the script runs on a worker according to the cluster identified by the cluster object \texttt{myCluster}. 
j = batch(fcn,N,{x1, ..., xn}) runs the function specified by a function handle or function name, fcn, on a worker in the cluster identified by the default cluster profile. The function returns j, a handle to the job object that runs the function. The function is evaluated with the given arguments, x1, ..., xn, returning N output arguments. The function file for fcn is copied to the worker. (Do not include the .m file extension with the function name argument.)

j = batch(myCluster,fcn,N,{x1,...,xn}) is identical to batch(fcn,N,{x1,...,xn}) except that the function runs on a worker in the cluster identified by the cluster object myCluster.

j = batch(...,'p1',v1,'p2',v2,...) allows additional parameter-value pairs that modify the behavior of the job. These parameters support batch for functions and scripts, unless otherwise indicated. The supported parameters are:

- **'Workspace'** — A 1-by-1 struct to define the workspace on the worker just before the script is called. The field names of the struct define the names of the variables, and the field values are assigned to the workspace variables. By default this parameter has a field for every variable in the current workspace where batch is executed. This parameter supports only the running of scripts.

- **'Profile'** — The name of a cluster profile used to identify the cluster. If this option is omitted, the default profile is used to identify the cluster and is applied to the job and task properties.

- **'AdditionalPaths'** — A character vector or cell array of character vectors that defines paths to be added to the MATLAB search path of the workers before the script or function executes. The default search path might not be the same on the workers as it is on the client; the path difference could be the result of different current working folders (pwd), platforms, or network file system access. The 'AdditionalPaths' property can assure that workers are looking in the correct locations for necessary code files, data files, model files, etc.

- **'AttachedFiles'** — A character vector or cell array of character vectors. Each character vector in the list identifies either a file or a folder, which gets transferred to the worker.

- **'AutoAddClientPath'** — A logical value (true or false) that controls whether user-added-entries on the client's path are added to each worker's path. The default is true.

- **'AutoAttachFiles'** — A logical value to specify whether code files should be automatically attached to the job. If true, the batch script or function is analyzed and
the code files that it depends on are automatically transferred to the worker. The default is true.

- 'CurrentFolder' — A character vector indicating in what folder the script executes. There is no guarantee that this folder exists on the worker. The default value for this property is the cwd of MATLAB when the batch command is executed. If the argument is '.', there is no change in folder before batch execution.

- 'CaptureDiary' — A logical flag to indicate that the toolbox should collect the diary from the function call. See the diary function for information about the collected data. The default is true.

- 'EnvironmentVariables'— A character vector, or cell array of character vectors that specifies the names of environment variables which are copied from the client session to the workers. The names specified here are appended to the 'EnvironmentVariables' property specified in the applicable parallel profile to form the complete list of environment variables. Any variables listed which are not set are not copied to the workers. These environment variables will be set on the workers for the duration of the batch job.

- 'Pool' — An integer specifying the number of workers to make into a parallel pool for the job in addition to the worker running the batch job itself. The script or function uses this pool for execution of statements such as parfor and spmd that are inside the batch code. Because the pool requires N workers in addition to the worker running the batch, there must be at least N+1 workers available on the cluster. You do not need a parallel pool already running to execute batch; and the new pool that batch creates is not related to a pool you might already have open. (See “Run a Batch Job with a Parallel Pool” on page 1-12.) The default value is 0, which causes the script or function to run on only a single worker without a parallel pool.

Examples

Run a batch script on a worker, without using a parallel pool:

```matlab
j = batch('script1');
```

Run a batch script that requires two additional files for execution:

```matlab
j = batch('myScript','AttachedFiles',{mscr1.m','mscr2.m'});
wait(j);
load(j);
```
Run a batch pool job on a remote cluster, using eight workers for the parallel pool in addition to the worker running the batch script. Capture the diary, and load the results of the job into the workspace. This job requires a total of nine workers:

```matlab
j = batch('script1','Pool',8,'CaptureDiary',true);
wait(j);  \% Wait for the job to finish
diary(j)  \% Display the diary
load(j)   \% Load job workspace data into client workspace
```

Run a batch pool job on a local worker, which employs two other local workers for the pool. Note, this requires a total of three workers in addition to the client, all on the local machine:

```matlab
j = batch('script1','Profile','local','Pool',2);
```

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

```matlab
delete(j)
```

Run a batch function on a cluster that generates a 10-by-10 random matrix:

```matlab
c = parcluster();
j = batch(c,@rand,1,{10,10});
wait(j)  \% Wait for the job to finish
diary(j) \% Display the diary
```

```matlab
r = fetchOutputs(j); \% Get results into a cell array
r{1}                \% Display result
```

**Tips**

To see your batch job’s status or to track its progress, use the Job Monitor, as described in “Job Monitor” on page 6-32. You can also use the Job Monitor to retrieve a job object for a batch job that was created in a different session, or for a batch job that was created without returning a job object from the `batch` call.

As a matter of good programming practice, when you no longer need it, you should delete the job created by the batch function so that it does not continue to consume cluster storage resources.
See Also
delete|diary|findJob|load|wait

Introduced in R2008a
bsxfun

Binary singleton expansion function for gpuArray

Syntax

C = bsxfun(FUN,A,B)

Description

bsxfun with gpuArray input is similar in behavior to the MATLAB function bsxfun, except that the actual evaluation of the function, FUN, happens on the GPU, not on the CPU.

C = bsxfun(FUN,A,B) applies the element-by-element binary operation specified by the function handle FUN to arrays A and B, with singleton expansion enabled. If A or B is a gpuArray, bsxfun moves all other required data to the GPU and performs its calculation on the GPU. The output array C is a gpuArray, which you can copy to the MATLAB workspace with gather.

For more detailed information, see “Run Element-wise MATLAB Code on GPU” on page 9-18. For the subset of the MATLAB language that is currently supported by bsxfun on the GPU, see “Supported MATLAB Code” on page 9-19.

The corresponding dimensions of A and B must be equal to each other, or equal to one. Whenever a dimension of A or B is singleton (equal to 1), bsxfun virtually replicates the array along that dimension to match the other array. In the case where a dimension of A or B is singleton and the corresponding dimension in the other array is zero, bsxfun virtually diminishes the singleton dimension to 0.

The size of the output array C is such that each dimension is the larger of the two input arrays in that dimension for nonzero size, or zero otherwise. Notice in the following code how dimensions of size 1 are scaled up or down to match the size of the corresponding dimension in the other argument:

R1 = rand(2,5,4,'gpuArray');
R2 = rand(2,1,4,3,'gpuArray');
R = bsxfun(@plus,R1,R2);
size(R)

    2   5   4   3

R1 = rand(2,2,0,4,'gpuArray');
R2 = rand(2,1,1,4,'gpuArray');
R = bsxfun(@plus,R1,R2);
size(R)

    2   2   0   4

**Examples**

Subtract the mean of each column from all elements in that column:

A = rand(8,'gpuArray');
M = bsxfun(@minus,A,mean(A));

**See Also**

arrayfun | gather | gpuArray | pagefun

**Introduced in R2012a**
cancel

Cancel job or task

Syntax

cancel(t)
cancel(j)

Arguments

t      Pending or running task to cancel.
j      Pending, running, or queued job to cancel.

Description

cancel(t) stops the task object, t, that is currently in the pending or running state. The task’s State property is set to finished, and no output arguments are returned. An error message stating that the task was canceled is placed in the task object’s ErrorMessage property, and the worker session running the task is restarted.

cancel(j) stops the job object, j, that is pending, queued, or running. The job’s State property is set to finished, and a cancel is executed on all tasks in the job that are not in the finished state. A job object that has been canceled cannot be started again.

If the job is running from an MJS, any worker sessions that are evaluating tasks belonging to the job object are restarted.

If the specified job or task is already in the finished state, no action is taken.

Examples

Cancel a task. Note afterward the task’s State, ErrorIdentifier, and ErrorMessage properties.
c = parcluster();
jobj1 = createJob(c);
t = createTask(jobj1, @rand, 1, {3,3});
cancel(t)

Task with properties:

ID: 1
State: finished
Function: @rand
Parent: Job 1
Start Time:
Running Duration: 0 days 0h 0m 0s

Error Identifier: parallel:task:UserCancellation
ErrorMessage: The task was cancelled by user "mylogin" on machine "myhost.mydomain.com".

See Also
delete|submit

Introduced before R2006a
cancel

Cancel queued or running future

Syntax

cancel(F)

Description

cancel(F) stops the queued and running futures contained in F. No action is taken for finished futures. Each element of F that is not already in state 'finished' has its State property set to 'finished', and its Error property is set to contain an MException indicating that execution was cancelled.

Examples

Run a function several times until a satisfactory result is found. In this case, the array of futures F is cancelled when a result is greater than 0.95.

```matlab
N = 100;
for idx = N:-1:1
    F(idx) = parfeval(@rand,1); % Create a random scalar
end
result = NaN; % No result yet.
for idx = 1:N
    [~, thisResult] = fetchNext(F);
    if thisResult > 0.95
        result = thisResult;
        % Have all the results needed, so break
        break;
    end
end
% With required result, cancel any remaining futures
cancel(F)
result
```
See Also
afterAll | afterEach | fetchNext | fetchOutputs | isequal | parfeval |
parfevalOnAll

Introduced in R2013b
changePassword

Prompt user to change MJS password

Syntax

changePassword(mjs)
changePassword(mjs,username)

Arguments

mjs MJS cluster object on which password is changing
username Character vector identifying the user whose password is changing

Description

changePassword(mjs) prompts you to change your password as the current user on the MATLAB job scheduler (MJS) cluster represented by cluster object mjs. (Use the parcluster function to create a cluster object.) In the dialog box that opens, you must enter your current password as well as the new password.

changePassword(mjs,username) prompts you as the MJS cluster admin user to change the password for another specified user. In the dialog box that opens, you must enter the cluster admin password in addition to the user’s new password. This allows the cluster admin to reset a password for a user who is not available or has forgotten the password. (Note: The cluster admin account was created when the MJS cluster was started with a security level of 1, 2, or 3.)
Examples

Change Password with a Cluster Profile

Change your password for the MJS cluster identified by an MJS cluster profile called MyMjsProfile.

mjs = parcluster('MyMjsProfile');
changePassword(mjs)

Change Password on a Parallel Pool

Change your password for the MJS cluster on which the parallel pool is running.

p = gcp;
mjs = p.Cluster;
changePassword(mjs)

Change Password of Another User

Change the password for another user named john. The admin user can access the MJS from a different session of MATLAB to do this, or change the Username property of the cluster object within john’s MATLAB client session.

mjs = parcluster('MyMjsProfile');
mjs.Username = 'admin'     % Generates prompt for admin user password.
changePassword(mjs,'john') % Generates prompt for both passwords.
At this point, the admin user might want to set the session user back to john.

```matlab
mjs.Username = 'john'  % Prompted again for password.
```

**See Also**

logout | parcluster | startjobmanager

**Topics**

“Set MJS Cluster Security” (MATLAB Distributed Computing Server)

**Introduced in R2010b**
classUnderlying

Class of elements within gpuArray or distributed array

Syntax

C = classUnderlying(D)

Description

C = classUnderlying(D) returns the name of the class of the elements contained within the gpuArray or distributed array D. Similar to the MATLAB class function, this returns a character vector indicating the class of the data.

Examples

Examine the class of the elements of a gpuArray.

N  = 1000;
G8 = ones(1,N,'uint8','gpuArray');
G1 = NaN(1,N,'single','gpuArray');
c8 = classUnderlying(G8)
c1 = classUnderlying(G1)

c8 =
    uint8

c1 =
    single

Examine the class of the elements of a distributed array.

N  = 1000;
D8 = ones(1,N,'uint8','distributed');
D1 = NaN(1,N,'single','distributed');
c8 = classUnderlying(D8)
c1 = classUnderlying(D1)

c8 =
uint8

c1 =
single

See Also
codistributed | distributed | gpuArray

Introduced in R2013b
clear

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.

Note Use parfevalOnAll instead of parfor or spmd if you want to use clear. This preserves workspace “Ensure Transparency in parfor-Loops” on page 2-58.

Examples

This example creates two job objects on the MATLAB job scheduler 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 MJS is shown to be identical to job1copy, while the second job is not.

c = parcluster();
delete(c.Jobs) % Assure there are no jobs
job1 = createJob(c);
job2 = createJob(c);
job1copy = job1;
clear job1 job2;
j1 = c.Jobs(1);
j2 = c.Jobs(2);
isequal (job1copy,j1)

ans =
   1

isequal (job1copy,j2)

ans =
   0

**Tips**

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

**See Also**

createJob | createTask | findJob | findTask | parcluster

*Introduced before R2006a*
codistributed

Create codistributed array from replicated local data

Syntax

C = codistributed(X)
C = codistributed(X,codist)
C = codistributed(X,lab,codist)
C = codistributed(C1,codist)

Description

C = codistributed(X) distributes a replicated array X using the default codistributor, creating a codistributed array C as a result. X must be a replicated array, that is, it must have the same value on all workers. size(C) is the same as size(X).

C = codistributed(X,codist) distributes a replicated array X using the distribution scheme defined by codistributor codist. X must be a replicated array, namely it must have the same value on all workers. size(C) is the same as size(X). For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

C = codistributed(X,lab,codist) distributes a local array X that resides on the worker identified by lab, using the codistributor codist. Local array X must be defined on all workers, but only the value from lab is used to construct C. size(C) is the same as size(X).

C = codistributed(C1,codist) accepts an array C1 that is already codistributed, and redistributes it into C according to the distribution scheme defined by the codistributor codist. This is the same as calling C = redistribute(C1,codist). If the existing distribution scheme for C1 is the same as that specified in codist, then the result C is the same as the input C1.
Examples

Create a 1000-by-1000 codistributed array C1 using the default distribution scheme.

```matlab
spmd
    N = 1000;
    X = magic(N); % Replicated on every worker
    C1 = codistributed(X); % Partitioned among the workers
end
```

Create a 1000-by-1000 codistributed array C2, distributed by rows (over its first dimension).

```matlab
spmd
    N = 1000;
    X = magic(N);
    C2 = codistributed(X,codistributor1d(1));
end
```

Tips

gather essentially performs the inverse of codistributed.

See Also

“What Is a Datastore?” (MATLAB) | codistributor1d | codistributor2dbc | distributed | gather | getLocalPart | globalIndices | redistribute | size | subsasgn | subsref
codistributed.build

Create codistributed array from distributed data

Syntax

D = codistributed.build(L,codist)
D = codistributed.build(L,codist,'noCommunication')

Description

D = codistributed.build(L,codist) forms a codistributed array with
getLocalPart(D) = L. The codistributed array D is created as if you had combined all
copies of the local array L. The distribution scheme is specified by codist. Global error
checking ensures that the local parts conform with the specified distribution scheme. For
information on constructing codistributor objects, see the reference pages for
codistributor1d and codistributor2dbc.

D = codistributed.build(L,codist,'noCommunication') builds a codistributed
array, without performing any interworker communications for error checking.

codist must be complete, which you can check by calling codist.isComplete(). The
requirements on the size and structure of the local part L depend on the class of codist.
For the 1-D and 2-D block-cyclic codistributors, L must have the same class and sparsity
on all workers. Furthermore, the local part L must represent the region described by the
globalIndices method on codist.

Examples

Create a codistributed array of size 1001-by-1001 such that column ii contains the value
ii.

spmd
    N = 1001;
    globalSize = [N,N];
    % Distribute the matrix over the second dimension (columns),
end
and let the codistributor derive the partition from the global size.
codistr = codistributor1d(2, ...
    codistributor1d.unsetPartition,globalSize)

On 4 workers, codistr.Partition equals [251,250,250,250].
Allocate storage for the local part.
localSize = [N, codistr.Partition(labindex)];
L = zeros(localSize);

Use globalIndices to map the indices of the columns of the local part into the global column indices.
globalInd = codistr.globalIndices(2);
On 4 workers, globalInd has the values:
1:251 on worker 1
252:501 on worker 2
502:751 on worker 3
752:1001 on worker 4

Initialize the columns of the local part to the correct value.
for localCol = 1:length(globalInd)
    globalCol = globalInd(localCol);
    L(:,localCol) = globalCol;
end
D = codistributed.build(L,codistr)
end

See Also
codistributor1d | codistributor2dbc | gather | getLocalPart | globalIndices | redistribute | size | subsasgn | subsref

Introduced in R2009b
codistributed.cell

Create codistributed cell array

Syntax

C = codistributed.cell(n)
C = codistributed.cell(m,n,p,...)
C = codistributed.cell([m,n,p,...])
C = cell(n,codist)
C = cell(m,n,p,...,codist)
C = cell([m,n,p,...],codist)

Description

C = codistributed.cell(n) creates an n-by-n codistributed array of underlying class cell, distributing along columns.

C = codistributed.cell(m,n,p,...) or C = codistributed.cell([m,n,p,...]) creates an m-by-n-by-p-by-... codistributed array of underlying class cell, using a default scheme of distributing along the last nonsingleton dimension.

Optional arguments to codistributed.cell must be specified after the required arguments, and in the following order:

- codist — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.
- 'noCommunication' — Specifies that no communication is to be performed when constructing the array, skipping some error checking steps.

C = cell(n,codist) is the same as C = codistributed.cell(n, codist). You can also use the 'noCommunication' object with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:
spmd
    C = cell(8,codistributor1d());
end

C = cell(m,n,p,...,codist) and C = cell([m,n,p,...],codist) are the same
as C = codistributed.cell(m,n,p,...) and C =
codistributed.cell([m,n,p,...]), respectively. You can also use the optional
'noCommunication' argument with this syntax.

Examples

With four workers,

spmd(4)
    C = codistributed.cell(1000);
end

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

spmd(4)
    codist = codistributor1d(2, 1:numlabs);
    C = cell(10, 10, codist);
end

creates a 10-by-10 codistributed cell array C, distributed by its columns. Each worker
contains a 10-by-labindex local piece of C.

See Also
cell|distributed.cell

Introduced in R2009b
codistributed.colon

Distributed colon operation

Syntax

codistributed.colon(a,d,b)
codistributed.colon(a,b)
codistributed.colon(___,codist)
codistributed.colon(___,codist,'noCommunication')
codistributed.colon(___,codist,'noCommunication')

Description

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

codistributed.colon(a,b) uses d = 1.

Optional arguments to codistributed.colon must be specified after the required arguments, and in the following order:

codistributed.colon(___,codist) uses the codistributor object codist to specify the distribution scheme of the resulting vector. If omitted, the result is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.

codistributed.colon(___,'noCommunication') or codistributed.colon(___,codist,'noCommunication') specifies that no communication is to be performed when constructing the vector, skipping some error checking steps.

Examples

Partition the vector 1:10 into four subvectors among four workers.
spmd(4); C = codistributed.colon(1,10), end

Lab 1:
    This worker stores C(1:3).
    LocalPart: [1 2 3]
    Codistributor: [1x1 codistributor1d]

Lab 2:
    This worker stores C(4:6).
    LocalPart: [4 5 6]
    Codistributor: [1x1 codistributor1d]

Lab 3:
    This worker stores C(7:8).
    LocalPart: [7 8]
    Codistributor: [1x1 codistributor1d]

Lab 4:
    This worker stores C(9:10).
    LocalPart: [9 10]
    Codistributor: [1x1 codistributor1d]

See Also

codistributor1d | codistributor2dbc | colon | for

Introduced in R2009b
codistributed.spalloc

Allocate space for sparse codistributed matrix

Syntax

SD = codistributed.spalloc(M,N,nzmax)
SD = spalloc(M,N,nzmax,codist)

Description

SD = codistributed.spalloc(M,N,nzmax) creates an M-by-N all-zero sparse codistributed matrix with room to hold nzmax nonzeros.

Optional arguments to codistributed.spalloc must be specified after the required arguments, and in the following order:

• codist — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. The allocated space for nonzero elements is consistent with the distribution of the matrix among the workers according to the Partition of the codistributor.
• 'noCommunication' — Specifies that no communication is to be performed when constructing the array, skipping some error checking steps. You can also use this argument with SD = spalloc(M, N, nzmax, codistr).

SD = spalloc(M,N,nzmax,codist) is the same as SD = codistributed.spalloc(M,N,nzmax,codist). You can also use the optional arguments with this syntax.

Examples

Allocate space for a 1000-by-1000 sparse codistributed matrix with room for up to 2000 nonzero elements. Use the default codistributor. Define several elements of the matrix.

spmd  % codistributed array created inside spmd statement
    N = 1000;

SD = codistributed.spalloc(N, N, 2*N);
for ii=1:N-1
    SD(ii,ii:ii+1) = [ii ii];
end
end

See Also
distributed.spalloc | spalloc | sparse

Introduced in R2009b
**codistributed.speye**

Create codistributed sparse identity matrix

**Syntax**

```matlab
CS = codistributed.speye(n)
CS = codistributed.speye(m,n)
CS = codistributed.speye([m,n])
CS = speye(n,codist)
CS = speye(m,n,codist)
CS = speye([m,n],codist)
```

**Description**

```matlab
CS = codistributed.speye(n) creates an n-by-n sparse codistributed array of underlying class double.

CS = codistributed.speye(m,n) or CS = codistributed.speye([m,n]) creates an m-by-n sparse codistributed array of underlying class double.

Optional arguments to codistributed.speye must be specified after the required arguments, and in the following order:

- **codist** — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.
- **'noCommunication'** — Specifies that no interworker communication is to be performed when constructing the array, skipping some error checking steps.

CS = speye(n,codist) is the same as CS = codistributed.speye(n,codist). You can also use the optional arguments with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:
```
```matlab
spmd
    CS = codistributed.speye(8,codistributor1d());
end

CS = speye(m,n,codist) and CS = speye([m,n],codist) are the same as CS = codistributed.speye(m,n) and CS = codistributed.speye([m,n]), respectively. You can also use the optional arguments with this syntax.

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

```matlab
CLS = logical(speye(m,n,codistributor1d()))
```

### Examples

With four workers,

```matlab
spmd(4)
    CS = speye(1000,codistributor());
end
```

creates a 1000-by-1000 sparse codistributed double array CS, distributed by its second dimension (columns). Each worker contains a 1000-by-250 local piece of CS.

```matlab
spmd(4)
    codist = codistributor1d(2,1:numlabs);
    CS = speye(10,10,codist);
end
```

creates a 10-by-10 sparse codistributed double array CS, distributed by its columns. Each worker contains a 10-by-labindex local piece of CS.

### See Also

distributed.speye | sparse | speye

**Introduced in R2009b**
codistributed.sprand

Create codistributed sparse array of uniformly distributed pseudo-random values

**Syntax**

CS = codistributed.sprand(m,n,density)
CS = sprand(n,codist)

**Description**

CS = codistributed.sprand(m,n,density) creates an m-by-n sparse codistributed array with approximately density*m*n uniformly distributed nonzero double entries.

Optional arguments to codistributed.sprand must be specified after the required arguments, and in the following order:

- **codist** — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.
- `'noCommunication'` — Specifies that no interworker communication is to be performed when constructing the array, skipping some error checking steps.

CS = sprand(n,codist) is the same as CS = codistributed.sprand(n, codist). You can also use the optional arguments with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:

```matlab
spmd
    CS = codistributed.sprand(8,8,0.2,codistributor1d());
end
```

**Examples**

With four workers,
spmd(4)
    CS = codistributed.sprand(1000,1000,0.001);
end

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

spmd(4)
    codist = codistributor1d(2,1:numlabs);
    CS = sprand(10,10,0.1,codist);
end

creates a 10-by-10 codistributed double array CS with approximately 10 nonzeros. CS is distributed by its columns, and each worker contains a 10-by-labindex local piece of CS.

**Tips**

When you use sprand on the workers in the parallel pool, or in an independent or communicating job (including pmode), each worker sets its random generator seed to a value that depends only on the labindex or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

**See Also**

distributed.sprandn | rand | sprand

*Introduced in R2009b*
**codistributed.sprandn**

Create codistributed sparse array of uniformly distributed pseudo-random values

**Syntax**

```plaintext
CS = codistributed.sprandn(m,n,density)
CS = sprandn(n,codist)
```

**Description**

`CS = codistributed.sprandn(m,n,density)` creates an `m`-by-`n` sparse codistributed array with approximately `density*m*n` normally distributed nonzero double entries.

Optional arguments to `codistributed.sprandn` must be specified after the required arguments, and in the following order:

- `codist` — A codistributor object specifying the distribution scheme of the resulting array. If omitted, the array is distributed using the default distribution scheme. For information on constructing codistributor objects, see the reference pages for `codistributor1d` and `codistributor2dbc`.

- `'noCommunication'` — Specifies that no interworker communication is to be performed when constructing the array, skipping some error checking steps.

`CS = sprandn(n,codist)` is the same as `CS = codistributed.sprandn(n,codist)`. You can also use the optional arguments with this syntax. To use the default distribution scheme, specify a codistributor constructor without arguments. For example:

```plaintext
spmd
    CS = codistributed.sprandn(8,8,0.2,codistributor1d());
end
```

**Examples**

With four workers,
```matlab
spmd(4)
    CS = codistributed.sprandn(1000,1000,0.001);
end
```

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

```matlab
spmd(4)
    codist = codistributor1d(2,1:numlabs);
    CS = sprandn(10,10,0.1,codist);
end
```

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

**Tips**

When you use `sprandn` on the workers in the parallel pool, or in an independent or communicating job (including pmode), each worker sets its random generator seed to a value that depends only on the `labindex` or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

**See Also**

codistributed.speye | codistributed.sprand | distributed.sprandn | rand | randn | sparse | sprandn

*Introduced in R2009b*
codistributor

Create codistributor object for codistributed arrays

Syntax

codist = codistributor()
codist = codistributor('1d')
codist = codistributor('1d',dim)
codist = codistributor('1d',dim,part)
codist = codistributor('2dbc')
codist = codistributor('2dbc',lbgrid)
codist = codistributor('2dbc',lbgrid,blksize)

Description

There are two schemes for distributing arrays. The scheme denoted by the character vector '1d' distributes an array along a single specified subscript, the distribution dimension, in a noncyclic, partitioned manner. The scheme denoted by '2dbc', employed by the parallel matrix computation software ScaLAPACK, applies only to two-dimensional arrays, and varies both subscripts over a rectangular computational grid of labs (workers) in a blocked, cyclic manner.

codist = codistributor(), with no arguments, returns a default codistributor object with zero-valued or empty parameters, which can then be used as an argument to other functions to indicate that the function is to create a codistributed array if possible with default distribution. For example,

Z = zeros(..., codistributor())
R = randn(..., codistributor())

codist = codistributor('1d') is the same as codist = codistributor().

codist = codistributor('1d',dim) also forms a codistributor object with codist.Dimension = dim and default partition.

codist = codistributor('1d',dim,part) also forms a codistributor object with codist.Dimension = dim and codist Partition = part.
codist = codistributor('2dbc') forms a 2-D block-cyclic codistributor object. For more information about '2dbc' distribution, see “2-Dimensional Distribution” on page 5-16.

codist = codistributor('2dbc',lbgrid) forms a 2-D block-cyclic codistributor object with the lab grid defined by lbgrid and with default block size.

codist = codistributor('2dbc',lbgrid,blksize) forms a 2-D block-cyclic codistributor object with the lab grid defined by lbgrid and with a block size defined by blksize.

codist = getCodistributor(D) returns the codistributor object of codistributed array D.

**Examples**

On four workers, create a 3-dimensional, 2-by-6-by-4 array with distribution along the second dimension, and partition scheme [1 2 1 2]. In other words, worker 1 contains a 2-by-1-by-4 segment, worker 2 a 2-by-2-by-4 segment, etc.

```matlab
spmd
    dim = 2; % distribution dimension
    codist = codistributor('1d',dim,[1 2 1 2],[2 6 4]);
    if mod(labindex,2)
        L = rand(2,1,4);
    else
        L = rand(2,2,4);
    end
    A = codistributed.build(L,codist)
end
A
```

On four workers, create a 20-by-5 codistributed array A, distributed by rows (over its first dimension) with a uniform partition scheme.

```matlab
spmd
    dim = 1; % distribution dimension
    partn = codistributor1d.defaultPartition(20);
    codist = codistributor('1d',dim,partn,[20 5]);
    L = magic(5) + labindex;
    A = codistributed.build(L,codist)
end
```
end
A

See Also
codistributed | codistributor1d | codistributor2dbc | getCodistributor |
getLocalPart | redistribute

Introduced in R2008b
### codistributor1d

Create 1-D codistributor object for codistributed arrays

#### Syntax

```matlab
codist = codistributor1d()
codist = codistributor1d(dim)
codist = codistributor1d(dim,part)
codist = codistributor1d(dim,part,gsize)
```

#### Description

The 1-D codistributor distributes arrays along a single, specified distribution dimension, in a noncyclic, partitioned manner.

- `codist = codistributor1d()` forms a codistributor1d object using default dimension and partition. The default dimension is the last nonsingleton dimension of the codistributed array. The default partition distributes the array along the default dimension as evenly as possible.

- `codist = codistributor1d(dim)` forms a 1-D codistributor object for distribution along the specified dimension: 1 distributes along rows, 2 along columns, etc.

- `codist = codistributor1d(dim,part)` forms a 1-D codistributor object for distribution according to the partition vector `part`. For example `C1 = codistributor1d(1,[1,2,3,4])` describes the distribution scheme for an array of ten rows to be codistributed by its first dimension (rows), to four workers, with 1 row to the first, 2 rows to the second, etc.

The resulting codistributor of any of the above syntax is incomplete because its global size is not specified. A codistributor constructed in this manner can be used as an argument to other functions as a template codistributor when creating codistributed arrays.

- `codist = codistributor1d(dim,part,gsize)` forms a codistributor object with distribution dimension `dim`, distribution partition `part`, and global size of its
codistributed arrays \texttt{gsize}. The resulting codistributor object is complete and can be used to build a codistributed array from its local parts with \texttt{codistributed.build}. To use a default dimension, specify \texttt{codistributor1d.unsetDimension} for that argument; the distribution dimension is derived from \texttt{gsize} and is set to the last non-singleton dimension. Similarly, to use a default partition, specify \texttt{codistributor1d.unsetPartition} for that argument; the partition is then derived from the default for that global size and distribution dimension.

The local part on worker \texttt{labidx} of a codistributed array using such a codistributor is of size \texttt{gsize} in all dimensions except \texttt{dim}, where the size is \texttt{part(labidx)}. The local part has the same class and attributes as the overall codistributed array. Conceptually, the overall global array could be reconstructed by concatenating the various local parts along dimension \texttt{dim}.

**Examples**

Use a \texttt{codistributor1d} object to create an N-by-N matrix of ones, distributed by rows.

```matlab
N = 1000;
spmd
codistr = codistributor1d(1); % 1st dimension (rows)
    C = ones(N,codistr);
end
```

Use a fully specified \texttt{codistributor1d} object to create a trivial N-by-N codistributed matrix from its local parts. Then visualize which elements are stored on worker 2.

```matlab
N = 1000;
spmd
codistr = codistributor1d( ... 
    codistributor1d.unsetDimension, ...
    codistributor1d.unsetPartition, ...
    [N,N]);
    myLocalSize = [N,N]; % start with full size on each lab
    % then set myLocalSize to default part of whole array:
    myLocalSize(codistr.Dimension) = codistr.Partition(labindex);
    myLocalPart = labindex*ones(myLocalSize); % arbitrary values
    D = codistributed.build(myLocalPart,codistr);
end
spy(D==2);
```
See Also

codistributed | codistributor2dbc | redistribute
codistributor1d.defaultPartition

Default partition for codistributed array

Syntax

P = codistributor1d.defaultPartition(n)

Description

P = codistributor1d.defaultPartition(n) is a vector with sum(P) = n and length(P) = numlabs. The first rem(n, numlabs) elements of P are equal to ceil(n/numlabs) and the remaining elements are equal to floor(n/numlabs). This function is the basis for the default distribution of codistributed arrays.

Examples

If numlabs = 4, the following code returns the vector [3 3 2 2] on all workers:

spmd
    P = codistributor1d.defaultPartition(10)
end

See Also

codistributed | codistributed.colon | codistributor1d

Introduced in R2009b
codistributor2dbc

Create 2-D block-cyclic codistributor object for codistributed arrays

Syntax

\[
\text{codist} = \text{codistributor2dbc}() \\
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}) \\
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}, \text{blksize}) \\
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}, \text{blksize}, \text{orient}) \\
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}, \text{blksize}, \text{orient}, \text{gsize})
\]

Description

The 2-D block-cyclic codistributor can be used only for two-dimensional arrays. It distributes arrays along two subscripts over a rectangular computational grid of labs (workers) in a block-cyclic manner. For a complete description of 2-D block-cyclic distribution, default parameters, and the relationship between block size and lab grid, see “2-Dimensional Distribution” on page 5-16. The 2-D block-cyclic codistributor is used by the ScaLAPACK parallel matrix computation software library.

\[
\text{codist} = \text{codistributor2dbc}() \text{ forms a 2-D block-cyclic codistributor2dbc} \\
\text{codist} \text{ codistributor object using default lab grid and block size.}
\]

\[
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}) \text{ forms a 2-D block-cyclic codistributor object} \\
\text{using the specified lab grid and default block size. lbgrid must be a two-element vector} \\
\text{defining the rows and columns of the lab grid, and the rows times columns must equal the} \\
\text{number of workers for the codistributed array.}
\]

\[
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}, \text{blksize}) \text{ forms a 2-D block-cyclic} \\
\text{codistributor object using the specified lab grid and block size.}
\]

\[
\text{codist} = \text{codistributor2dbc}(\text{lbgrid}, \text{blksize}, \text{orient}) \text{ allows an orientation} \\
\text{argument. Valid values for the orientation argument are 'row' for row orientation, and} \\
\text{'col' for column orientation of the lab grid. The default is row orientation.}
\]
The resulting codistributor of any of the above syntax is incomplete because its global size is not specified. A codistributor constructed this way can be used as an argument to other functions as a template codistributor when creating codistributed arrays.

codist = codistributor2dbc(lbgrid,blksize,orient,gsize) forms a codistributor object that distributes arrays with the global size gsize. The resulting codistributor object is complete and can therefore be used to build a codistributed array from its local parts with codistributed.build. To use the default values for lab grid, block size, and orientation, specify them using codistributor2dbc.defaultLabGrid, codistributor2dbc.defaultBlockSize, and codistributor2dbc.defaultOrientation, respectively.

Examples

Use a codistributor2dbc object to create an N-by-N matrix of ones.

N = 1000;
spmd
codistr = codistributor2dbc();
D = ones(N,codistr);
end

Use a fully specified codistributor2dbc object to create a trivial N-by-N codistributed matrix from its local parts. Then visualize which elements are stored on worker 2.

N = 1000;
spmd
codistr = codistributor2dbc(...
codistributor2dbc.defaultLabGrid, ...
codistributor2dbc.defaultBlockSize, ...
'row',[N,N]);
myLocalSize = [length(codistr.globalIndices(1)), ...
    length(codistr.globalIndices(2))];
myLocalPart = labindex*ones(myLocalSize);
D = codistributed.build(myLocalPart,codistr);
end
spy(D==2);
See Also

codistributed | codistributor1d | getLocalPart | redistribute
codistributor2dbc.defaultLabGrid

Default computational grid for 2-D block-cyclic distributed arrays

Syntax

grid = codistributor2dbc.defaultLabGrid()

Description

grid = codistributor2dbc.defaultLabGrid() returns a vector, grid = [nrow ncol], defining a computational grid of nrow-by-ncol workers in the open parallel pool, such that numlabs = nrow * ncol.

The grid defined by codistributor2dbc.defaultLabGrid is as close to a square as possible. The following rules define nrow and ncol:

• If numlabs is a perfect square, nrow = ncol = sqrt(numlabs).
• If numlabs is an odd power of 2, then nrow = ncol/2 = sqrt(numlabs/2).
• nrow <= ncol.
• If numlabs is a prime, nrow = 1, ncol = numlabs.
• nrow is the greatest integer less than or equal to sqrt(numlabs) for which ncol = numlabs/nrow is also an integer.

Examples

View the computational grid layout of the default distribution scheme for the open parallel pool.

    spmd
    grid = codistributor2dbc.defaultLabGrid
    end
See Also

codistributed | codistributor2dbc | numlabs
**Composite**

Create Composite object

**Syntax**

\[ C = \text{Composite}() \]

\[ C = \text{Composite}(\text{nlabs}) \]

**Description**

\[ C = \text{Composite}() \] creates a Composite object on the client using workers from the parallel pool. The actual number of workers referenced by this Composite object depends on the size of the pool and any existing Composite objects. Generally, you should construct Composite objects outside any \texttt{spmd} statement.

\[ C = \text{Composite}(\text{nlabs}) \] creates a Composite object on the parallel pool set that matches the specified constraint. \texttt{nlabs} must be a vector of length 1 or 2, containing integers or \texttt{Inf}. If \texttt{nlabs} is of length 1, it specifies the exact number of workers to use. If \texttt{nlabs} is of size 2, it specifies the minimum and maximum number of workers to use. The actual number of workers used is the maximum number of workers compatible with the size of the parallel pool, and with other existing Composite objects. An error is thrown if the constraints on the number of workers cannot be met.

A Composite object has one entry for each lab; initially each entry contains no data. Use either indexing or an \texttt{spmd} block to define values for the entries.

**Examples**

The following examples all use a local parallel pool of four workers, opened with the statement:

\[ p = \text{parpool}('\text{local}',4); \]

This example shows how to create a Composite object with no defined elements, then assign values using a \texttt{for}-loop in the client.
c = Composite();  % One element per worker in the pool
for w = 1:length(c)
    c{w} = 0;    % Value stored on each worker
end

This example shows how to assign Composite elements in an spmd block.

c = Composite();
spmd
    c = 0;    % Value stored on each worker
end

This example shows how to assign the elements of a Composite with a value from each worker.

c = Composite();
spmd
    c = labindex;
end
c{:}

1
2
3
4

This example shows how to use a distributed array vector to set the values of a Composite.

d = distributed([3 1 4 2]);  % One integer per worker
spmd
    c = getLocalPart(d);    % Unique value on each worker
end
c{::}

3
1
Tips

• A Composite is created on the workers of the existing parallel pool. If no pool exists, 
  Composite starts a new parallel pool, unless the automatic starting of pools is 
  disabled in your parallel preferences. If there is no parallel pool and Composite 
  cannot start one, the result is a 1-by-1 Composite in the client workspace.

See Also
parallel.pool.Constant | parpool | spmd
createCommunicatingJob

Create communicating job on cluster

Syntax

job = createCommunicatingJob(cluster)
job = createCommunicatingJob(...,'p1',v1,'p2',v2,...)
job = createCommunicatingJob(...,'Type','pool',...)
job = createCommunicatingJob(...,'Type','spmd',...)
job = createCommunicatingJob(...,'Profile','profileName',...)

Description

job = createCommunicatingJob(cluster) creates a communicating job object for the identified cluster.

job = createCommunicatingJob(...,'p1',v1,'p2',v2,...) creates a communicating job object with the specified property values. For a listing of the valid properties of the created object, see the parallel.Job object reference page. The property name must be a character vector, with the value being the appropriate type for that property. In most cases, the values specified in these property-value pairs override the values in the profile. But when you specify AttachedFiles or AdditionalPaths at the time of creating a job, the settings are combined with those specified in the applicable profile. If an invalid property name or property value is specified, the object will not be created.

job = createCommunicatingJob(...,'Type','pool',...) creates a communicating job of type 'pool'. This is the default if 'Type' is not specified. A 'pool' job runs the specified task function with a parallel pool available to run the body of parfor loops or spmd blocks. Note that only one worker runs the task function, and the rest of the workers in the cluster form the parallel pool. So on a cluster of N workers for a 'pool' type job, only N-1 workers form the actual pool that performs the spmd and parfor code found within the task function.
job = createCommunicatingJob(...,'Type','spmd',...) creates a communicating job of type 'spmd', where the specified task function runs simultaneously on all workers, and lab* functions can be used for communication between workers.

job = createCommunicatingJob(...,'Profile','profileName',...) creates a communicating job object with the property values specified in the profile 'profileName'. If no profile is specified and the cluster object has a value specified in its 'Profile' property, the cluster’s profile is automatically applied.

**Examples**

**Example 11.1. Pool Type Communicating Job**

Consider the function 'myFunction' which uses a `parfor` loop:

```matlab
function result = myFunction(N)
    result = 0;
    parfor ii=1:N
        result = result + max(eig(rand(ii)));
    end
end
```

Create a communicating job object to evaluate `myFunction` on the default cluster:

```matlab
myCluster = parcluster;
j = createCommunicatingJob(myCluster,'Type','pool');
```

Add the task to the job, supplying an input argument:

```matlab
createTask(j, @myFunction, 1, {100});
```

Set the number of workers required for parallel execution:

```matlab
j.NumWorkersRange = [5 10];
```

Run the job.

```matlab
submit(j);
```

Wait for the job to finish and retrieve its results:

```matlab
wait(j)
out = fetchOutputs(j)
```
Delete the job from the cluster.

delete(j);

See Also
createJob|createTask|findJob|parcluster|recreate|submit

Introduced in R2012a
createJob

Create independent job on cluster

Syntax

\[
\text{obj} = \text{createJob}(\text{cluster}) \\
\text{obj} = \text{createJob}(...,'p1',v1,'p2',v2,...) \\
\text{job} = \text{createJob}(...,'Profile','profileName',...) \\
\]

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The job object.</td>
</tr>
<tr>
<td>cluster</td>
<td>The cluster object created by parcluster.</td>
</tr>
<tr>
<td>p1, p2</td>
<td>Object properties configured at object creation.</td>
</tr>
<tr>
<td>v1, v2</td>
<td>Initial values for corresponding object properties.</td>
</tr>
</tbody>
</table>

Description

\[
\text{obj} = \text{createJob}(\text{cluster}) \text{ creates an independent job object for the identified cluster.} \\
\text{obj} = \text{createJob}(...,'p1',v1,'p2',v2,...) \text{ creates a job object with the specified property values. For a listing of the valid properties of the created object, see the parallel.Job object reference page. The property name must be a character vector, with the value being the appropriate type for that property. In most cases, the values specified in these property-value pairs override the values in the profile; but when you specify AttachedFiles or AdditionalPaths at the time of creating a job, the settings are combined with those specified in the applicable profile. If an invalid property name or property value is specified, the object will not be created.} \\
\]

11-80
job = createJob(...,'Profile','profileName',...) creates an independent job object with the property values specified in the profile 'profileName'. If a profile is not specified and the cluster has a value specified in its 'Profile' property, the cluster’s profile is automatically applied. For details about defining and applying profiles, see “Discover Clusters and Use Cluster Profiles” on page 6-15.

**Examples**

**Example 11.2. Create and Run a Basic Job**

Construct an independent job object using the default profile.

```matlab
c = parcluster
j = createJob(c);
```

Add tasks to the job.

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

Run the job.

```matlab
submit(j);
```

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

```matlab
wait(j);
out = fetchOutputs(j);
```

Display the random matrix returned from the third task.

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

Delete the job.

```matlab
delete(j);
```

**Example 11.3. Create a Job with Attached Files**

Construct an independent job with attached files in addition to those specified in the default profile.
c = parcluster
j = createJob(c,'AttachedFiles',...  
    {'myapp/folderA','myapp/folderB','myapp/file1.m'});

See Also
createCommunicatingJob | createTask | findJob | parcluster | recreate | submit

Introduced before R2006a
createTask

Create new task in job

Syntax

t = createTask(j, F, N, {inputargs})
t = createTask(j, F, N, {C1,...,Cm})
t = createTask(..., 'p1',v1,'p2',v2,...)
t = createTask(...,'Profile', 'ProfileName',...)

Arguments

t
j
F
N
{inputargs}
{C1,...,Cm}
p1,p2
v1,v2

Task object or vector of task objects.
The job that the task object is created in.
A handle to the function that is called when the task is evaluated, or an array of function handles.
The number of output arguments to be returned from execution of the task function. This is a double or array of doubles.
A row cell array specifying the input arguments to be passed to the function F. Each element in the cell array will be passed as a separate input argument. If this is a cell array of cell arrays, a task is created for each cell array.
Cell array of cell arrays defining input arguments to each of m tasks.
Task object properties configured at object creation.
Initial values for corresponding task object properties.
**Description**

\( t = \text{createTask}(j, F, N, \{\text{inputargs}\}) \) creates a new task object in job \( j \), and returns a reference, \( t \), to the added task object. This task evaluates the function specified by a function handle or function name \( F \), with the given input arguments \( \{\text{inputargs}\} \), returning \( N \) output arguments.

\( t = \text{createTask}(j, F, N, \{C_1,\ldots,C_m\}) \) uses a cell array of \( m \) cell arrays to create \( m \) task objects in job \( j \), and returns a vector, \( t \), of references to the new task objects. Each task evaluates the function specified by a function handle or function name \( F \). The cell array \( C_1 \) provides the input arguments to the first task, \( C_2 \) to the second task, and so on, so that there is one task per cell array. Each task returns \( N \) output arguments. If \( F \) is a cell array, each element of \( F \) specifies a function for each task in the vector; it must have \( m \) elements. If \( N \) is an array of doubles, each element specifies the number of output arguments for each task in the vector. Multidimensional matrices of inputs \( F \), \( N \) and \( \{C_1,\ldots,C_m\} \) are supported; if a cell array is used for \( F \), or a double array for \( N \), its dimensions must match those of the input arguments cell array of cell arrays. The output \( t \) will be a vector with the same number of elements as \( \{C_1,\ldots,C_m\} \). Note that because a communicating job has only one task, this form of vectorized task creation is not appropriate for such jobs.

\( t = \text{createTask}(\ldots, 'p1',v1,'p2',v2,\ldots) \) adds a task object with the specified property values. For a listing of the valid properties of the created object, see the parallel.Task object reference page. The property name must be a character vector, with the value being the appropriate type for that property. The values specified in these property-value pairs override the values in the profile. If an invalid property name or property value is specified, the object will not be created.

\( t = \text{createTask}(\ldots,'Profile', 'ProfileName',\ldots) \) creates a task object with the property values specified in the cluster profile \( ProfileName \). For details about defining and applying cluster profiles, see “Discover Clusters and Use Cluster Profiles” on page 6-15.

**Examples**

**Example 11.4. Create a Job with One Task**

Create a job object.
c = parcluster(); % Use default profile
j = createJob(c);

Add a task object which generates a 10-by-10 random matrix.
t = createTask(j, @rand, 1, {10,10});

Run the job.
submit(j);

Wait for the job to finish running, and get the output from the task evaluation.
wait(j);
taskoutput = fetchOutputs(j);

Show the 10-by-10 random matrix.
disp(taskoutput{1});

**Example 11.5. Create a Job with Three Tasks**

This example creates a job with three tasks, each of which generates a 10-by-10 random matrix.

c = parcluster(); % Use default profile
j = createJob(c);
t = createTask(j, @rand, 1, {{10,10} {10,10} {10,10}});

**Example 11.6. Create a Task with Different Property Values**

This example creates a task that captures the worker diary, regardless of the setting in the profile.

c = parcluster(); % Use default profile
j = createJob(c);
t = createTask(j,@rand,1,{10,10},'CaptureDiary',true);

**See Also**
createCommunicatingJob|createJob|findTask|recreate

**Introduced before R2006a**
delete

Remove job or task object from cluster and memory

Syntax

delete(obj)

Description

delete(obj) removes the job or task object, obj, from the local MATLAB session, and removes it from the cluster’s JobStorageLocation. When the object is deleted, references to it become invalid. Invalid objects should be removed from the workspace with the clear command. If multiple references to an object exist in the workspace, deleting one reference to that object invalidates the remaining references to it. These remaining references should be cleared from the workspace with the clear command.

When you delete a job object, this also deletes all the task objects contained in that job. Any references to those task objects will also be invalid, and you should clear them from the workspace.

If obj is an array of objects and one of the objects cannot be deleted, the other objects in the array are deleted and a warning is returned.

Because its data is lost when you delete an object, delete should be used only after you have retrieved all required output data from the effected object.

Examples

Create a job object using the default profile, then delete the job:

```matlab
myCluster = parcluster;
j = createJob(myCluster,'Name','myjob');
t = createTask(j,@rand,1,{10});
delete(j);
clear j t
```
Delete all jobs on the cluster identified by the profile `myProfile`:

```matlab
myCluster = parcluster('myProfile');
delete(myCluster.Jobs)
```

**See Also**

`batch | createJob | createTask | findJob | findTask | wait`

**Introduced in R2012a**
delete (Pool)

Shut down parallel pool

Syntax

delete(poolobj)

Description

delete(poolobj) shuts down the parallel pool associated with the object poolobj, and destroys the communicating job that comprises the pool. Subsequent parallel language features will automatically start a new parallel pool, unless your parallel preferences disable this behavior.

References to the deleted pool object become invalid. Invalid objects should be removed from the workspace with the clear command. If multiple references to an object exist in the workspace, deleting one reference to that object invalidates the remaining references to it. These remaining references should be cleared from the workspace with the clear command.

Examples

Get the current pool and shut it down.

poolobj = gcp('nocreate');
delete(poolobj);

See Also

gcp | parpool

Introduced in R2013b
demote

Demote job in cluster queue

Syntax

demote(c,job)

Arguments

c            Cluster object that contains the job.
job           Job object demoted in the job queue.

Description

demote(c,job) demotes the job object job that is queued in the cluster c.

If job is not the last job in the queue, demote exchanges the position of job and the job that follows it in the queue.

Examples

Create and submit multiple jobs to the MATLAB job scheduler (MJS) identified by the default parallel configuration:

c = parcluster();
pause(c) % Prevent submissions from running.

j1 = createJob(c,'Name','Job A');
j2 = createJob(c,'Name','Job B');
j3 = createJob(c,'Name','Job C');
submit(j1);submit(j2);submit(j3);

Demote one of the jobs by one position in the queue:
demote(c,j2)

Examine the new queue sequence:

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

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

**Tips**

After a call to `demote` or `promote`, there is no change in the order of job objects contained in the Jobs property of the cluster object. To see the scheduled order of execution for jobs in the queue, use the `findJob` function in the form `jobs = findJob(c)`. 

**See Also**

`createJob` | `findJob` | `promote` | `submit`

*Introduced before R2006a*
diary

Display or save Command Window text of batch job

Syntax

diary(job)
diary(job, 'filename')

Arguments

job           Job from which to view Command Window output text.
'filename'    File to append with Command Window output text from batch job

Description

diary(job) displays the Command Window output from the batch job in the MATLAB Command Window. The Command Window output will be captured only if the batch command included the 'CaptureDiary' argument with a value of true.

diary(job, 'filename') causes the Command Window output from the batch job to be appended to the specified file.

The captured Command Window output includes only the output generated by execution of the task function. Output is not captured from code that runs asynchronously from the task.

See Also

batch | diary | load

Introduced in R2008a
distributed

Create distributed array from data in the client workspace or a datastore

Syntax

D = distributed(X)

Description

D = distributed(X) creates a distributed array from X. X can be an array stored on the MATLAB client workspace or a datastore. D is a distributed array stored in parts on the workers of the open parallel pool.

Constructing a distributed array from local data this way is appropriate only if the MATLAB client can store the entirety of X in its memory. To construct large distributed arrays, use one of the constructor methods such as ones(___,'distributed'), zeros(___,'distributed'), etc.

If the input argument is already a distributed array, the result is the same as the input.

Use gather to retrieve the distributed array elements from the pool back to an array in the MATLAB workspace.

Examples

Create a small array and distribute it:

Nsmall = 50;
D1 = distributed(magic(Nsmall));

Create a large distributed array directly, using a build method:

Nlarge = 1000;
D2 = rand(Nlarge,'distributed');
Retrieve elements of a distributed array, and note where the arrays are located by their Class:

D3 = gather(D2);
whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>50x50</td>
<td>733</td>
<td>distributed</td>
</tr>
<tr>
<td>D2</td>
<td>1000x1000</td>
<td>733</td>
<td>distributed</td>
</tr>
<tr>
<td>D3</td>
<td>1000x1000</td>
<td>8000000</td>
<td>double</td>
</tr>
<tr>
<td>Nlarge</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Nsmall</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>

This example shows how to create and load distributed arrays using datastore. You first create a datastore using an example data set. This data set is too small to show equal partitioning of the data over the workers. To simulate a real big data set, artificially increase the size of the datastore using repmat:

files = repmat({'airlinesmall.csv'}, 10, 1);
ds = tabularTextDatastore(files);

Select the example variables:

ds.SelectedVariableNames = {'DepTime','DepDelay'};
ds.TreatAsMissing = 'NA';

Create a distributed table by reading the datastore in parallel. Partition the datastore with one partition per worker. Each worker then reads all data from the corresponding partition. The files must be in a shared location accessible from the workers.

dt = distributed(ds);

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

Finally, display summary information about the distributed table:

summary(dt)

Variables:

DepTime: 1,235,230×1 double

Values:

    min     1
max       2505
NaNs     23,510

DepDelay: 1,235,230×1 double
Values:

    min      -1036
    max       1438
    NaNs     23,510

Tips

- A distributed array is created on the workers of the existing parallel pool. If no pool exists, `distributed` will start a new parallel pool, unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `distributed` cannot start one, the result is the full array in the client workspace.

See Also

codistributed | datastore | gather | ones | parpool | spmd | tall | zeros
distributed.cell

Create distributed cell array

**Syntax**

\[
D = \text{distributed.cell}(n) \\
D = \text{distributed.cell}(m, n, p, ...) \\
D = \text{distributed.cell}([m, n, p, ...])
\]

**Description**

\(D = \text{distributed.cell}(n)\) creates an \(n\)-by-\(n\) distributed array of underlying class cell.

\(D = \text{distributed.cell}(m, n, p, ...)\) or \(D = \text{distributed.cell}([m, n, p, ...])\) create an \(m\)-by-\(n\)-by-\(p\)-by-... distributed array of underlying class cell.

**Examples**

Create a distributed 1000-by-1000 cell array:

\[
D = \text{distributed.cell}(1000)
\]

**See Also**

cell | codistributed.cell

**Introduced in R2009b**
distributed.spalloc

Allocate space for sparse distributed matrix

Syntax

SD = distributed.spalloc(M,N,nzmax)

Description

SD = distributed.spalloc(M,N,nzmax) creates an M-by-N all-zero sparse distributed matrix with room to hold nzmax nonzeros.

Examples

Allocate space for a 1000-by-1000 sparse distributed matrix with room for up to 2000 nonzero elements, then define several elements:

N = 1000;
SD = distributed.spalloc(N,N,2*N);
for ii=1:N-1
    SD(ii,ii:ii+1) = [ii ii];
end

See Also

codistributed.spalloc | spalloc | sparse

Introduced in R2009b
distributed.speye

Create distributed sparse identity matrix

Syntax

DS = distributed.speye(n)
DS = distributed.speye(m,n)
DS = distributed.speye([m,n])

Description

DS = distributed.speye(n) creates an n-by-n sparse distributed array of underlying class double.

DS = distributed.speye(m,n) or DS = distributed.speye([m,n]) creates an m-by-n sparse distributed array of underlying class double.

Examples

Create a distributed 1000-by-1000 sparse identity matrix:

```plaintext
N = 1000;
DS = distributed.speye(N);
```

See Also

codistributed.speye | eye | speye

Introduced in R2009b
distributed.sprand

Create distributed sparse array of uniformly distributed pseudo-random values

Syntax

DS = distributed.sprand(m,n,density)

Description

DS = distributed.sprand(m,n,density) creates an m-by-n sparse distributed array with approximately density*m*n uniformly distributed nonzero double entries.

Examples

Create a 1000-by-1000 sparse distributed double array DS with approximately 1000 nonzeros.

DS = distributed.sprand(1000,1000,0.001);

Tips

When you use sprand on the workers in the parallel pool, or in an independent or communicating job (including pmode), each worker sets its random generator seed to a value that depends only on the labindex or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

See Also

codistributed.sprand | distributed.speye | distributed.sprandn | rand | randn | sparse | sprand
Introduced in R2009b
distributed.sprandn

Create distributed sparse array of normally distributed pseudo-random values

Syntax

DS = distributed.sprandn(m,n,density)

Description

DS = distributed.sprandn(m,n,density) creates an m-by-n sparse distributed array with approximately density*m*n normally distributed nonzero double entries.

Examples

Create a 1000-by-1000 sparse distributed double array DS with approximately 1000 nonzeros.

DS = distributed.sprandn(1000,1000,0.001);

Tips

When you use sprandn on the workers in the parallel pool, or in an independent or communicating job (including pmode), each worker sets its random generator seed to a value that depends only on the labindex or task ID. Therefore, the array on each worker is unique for that job. However, if you repeat the job, you get the same random data.

See Also

codistributed.sprandn | distributed.speye | distributed.sprand | rand | randn | sparse | sprandn
Introduced in R2009b
**dload**

Load distributed arrays and Composite objects from disk

**Syntax**

```
dload

  dload filename
  dload filename X
  dload filename X Y Z ...
  dload -scatter ...

  [X,Y,Z,...] = dload('filename','X','Y','Z',...)
```

**Description**

dload without any arguments retrieves all variables from the binary file named `matlab.mat`. If `matlab.mat` is not available, the command generates an error.

`dload filename` retrieves all variables from a file given a full pathname or a relative partial pathname. If `filename` has no extension, `dload` looks for `filename.mat`. `dload` loads the contents of distributed arrays and Composite objects onto parallel pool workers, other data types are loaded directly into the workspace of the MATLAB client.

`dload filename X` loads only variable `X` from the file. `dload filename X Y Z ...` loads only the specified variables. `dload` does not support wildcards, nor the `-regexp` option. If any requested variable is not present in the file, a warning is issued.

`dload -scatter ...` distributes nondistributed data if possible. If the data cannot be distributed, a warning is issued.

```
[X,Y,Z,...] = dload('filename','X','Y','Z','Z',...)
```
returns the specified variables as separate output arguments (rather than a structure, which the `load` function returns). If any requested variable is not present in the file, an error occurs.

When loading distributed arrays, the data is distributed over the available parallel pool workers using the default distribution scheme. It is not necessary to have the same size pool open when loading as when saving using `dsave`. 
When loading Composite objects, the data is sent to the available parallel pool workers. If the Composite is too large to fit on the current parallel pool, the data is not loaded. If the Composite is smaller than the current parallel pool, a warning is issued.

**Examples**

Load variables X, Y, and Z from the file *fname.mat*:

```matlab
dload fname X Y Z
```

Use the function form of `dload` to load distributed arrays P and Q from file *fname.mat*:

```matlab
[P,Q] = dload('fname.mat','P','Q');
```

**See Also**

Composite | distributed | dsave | load | parpool

*Introduced in R2010a*
dsave

Save workspace distributed arrays and Composite objects to disk

Syntax

dsave

dsave filename

dsave filename X

dsave filename X Y Z

Description

dsave without any arguments creates the binary file named matlab.mat and writes to the file all workspace variables, including distributed arrays and Composite objects. You can retrieve the variable data using dload.

dsave filename saves all workspace variables to the binary file named filename.mat. If you do not specify an extension for filename, it assumes the extension .mat.

dsave filename X saves only variable X to the file.

dsave filename X Y Z saves X, Y, and Z. dsave does not support wildcards, nor the -regexp option.

dsave does not support saving sparse distributed arrays.

Examples

With a parallel pool open, create and save several variables to mydatafile.mat:

D = rand(1000,'distributed'); % Distributed array
C = Composite(); %
C{1} = magic(20); % Data on worker 1 only
X = rand(40); % Client workspace only
dssave mydatafile D C X % Save all three variables
See Also
Composite | distributed | dload | parpool | save

Introduced in R2010a
exist

Check whether Composite is defined on workers

Syntax

h = exist(C,labidx)
h = exist(C)

Description

h = exist(C,labidx) returns true if the entry in Composite C has a defined value on the worker with labindex labidx, false otherwise. In the general case where labidx is an array, the output h is an array of the same size as labidx, and h(i) indicates whether the Composite entry labidx(i) has a defined value.

h = exist(C) is equivalent to h = exist(C, 1:length(C)).

If exist(C,labidx) returns true, C(labidx) does not throw an error, provided that the values of C on those workers are serializable. The function throws an error if any labidx is invalid.

Examples

Define a variable on a random number of workers. Check on which workers the Composite entries are defined, and get all those values:

```matlab
spmd
    if rand() > 0.5
        c = labindex;
    end
end
ind = exist(c);
cvals = c(ind);
```
See Also

Composite

Introduced in R2008b
existsOnGPU

Determine if gpuArray or CUDAKernel is available on GPU

Syntax

TF = existsOnGPU(DATA)

Description

TF = existsOnGPU(DATA) returns a logical value indicating whether the gpuArray or CUDAKernel object represented by DATA is still present on the GPU and available from your MATLAB session. The result is false if DATA is no longer valid and cannot be used. Such arrays and kernels are invalidated when the GPU device has been reset with any of the following:

reset(dev) % Where dev is the current gpuDevice
gpuDevice(ix) % Where ix is valid index of current or different device
gpuDevice([]) % With an empty argument (as opposed to no argument)

Examples

Query Existence of gpuArray

Create a gpuArray on the selected GPU device, then reset the device. Query array’s existence and content before and after resetting.

g = gpuDevice(1);
M = gpuArray(magic(4));
M_exists = existsOnGPU(M)

1

M % Display gpuArray

16 2 3 13
5 11 10 8
reset(g);
M_exists = existsOnGPU(M)

0

M  % Try to display gpuArray

Data no longer exists on the GPU.
clear M

See Also

gpuArray | gpuDevice | parallel.gpu.CUDAKernel | reset

Introduced in R2012a
eye
Identity matrix

Syntax

\[
E = \text{eye}(sz, \text{arraytype})
\]
\[
E = \text{eye}(sz, \text{datatype}, \text{arraytype})
\]
\[
E = \text{eye}(sz, 'like', P)
\]
\[
E = \text{eye}(sz, \text{datatype}, 'like', P)
\]
\[
C = \text{eye}(sz, \text{codist})
\]
\[
C = \text{eye}(sz, \text{datatype}, \text{codist})
\]
\[
C = \text{eye}(sz, __, \text{codist}, 'noCommunication')
\]
\[
C = \text{eye}(sz, __, \text{codist}, 'like', P)
\]

Description

\[
E = \text{eye}(sz, \text{arraytype})
\]
creates an \text{arraytype} identity matrix with underlying class of double, with ones on the main diagonal and zeros elsewhere.

\[
E = \text{eye}(sz, \text{datatype}, \text{arraytype})
\]
creates an \text{arraytype} identity matrix with underlying class of \text{datatype}, with ones on the main diagonal and zeros elsewhere.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sz)</td>
<td>(n)</td>
<td>Specifies array size as an (n)-by-(n) matrix.</td>
</tr>
<tr>
<td></td>
<td>(m,n)</td>
<td>Specifies array size as an (m)-by-(n) matrix.</td>
</tr>
<tr>
<td></td>
<td>([m\ n])</td>
<td></td>
</tr>
<tr>
<td>(arraytype)</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
<tr>
<td>Argument</td>
<td>Values</td>
<td>Descriptions</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>'codistributed'</td>
<td>Specifies codistributed array, using the default</td>
</tr>
<tr>
<td></td>
<td>'gpuArray'</td>
<td>distribution scheme.</td>
</tr>
<tr>
<td>datatype</td>
<td>'double' (default),</td>
<td>Specifies underlying class of the array, i.e.,</td>
</tr>
<tr>
<td></td>
<td>'single', 'int8',</td>
<td>the data type of its elements.</td>
</tr>
<tr>
<td></td>
<td>'uint8', 'int16',</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'uint16', 'int32',</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'uint32', 'int64',</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'uint64'</td>
<td></td>
</tr>
</tbody>
</table>

E = eye(sz, 'like', P) creates an identity matrix of the same type and underlying class (data type) as array P.

E = eye(sz, datatype, 'like', P) creates an identity matrix of the specified underlying class (datatype), and the same type as array P.

C = eye(sz, codist) or C = eye(sz, datatype, codist) creates a codistributed identity matrix of the specified size and underlying class (the default datatype is 'double'). The codistributor object, codist, specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```matlab
spmd
    C = eye(8, codistributor1d());
end
```

C = eye(sz, __, codist, 'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = eye(sz, __, codist, 'like', P) creates a codistributed identity matrix with the specified size, underlying class (datatype), and distribution scheme. If either the datatype or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.
Examples

Create Distributed Identity Matrix

Create a 1000-by-1000 distributed identity matrix of underlying class double:

\[
D = \text{eye}(1000,'\text{distributed}');
\]

Create Codistributed Identity Matrix

Create a 1000-by-1000 codistributed double identity matrix, distributed by its second dimension (columns).

\[
\text{spmd}(4) \\
\quad C = \text{eye}(1000,'\text{codistributed}'); \\
\text{end}
\]

With four workers, each worker contains a 1000-by-250 local piece of \( C \).

Create a 1000-by-1000 codistributed \texttt{uint16} identity matrix, distributed by its columns.

\[
\text{spmd}(4) \\
\quad \text{codist} = \text{codistributor}('1d',2,100*[1:numlabs]); \\
\quad C = \text{eye}(1000,1000,'\text{uint16}',\text{codist}); \\
\text{end}
\]

Each worker contains a 100-by-	ext{labindex} local piece of \( C \).

Create \texttt{gpuArray} Identity Matrix

Create a 1000-by-1000 \texttt{gpuArray} identity matrix of underlying class \texttt{uint32}:

\[
G = \text{eye}(1000,'\text{uint32}','\text{gpuArray}');
\]

See Also

\texttt{Inf} | \texttt{NaN} | \texttt{codistributed.speye} | \texttt{distributed.speye} | \texttt{eye} | \texttt{false} | \texttt{ones} | \texttt{true} | \texttt{zeros}

Introduced in R2006b
false

Array of logical 0 (false)

Syntax

F = false(sz,arraytype)

F = false(sz,'like',P)

C = false(sz,codist)
C = false(sz,___,codist,'noCommunication')
C = false(sz,___,codist,'like',P)

Description

F = false(sz,arraytype) creates a matrix with false values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n,...,k or [m n ... k]</td>
<td>Specifies size as an m-by-n-by-....-by-k array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
<tr>
<td></td>
<td>'codistributed'</td>
<td>Specifies codistributed array, using the default distribution scheme.</td>
</tr>
<tr>
<td></td>
<td>'gpuArray'</td>
<td>Specifies gpuArray.</td>
</tr>
</tbody>
</table>

F = false(sz,'like',P) creates an array of false values with the same type as array P.
C = false(sz,codist) creates a codistributed array of false values with the specified size. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```matlab
spmd
    C = false(8,codistributor1d());
end
```

C = false(sz,___,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = false(sz,___,codist,'like',P) creates a codistributed array of false values with the specified size and distribution scheme. If the codistributor argument is omitted, the distribution scheme is taken from the codistributed array P.

### Examples

#### Create Distributed False Matrix

Create a 1000-by-1000 distributed array of falses with underlying class double:

```matlab
D = false(1000,'distributed');
```

#### Create Codistributed False Matrix

Create a 1000-by-1000 codistributed matrix of falses, distributed by its second dimension (columns).

```matlab
spmd(4)
    C = false(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed matrix of falses, distributed by its columns.
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = false(1000,1000,codist);
end

Each worker contains a 100-by-
               labindex local piece of C.

Create gpuArray False Matrix

Create a 1000-by-1000 gpuArray of falses:
G = false(1000,'gpuArray');

See Also
Inf | NaN | eye | false | ones | true | zeros

Introduced in R2006b
fetchNext

Retrieve next available unread FevalFuture outputs

Syntax

\[ [\text{id}x, B_1, B_2, \ldots, B_n] = \text{fetchNext}(F) \]
\[ [\text{id}x, B_1, B_2, \ldots, B_n] = \text{fetchNext}(F, \text{TIMEOUT}) \]

Description

\[ [\text{id}x, B_1, B_2, \ldots, B_n] = \text{fetchNext}(F) \] waits for an unread FevalFuture in the array of futures \( F \) to finish, and then returns the linear index of that future in array \( F \) as \( \text{id}x \), along with the future’s results in \( B_1, B_2, \ldots, B_n \). Before this call, the 'Read' property of the particular future is false; afterward it is true.

\[ [\text{id}x, B_1, B_2, \ldots, B_n] = \text{fetchNext}(F, \text{TIMEOUT}) \] waits no longer than \( \text{TIMEOUT} \) seconds for a result to become available. If the timeout expires before any result becomes available, all output arguments are empty.

If there are no futures in \( F \) whose 'Read' property is false, then an error is reported. You can check whether there are any unread futures using

\[ \text{anyUnread} = \sim \text{all}([F.\text{Read}]) \]

If the element of \( F \) which has become finished encountered an error during execution, that error will be thrown by \text{fetchNext}. However, that future’s 'Read' property is set true, so that any subsequent calls to \text{fetchNext} can proceed.

Examples

Request several function evaluations, and update a progress bar while waiting for completion.

\[ N = 100; \]
\[ \text{for} \ \text{idx} = N:-1:1 \]
\[ \quad \text{% Compute the rank of N magic squares} \]
F(idx) = parfeval(@rank,1,magic(idx));
end

% Build a waitbar to track progress
h = waitbar(0,'Waiting for FevalFutures to complete...');
results = zeros(1,N);
for idx = 1:N
    [completedIdx,thisResult] = fetchNext(F);
    % store the result
    results(completedIdx) = thisResult;
    % update waitbar
    waitbar(idx/N,h,sprintf('Latest result: %d',thisResult));
end
delete(h)

**Tips**

The `fetchNext` function returns the linear index of the future from its array. If instead, you need the subscript values of a multidimensional array, you can use the `ind2sub` function to convert the values.

**See Also**

fetchOutputs | isequal | parfeval | parfevalOnAll | parpool

*Introduced in R2013b*
fetchOutputs (Job)

Retrieve output arguments from all tasks in job

Syntax

data = fetchOutputs(job)

Description

data = fetchOutputs(job) retrieves the output arguments contained in the tasks of a finished job. If the job has M tasks, each row of the M-by-N cell array data contains the output arguments for the corresponding task in the job. Each row has N elements, where N is the greatest number of output arguments from any one task in the job. The N elements of a row are arrays containing the output arguments from that task. If a task has less than N output arguments, the excess arrays in the row for that task are empty. The order of the rows in data is the same as the order of the tasks contained in the job’s Tasks property.

Calling fetchOutputs does not remove the output data from the location where it is stored. To remove the output data, use the delete function to remove individual tasks or entire jobs.

fetchOutputs reports an error if the job is not in the ‘finished’ state, or if one of its tasks encountered an error during execution. If some tasks completed successfully, you can access their output arguments directly from the OutputArguments property of the tasks.

Examples

Create a job to generate a random matrix:

```matlab
myCluster = parcluster;  % Use default profile
j = createJob(myCluster,'Name','myjob');
t = createTask(j,@rand,1,[10]);
submit(j);
```
Wait for the job to finish and retrieve the random matrix:

```matlab
wait(j)
data = fetchOutputs(j);
data{1}
```

**Introduced in R2012a**
**fetchOutputs (FevalFuture)**

Retrieve all output arguments from Future

**Syntax**

```
[B1,B2,...,Bn] = fetchOutputs(F)
[B1,B2,...,Bn] = fetchOutputs(F,'UniformOutput',false)
```

**Description**

```
[B1,B2,...,Bn] = fetchOutputs(F) fetches all outputs of future object F after first
waiting for each element of F to reach the state 'finished'. An error results if any
element of F has NumOutputArguments less than the requested number of outputs.
```

When F is a vector of FevalFutures, each output argument is formed by concatenating the
corresponding output arguments from each future in F. An error results if these outputs
cannot be concatenated. To avoid this error, set the 'UniformOutput' option to false.

```
[B1,B2,...,Bn] = fetchOutputs(F,'UniformOutput',false) requests that
fetchOutputs combine the future outputs into cell arrays B1,B2,...,Bn. The outputs
of F can be of any size or type.
```

After the call to fetchOutputs, all futures in F have their 'Read' property set to true.
fetchOutputs returns outputs for all futures in F regardless of the value of each
future’s 'Read' property.

**Examples**

Create an FevalFuture, and fetch its outputs.

```
f = parfeval(@rand,1,3);
R = fetchOutputs(f)
```

```
0.5562    0.6218    0.3897
0.0084    0.4399    0.2700
0.0048    0.9658    0.8488
```
Create an FevalFuture vector, and fetch all its outputs.

```matlab
for idx = 1:10
    F(idx) = parfeval(@rand,1,1,10); % One row each future
end
R = fetchOutputs(F); % 10-by-10 concatenated output
```

**See Also**
fetchNext | parfeval | parpool

**Introduced in R2013b**
feval

Evaluate kernel on GPU

Syntax

feval(KERN, x1, ..., xn)
[y1, ..., ym] = feval(KERN, x1, ..., xn)

Description

feval(KERN, x1, ..., xn) evaluates the CUDA kernel KERN with the given arguments x1, ..., xn. The number of input arguments, n, must equal the value of the NumRHSArguments property of KERN, and their types must match the description in the ArgumentTypes property of KERN. The input data can be regular MATLAB data, GPU arrays, or a mixture of the two.

[y1, ..., ym] = feval(KERN, x1, ..., xn) returns multiple output arguments from the evaluation of the kernel. Each output argument corresponds to the value of the non-const pointer inputs to the CUDA kernel after it has executed. The output from feval running a kernel on the GPU is always gpuArray type, even if all the inputs are data from the MATLAB workspace. The number of output arguments, m, must not exceed the value of the MaxNumLHSArguments property of KERN.

Examples

If the CUDA kernel within a CU file has the following signature:

void myKernel(const float * pIn, float * pInOut1, float * pInOut2)

The corresponding kernel object in MATLAB then has the properties:

MaxNumLHSArguments: 2
NumRHSArguments: 3
ArgumentTypes: {'in single vector' ... 'inout single vector' 'inout single vector'}
You can use `feval` on this code’s kernel (KERN) with the syntax:

```
[y1, y2] = feval(KERN, x1, x2, x3)
```

The three input arguments, \( x_1, x_2, \) and \( x_3 \), correspond to the three arguments that are passed into the CUDA function. The output arguments, \( y_1 \) and \( y_2 \), are `gpuArray` types, and correspond to the values of \( p\text{In0ut1} \) and \( p\text{In0ut2} \) after the CUDA kernel has executed.

**See Also**

`arrayfun` | `gather` | `gpuArray` | `parallel.gpu.CUDAKernel`

**Introduced in R2010b**
findJob

Find job objects stored in cluster

Syntax

out = findJob(c)
[pending queued running completed] = findJob(c)
out = findJob(c,'p1',v1,'p2',v2,...)

Arguments

c Cluster object in which to find the job.
pending Array of jobs whose State is pending in cluster c.
queued Array of jobs whose State is queued in cluster c.
running Array of jobs whose State is running in cluster c.
completed Array of jobs that have completed running, i.e., whose State is finished or failed in cluster c.
out Array of jobs found in cluster c.
p1, p2 Job object properties to match.
v1, v2 Values for corresponding object properties.

Description

out = findJob(c) returns an array, out, of all job objects stored in the cluster c. Jobs in the array are ordered by the ID property of the jobs, indicating the sequence in which they were created.

[pending queued running completed] = findJob(c) returns arrays of all job objects stored in the cluster c, by state. Within pending, running, and completed, the jobs are returned in sequence of creation. Jobs in the array queued are in the order in which they are queued, with the job at queued(1) being the next to execute. The
completed jobs include those that failed. Jobs that are deleted or whose status is unavailable are not returned by this function.

\[
\text{out} = \text{findJob}(c,'p1',v1,'p2',v2,...)
\]

returns an array, \text{out}, of job objects whose property values match those passed as property-value pairs, \text{p1, v1, p2, v2}, etc. The property name must be a character vector, with the value being the appropriate type for that property. For a match, the object property value must be exactly the same as specified, including letter case. For example, if a job’s \text{Name} property value is \text{MyJob}, then \text{findJob} will not find that object while searching for a \text{Name} property value of \text{myjob}.

**See Also**

createJob | findTask | parcluster | recreate | submit

**Introduced before R2006a**
findTask

Task objects belonging to job object

Syntax

tasks = findTask(j)
tasks = findTask(j, taskFcn)
[pending running completed] = findTask(j)
tasks = findTask(j,'p1',v1,'p2',v2,...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>Job object.</td>
</tr>
<tr>
<td>tasks</td>
<td>Returned task objects.</td>
</tr>
<tr>
<td>pending</td>
<td>Array of tasks in job obj whose State is pending.</td>
</tr>
<tr>
<td>running</td>
<td>Array of tasks in job obj whose State is running.</td>
</tr>
<tr>
<td>completed</td>
<td>Array of completed tasks in job obj, i.e., those whose State is finished or failed.</td>
</tr>
<tr>
<td>p1, p2</td>
<td>Task object properties to match.</td>
</tr>
<tr>
<td>v1, v2</td>
<td>Values for corresponding object properties.</td>
</tr>
</tbody>
</table>

Description

tasks = findTask(j) gets a 1-by-N array of task objects belonging to a job object j. Tasks in the array are ordered by the ID property of the tasks, indicating the sequence in which they were created.

tasks = findTask(j, taskFcn) returns an array of task objects that belong to the job j, using taskFcn to select them. taskFcn is a function handle that accepts j.Tasks as an input argument, and returns a logical array indicating the tasks to return.
[pending running completed] = findTask(j) returns arrays of all task objects stored in the job object j, sorted by state. Within each array (pending, running, and completed), the tasks are returned in sequence of creation.

tasks = findTask(j,'p1',v1,'p2',v2,...) returns an array of task objects belonging to a job object j. The returned task objects will be only those matching the specified property-value pairs, p1, v1, p2, v2, etc. The property name must be a character vector, with the value being the appropriate type for that property. For a match, the object property value must be exactly the same as specified, including letter case. For example, if a task’s Name property value is MyTask, then findTask will not find that object while searching for a Name property value of mytask.

**Examples**

Create a job object.

c = parcluster();
j = createJob(c);

Add a task to the job object.

createTask(j,@rand,1,{10})

Find all task objects now part of job j.

t = findTask(j)

**Tips**

If job j 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.

**See Also**

createJob | createTask | findJob
Introduced before R2006a
for

for-loop over distributed range

Syntax

for variable = drange(colonop)
    statement
    ...
    statement
end

Description

The general format is

for variable = drange(colonop)
    statement
    ...
    statement
end

The colonop is an expression of the form start:increment:finish or start:finish. The default value of increment is 1. The colonop is partitioned by codistributed.colon into numlabs contiguous segments of nearly equal length. Each segment becomes the iterator for a conventional for-loop on an individual worker.

The most important property of the loop body is that each iteration must be independent of the other iterations. Logically, the iterations can be done in any order. No communication with other workers is allowed within the loop body. The functions that perform communication are gop, gcat, gplus, codistributor, codistributed, gather, and redistribute.

It is possible to access portions of codistributed arrays that are local to each worker, but it is not possible to access other portions of codistributed arrays.

The break statement can be used to terminate the loop prematurely.
Examples

Find the rank of magic squares. Access only the local portion of a codistributed array.

```matlab
r = zeros(1, 40, codistributor());
for n = drange(1:40)
    r(n) = rank(magic(n));
end
r = gather(r);
```

Perform Monte Carlo approximation of pi. Each worker is initialized to a different random number state.

```matlab
m = 10000;
for p = drange(1:numlabs)
    z = rand(m,1) + i*rand(m,1);
    c = sum(abs(z) < 1)
end
k = gplus(c)
p = 4*k/(m*numlabs);
```

Attempt to compute Fibonacci numbers. This will not work, because the loop bodies are dependent.

```matlab
f = zeros(1, 50, codistributor());
f(1) = 1;
f(2) = 2;
for n = drange(3:50)
    f(n) = f(n-1) + f(n-2)
end
```

See Also

for | numlabs | parfor

Introduced in R2007b
**gather**

Transfer distributed array or gpuArray to local workspace

**Syntax**

\[ X = \text{gather}(A) \]
\[ X = \text{gather}(C,\text{lab}) \]

**Description**

\( X = \text{gather}(A) \) can operate inside an \texttt{spmd} statement, \texttt{pmode}, or communicating job to gather together the elements of a codistributed array, or outside an \texttt{spmd} statement to gather the elements of a distributed array. If you execute this inside an \texttt{spmd} statement, \texttt{pmode}, or communicating job, \( X \) is a replicated array with all the elements of the array on every worker. If you execute this outside an \texttt{spmd} statement, \( X \) is an array in the local workspace, with the elements transferred from the multiple workers.

\[ X = \text{gather}(\text{distributed}(X)) \text{ or } X = \text{gather}(\text{codistributed}(X)) \text{ returns the original array } X. \]

\[ X = \text{gather}(C,\text{lab}) \text{ converts a codistributed array } C \text{ to a variant array } X, \text{ such that all of the elements are contained on worker } \text{lab}, \text{ and } X \text{ is a 0-by-0 empty double on all other workers.} \]

For a \texttt{gpuArray} input, \( X = \text{gather}(A) \) transfers the array elements from the GPU to the local workspace.

If the input argument to \texttt{gather} is not a distributed, a codistributed, or a \texttt{gpuArray}, the output is the same as the input.

**Examples**

Distribute a magic square across your workers, then gather the whole matrix onto every worker and then onto the client. This code results in the equivalent of \( M = \text{magic}(n) \) on all workers and the client.
n = 10;
spmd
    C = codistributed(magic(n));
    M = gather(C) % Gather all elements to all workers
end
S = gather(C)  % Gather elements to client

Gather all of the elements of C onto worker 1, for operations that cannot be performed across distributed arrays.

n = 10;
spmd
    C = codistributed(magic(n));
    out = gather(C,1);
    if labindex == 1
        % Characteristic sum for this magic square:
        characteristicSum = sum(1:n^2)/n;
        % Ensure that the diagonal sums are equal to the characteristic sum:
        areDiagonalsEqual = isequal ...
            (trace(out),trace(flipud(out)),characteristicSum)
    end
end

Lab 1:
    areDiagonalsEqual =
        1

Gather all of the elements from a distributed array D onto the client.

n = 10;
D = distributed(magic(n)); % Distribute array to workers
M = gather(D)              % Return array to client

Gather the results of a GPU operation to the MATLAB workspace.

G = gpuArray(rand(1024,1));
F = sqrt(G);  % Input and output are both gpuArray
W = gather(G); % Return array to workspace
whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1024x1</td>
<td>108</td>
<td>gpuArray</td>
</tr>
<tr>
<td>G</td>
<td>1024x1</td>
<td>108</td>
<td>gpuArray</td>
</tr>
<tr>
<td>W</td>
<td>1024x1</td>
<td>8192</td>
<td>double</td>
</tr>
</tbody>
</table>
Tips

Note that `gather` assembles the codistributed or distributed array in the workspaces of all the workers on which it executes, or on the MATLAB client, respectively, but not both. If you are using `gather` within an `spmd` statement, the gathered array is accessible on the client via its corresponding `Composite` object; see “Access Worker Variables with Composites” on page 3-8. If you are running `gather` in a communicating job, you can return the gathered array to the client as an output argument from the task.

As the `gather` function requires communication between all the workers, you cannot gather data from all the workers onto a single worker by placing the function inside a conditional statement such as `if labindex == 1`.

See Also
arrayfun | bsxfun | codistributed | distributed | gpuArray | pagefun | pmode

Introduced in R2006b
gcat

Global concatenation

Syntax

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

Description

Xs = gcat(X) concatenates the variant array X from each worker in the second dimension. The result is replicated on all workers.

Xs = gcat(X,dim) concatenates the variant array X from each worker in the dimension indicated by dim.

Xs = gcat(X,dim,targetlab) performs the reduction, and places the result into res only on the worker indicated by targetlab. res is set to [] on all other workers.

Examples

With four workers,

Xs = gcat(labindex)

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

See Also

cat | gop | labindex | numlabs

Introduced in R2006b
**gcp**

Get current parallel pool

**Syntax**

```matlab
p = gcp
p = gcp('nocreate')
```

**Description**

`p = gcp` returns a `parallel.Pool` object representing the current parallel pool. The current pool is where parallel language features execute, such as `parfor`, `spmd`, `distributed`, `Composite`, `parfeval` and `parfevalOnAll`.

If no parallel pool exists, `gcp` starts a new parallel pool and returns a pool object for that, unless automatic pool starts are disabled in your parallel preferences. If no parallel pool exists and automatic pool starts are disabled, `gcp` returns an empty pool object.

`p = gcp('nocreate')` returns the current pool if one exists. If no pool exists, the `'nocreate'` option causes `gcp` not to create a pool, regardless of your parallel preferences settings.

**Examples**

**Find Size of Current Pool**

Find the number of workers in the current parallel pool.

```matlab
p = gcp('nocreate'); % If no pool, do not create new one.
if isempty(p)
    poolsize = 0;
else
```
poolsize = p.NumWorkers
end

**Delete Current Pool**

Use the parallel pool object to delete the current pool.

delete(gcp('nocreate'))

**See Also**

Composite | delete | distributed | parfeval | parfevalOnAll | parfor | parpool | spmd

**Introduced in R2013b**
getAttachedFilesFolder

Folder into which AttachedFiles are written

Syntax

g = getAttachedFilesFolder

g = getAttachedFilesFolder(FileName)

Arguments

folder Character vector indicating location where files from job’s AttachedFiles property are placed

FileName Character vector specifying all or part of the attached file or folder name

Description

folder = getAttachedFilesFolder returns the path to the local folder into which AttachedFiles are written on the worker. This function returns an empty array if it is not called on a MATLAB worker.

g = getAttachedFilesFolder(FileName) returns the path name to the specified attached folder on the worker, or the folder containing the specified attached file. FileName can match either the full name of the attached file or folder, or on the ending part of the name. Multiple match results return a cell array.

If you have attached a folder, this does not match on file names within that folder.

Suppose you attach the folder 'C:\monday\tuesday\wednesday\thursday', which on the workers is stored in /tmp/MDCS/tp12345. The following table displays the results of various match attempts.
### Examples

Attach a folder to a parallel pool, then find its location on the worker to execute one of its files.

```matlab
myPool = parpool;
addAttachedFiles(myPool,'mydir');
spmd
    folder = getAttachedFilesFolder('mydir');
    oldFolder = cd(folder);  % Change to that folder
    [OK,output] = system('myExecutable');
    cd(oldFolder);            % Change to original folder
end
```

Attach an executable file to a parallel pool, then change to its folder for accessing and processing some data.

```matlab
myPool = parpool;
addAttachedFiles(myPool,'myExecutable');
spmd
    system('myExecutable');   % Now on MATLAB path
    folder = getAttachedFilesFolder('myExecutable');
    oldFolder = cd(folder);
    fid = open('myData.txt'); % Access data file
end
```
% Process fid
close(fid)
cd(oldFolder);
end

See Also

Functions
addAttachedFiles | getCurrentCluster | getCurrentJob | getCurrentTask | getCurrentWorker

Introduced in R2012a
getCodistributor

Codistributor object for existing codistributed array

Syntax

codist = get Codistributor(D)

Description

codist = get Codistributor(D) returns the codistributor object of codistributed array D. Properties of the object are Dimension and Partition for 1-D distribution; and BlockSize, LabGrid, and Orientation for 2-D block cyclic distribution. For any one codistributed array, get Codistributor returns the same values on all workers. The returned codistributor object is complete, and therefore suitable as an input argument for codistributed.build.

Examples

Get the codistributor object for a 1-D codistributed array that uses default distribution on 4 workers:

```matlab
spmd (4)
    I1 = eye(64,codistributor1d());
    codist1 = get Codistributor(I1)
    dim     = codist1.Dimension
    partn   = codist1.Partition
end
```

Get the codistributor object for a 2-D block cyclic codistributed array that uses default distribution on 4 workers:

```matlab
spmd (4)
    I2 = eye(128,codistributor2dbc());
    codist2 = get Codistributor(I2)
    blocksz = codist2.BlockSize
end
```
partn = codist2.LabGrid
ornt = codist2.Orientation
end

Demonstrate that these codistributor objects are complete:

spmd (4)
    isComplete(codist1)
    isComplete(codist2)
end

See Also
codistributed | codistributed.build | getLocalPart | redistribute

Introduced in R2009b
getCurrentCluster

Cluster object that submitted current task

Syntax

c = getCurrentCluster

Arguments

c The cluster object that scheduled the task currently being evaluated by the worker session.

Description

c = getCurrentCluster returns the parallel.Cluster object that has sent the task currently being evaluated by the worker session. Cluster object c is the Parent of the task’s parent job.

Examples

Find the current cluster.

myCluster = getCurrentCluster;

Get the host on which the cluster is running.

host = myCluster.Host;

Tips

If this function is executed in a MATLAB session that is not a worker, you get an empty result.
See Also
getAttachedFilesFolder | getCurrentJob | getCurrentTask | getCurrentWorker

Introduced in R2012a
**get_CurrentJob**

Job object whose task is currently being evaluated

**Syntax**

`job = getCurrentJob`

**Arguments**

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

**Description**

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

**Tips**

If the function is executed in a MATLAB session that is not a worker, you get an empty result.

**See Also**

`getAttachedFilesFolder` | `getCurrentCluster` | `getCurrentTask` | `getCurrentWorker`

*Introduced before R2006a*
**getCurrentTask**

Task object currently being evaluated in this worker session

**Syntax**

```matlab
task = getCurrentTask
```

**Arguments**

*task*  
The task object that the worker session is currently evaluating.

**Description**

`task = getCurrentTask` returns the `Parallel.Task` object whose function is currently being evaluated by the MATLAB worker session on the cluster.

**Tips**

If the function is executed in a MATLAB session that is not a worker, you get an empty result.

**See Also**

`getAttachedFilesFolder` | `getCurrentCluster` | `getCurrentJob` | `getCurrentWorker`

*Introduced before R2006a*
**getCurrentWorker**

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 `Parallel.Worker` object representing the MATLAB worker session that is currently evaluating the task function that contains this call.

If the function runs in a MATLAB session that is not a worker, it returns an empty result.

**Examples**

Find the Host property of a worker that runs a task. The file `identifyWorkerHost.m` contains the following function code.

```
function localHost = identifyWorkerHost()
    thisworker = getCurrentWorker; % Worker object
    localHost = thisworker.Host;   % Host property
end
```

Create a job with a task to execute this function on a worker and return the worker’s host name. This example manually attaches the necessary code file.
c = parcluster();
j = createJob(c);
j.AttachedFiles = {'identifyWorkerHost.m'};
t = createTask(j,@identifyWorkerHost,1,{});
submit(j)
wait(j)
workerhost = fetchOutputs(j)

See Also
getAttachedFilesFolder | getCurrentCluster | getCurrentJob | getCurrentTask

Introduced before R2006a
**getDebugLog**

Read output messages from job run in CJS cluster

**Syntax**

str = getDebugLog(cluster, job_or_task)

**Arguments**

- **str**
  - Variable to which messages are returned as a character vector expression.
- **cluster**
  - Cluster object referring to Microsoft Windows HPC Server (or CCS), Platform LSF, PBS Pro, or TORQUE cluster, created by parcluster.
- **job_or_task**
  - Object identifying job or task whose messages you want.

**Description**

str = getDebugLog(cluster, job_or_task) returns any output written to the standard output or standard error stream by the job or task identified by job_or_task, being run in the cluster identified by cluster.

**Examples**

This example shows how to create and submit a communicating job, and how to retrieve the job’s debug log. Assume that you already have a cluster profile called My3pCluster that defines the properties of the cluster.

```matlab
j = createCommunicatingJob(c);
createTask(j,@labindex,1,{});
```
submit(j);

getDebugLog(c,j);

**See Also**
createCommunicatingJob | createJob | createTask | parcluster

*Introduced before R2006a*
**getJobClusterData**

Get specific user data for job on generic cluster

**Syntax**

`userdata = getJobClusterData(cluster,job)`

**Arguments**

- `userdata`: Information that was previously stored for this job
- `cluster`: Cluster object identifying the generic third-party cluster running the job
- `job`: Job object identifying the job for which to retrieve data

**Description**

`userdata = getJobClusterData(cluster,job)` returns data stored for the job `job` that was derived from the generic cluster `cluster`. The information was originally stored with the function `setJobClusterData`. For example, it might be useful to store the third-party scheduler’s external ID for this job, so that the function specified in `GetJobStateFcn` can later query the scheduler about the state of the job.

For more information and examples on using these functions and properties, see “Integration Scripts for Generic Schedulers” on page 7-21.

**See Also**

- `setJobClusterData`

**Introduced in R2012a**
getJobFolder

Folder on client where jobs are stored

Syntax

joblocation = getJobFolder(cluster,job)

Description

joblocation = getJobFolder(cluster,job) returns the path to the folder on disk where files are stored for the specified job and cluster. This folder is valid only the client MATLAB session, not necessarily the workers. This method exists only on clusters using the generic interface.

See Also

getJobFolderOnCluster | parcluster

Introduced in R2012a
getJobFolderOnCluster

Folder on cluster where jobs are stored

Syntax

joblocation = getJobFolderOnCluster(cluster,job)

Description

joblocation = getJobFolderOnCluster(cluster,job) returns the path to the folder on disk where files are stored for the specified job and cluster. This folder is valid only in worker MATLAB sessions. An error results if the HasSharedFilesystem property of the cluster is false. This method exists only on clusters using the generic interface.

See Also

getJobFolder | parcluster

Introduced in R2012a
getLocalPart

Local portion of codistributed array

Syntax

L = getLocalPart(A)

Description

L = getLocalPart(A) returns the local portion of a codistributed array.

Examples

With four workers,

A = magic(4); %replicated on all workers
D = codistributed(A, codistributor1d(1));
L = getLocalPart(D)

returns

Lab 1: L = [16 2 3 13]
Lab 2: L = [5 11 10 8]
Lab 3: L = [9 7 6 12]
Lab 4: L = [4 14 15 1]

See Also

codistributed | codistributor

Introduced in R2009b
getLogLocation

Log location for job or task

Syntax

logfile = getLogLocation(cluster,cj)
logfile = getLogLocation(cluster,it)

Description

logfile = getLogLocation(cluster,cj) for a generic cluster cluster and communicating job cj, returns the location where the log data should be stored for the whole job cj.

logfile = getLogLocation(cluster,it) for a generic cluster cluster and task it of an independent job returns the location where the log data should be stored for the task it.

This function can be useful during submission, to instruct the third-party cluster to put worker output logs in the correct location.

See Also

parcluster

Introduced in R2012a
**globalIndices**

Global indices for local part of codistributed array

**Syntax**

```
K = globalIndices(C,dim)
K = globalIndices(C,dim,lab)
[E,F] = globalIndices(C,dim)
[E,F] = globalIndices(C,dim,lab)
K = globalIndices(codist,dim,lab)
[E,F] = globalIndices(codist,dim,lab)
```

**Description**

`globalIndices` tells you the relationship between indices on a local part and the corresponding index range in a given dimension on the codistributed array. The `globalIndices` method on a codistributor object allows you to get this relationship without actually creating the array.

`K = globalIndices(C,dim)` or `K = globalIndices(C,dim,lab)` returns a vector `K` so that `getLocalPart(C) = C(...,K,...)` in the specified dimension `dim` of codistributed array `C` on the specified worker. If the `lab` argument is omitted, the default is `labindex`.

`[E,F] = globalIndices(C,dim)` or `[E,F] = globalIndices(C,dim,lab)` returns two integers `E` and `F` so that `getLocalPart(C) = C(...,E:F,...)` of codistributed array `C` in the specified dimension `dim` on the specified worker. If the `lab` argument is omitted, the default is `labindex`.

`K = globalIndices(codist,dim,lab)` is the same as `K = globalIndices(C,dim,lab)`, where `codist` is the codistributor to be used for `C`, or `codist = getCodistributor(C)`. This allows you to get the global indices for a codistributed array without having to create the array itself.

`[E,F] = globalIndices(codist,dim,lab)` is the same as `[E,F] = globalIndices(C,dim,lab)`, where `codist` is the codistributor to be used for `C`, or
codist = getCodistributor(C). This allows you to get the global indices for a codistributed array without having to create the array itself.

Examples

Create a 2-by-22 codistributed array among four workers, and view the global indices on each lab:

```matlab
spmd
    C = zeros(2,22,codistributor1d(2,[6 6 5 5]));
    if labindex == 1
        K = globalIndices(C,2) % returns K = 1:6.
    elseif labindex == 2
        [E,F] = globalIndices(C,2) % returns E = 7, F = 12.
    end
    K = globalIndices(C,2,3) % returns K = 13:17.
    [E,F] = globalIndices(C,2,4) % returns E = 18, F = 22.
end
```

Use `globalIndices` to load data from a file and construct a codistributed array distributed along its columns, i.e., dimension 2. Notice how `globalIndices` makes the code not specific to the number of workers and alleviates you from calculating offsets or partitions.

```matlab
spmd
    siz = [1000,1000];
    codistr = codistributor1d(2,[],siz);

    % Use globalIndices to figure out which columns
    % each worker should load.
    [firstCol,lastCol] = globalIndices(codistr,2);

    % Call user-defined function readRectangleFromFile to
    % load all the values that should go into
    % the local part for this worker.
    labLocalPart = readRectangleFromFile(fileName, ... 
                    1,siz(1),firstCol,lastCol);

    % With the local part and codistributor,
    % construct the corresponding codistributed array.
    C = codistributed.build(labLocalPart,codistr);
end
```
See Also
getLocalPart | labindex

Introduced in R2008a
**gop**

Global operation across all workers

**Syntax**

\[
\text{res} = \text{gop}(\text{FUN}, \text{x})
\]

\[
\text{res} = \text{gop}(\text{FUN}, \text{x}, \text{targetlab})
\]

**Arguments**

- **FUN** Function to operate across workers.
- **x** Argument to function F, should be the same variable on all workers, but can have different values.
- **res** Variable to hold reduction result.
- **targetlab** Lab to which reduction results are returned. This value is returned by that worker’s `labindex`.

**Description**

\[ \text{res} = \text{gop}(\text{FUN}, x) \] is the reduction via the function \( \text{FUN} \) of the quantities \( x \) from each worker. The result is duplicated on all workers.

\( \text{FUN} \) can be a handle to any function, including user-written functions and user-defined anonymous functions. It should accept two arguments of the same type, and return one result of that same type, so it can be used iteratively in the form:

\[
\text{FUN}(\text{FUN}(x_1, x_2), \text{FUN}(x_3, x_4))
\]

The function \( \text{FUN} \) should be associative, that is,

\[
\text{FUN}(\text{FUN}(x_1, x_2), x_3) = \text{FUN}(x_1, \text{FUN}(x_2, x_3))
\]

\[ \text{res} = \text{gop}(\text{FUN}, x, \text{targetlab}) \] performs the reduction, and places the result into \( \text{res} \) only on the worker indicated by \( \text{targetlab} \). \( \text{res} \) is set to \( \textbf{[ ]} \) on all other workers.
Examples

This example shows how to calculate the sum and maximum values for x among all workers.

```matlab
p = parpool('local',4);
x = Composite();
x{1} = 3;
x{2} = 1;
x{3} = 4;
x{4} = 2;
spmd
    xsum = gop(@plus,x);
    xmax = gop(@max,x);
end
dxsum{1}
10
xmax{1}
4
```

This example shows how to horizontally concatenate the column vectors of x from all workers into a matrix. It uses the same 4-worker parallel pool opened by the previous example.

```matlab
x{1} = [3;30];
x{2} = [1;10];
x{3} = [4;40];
x{4} = [2;20];
spmd
    res = gop(@horzcat,x);
end
res{1}
```

```
3     1     4     2
30    10    40    20
```

This example shows how to use an anonymous function with `gop` to join character vectors with spaces between them. In this case, the character vectors are created from each worker's `labindex` value.

```matlab
afun = @(a,b)[a,' ',b]
spmd
```
res = gop(afun,num2str(labindex));
end
res{1}
1 2 3 4

See Also
labBarrier | labindex | numlabs

Topics
Using GOP to Achieve MPI_Allreduce Functionality

Introduced before R2006a
gplus

Global addition

Syntax

S = gplus(X)
S = gplus(X,targetlab)

Description

S = gplus(X) returns the addition of the variant array X from each worker. The result S is replicated on all workers.

S = gplus(X,targetlab) performs the addition, and places the result into S only on the worker indicated by targetlab. S is set to [] on all other workers.

Examples

With four workers,

S = gplus(labindex)

calculates S = 1 + 2 + 3 + 4, and returns 10 on all four workers.

See Also
gop | labindex

Introduced in R2006b
gpuArray
Create array on GPU

Syntax

G = gpuArray(X)

Description

G = gpuArray(X) copies the array X to the GPU, and returns a gpuArray object. To work with gpuArrays, use any GPU-enabled MATLAB function. For more information, see “Run MATLAB Functions on a GPU” on page 9-8. If you need to retrieve the array back from the GPU, for example when using a function that does not support GPU, use the gather function.

To avoid transferring data from the CPU, you can create arrays directly on the GPU. For more information, see “Create GPU Arrays Directly” on page 9-4.

If you need increased performance, or if a function is not available for GPU, gpuArrays support the following options:

1. To precompile and run purely element-wise code on gpuArrays, use the arrayfun function. For more information, see “Run Element-wise MATLAB Code on GPU” on page 9-18.
2. To run existing GPU kernels written in CUDA C++, use the MATLAB CUDAKernel interface. For more information, see “Run CUDA or PTX Code on GPU” on page 9-25.
3. To run C++ code containing CUDA device code or library calls, use a MEX-function. For more information, see “Run MEX-Functions Containing CUDA Code” on page 9-36.
4. To generate CUDA code from MATLAB code, use GPU Coder™. For more information, see “Getting Started with GPU Coder” (GPU Coder).

Examples
Use MATLAB Functions with a GPU

This example shows how to use GPU-enabled MATLAB functions to operate with gpuArrays. Check your GPU using the `gpuDevice` function.

Create a row vector. To transfer it to the GPU and create a gpuArray, use the `gpuArray` function.

```matlab
range = [-15:15 0 -15:15 0 -15:15];
n = gpuArray(range);
whos n
```

```
Name      Size            Bytes  Class       Attributes
n         1x95                4  gpuArray
```

To operate with gpuArrays, use any GPU-enabled MATLAB function. MATLAB automatically runs calculations on the GPU. For more information, see “Run MATLAB Functions on a GPU” on page 9-8. For example, use a combination of `diag`, `expm`, `mod`, `abs`, and `fliplr`.

```matlab
E = expm(diag(n,-1)) * expm(diag(n,1));
M = mod(abs(E),2);
F = M + fliplr(M);
```

Then, plot the results.

```matlab
imagesc(F);
colormap(flip(Gray));
```
If you need to transfer the data back from the GPU, use `gather`. Gathering back to the CPU can be costly, and is generally not necessary unless you need to use your result with functions that do not support `gpuArray`.

```matlab
result = gather(F);
whos result
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>96x96</td>
<td>73728</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
Perform a Monte Carlo Integration Using GPU-Enabled Functions

This example shows how to use MATLAB functions and operators with gpuArrays to compute the integral of a function, using the Monte Carlo integration method. Check your GPU using the `gpuDevice` function.

Define the number of points to sample. Sample points in the domain of the function, the interval [-1,1] in both x and y coordinates, by creating random points with the `rand` function. To create random arrays directly in the GPU, specify 'gpuArray' as the type of array argument. For more information, see “Establish Arrays on a GPU” on page 9-3.

\[ n = 10^6; \]
\[ x = 2 \cdot \text{rand}(n,1,'gpuArray') - 1; \]
\[ y = 2 \cdot \text{rand}(n,1,'gpuArray') - 1; \]

Define the function to integrate, and use the Monte Carlo integration formula on it. Because the code uses GPU-enabled functions and operators on gpuArrays, the computations automatically run on the GPU. To learn more about GPU-enabled functions, see “Run MATLAB Functions on a GPU” on page 9-8.

\[ f = x^2 + y^2 \leq 1; \]
\[ \text{result} = 4 \cdot 1/n \cdot f \cdot \text{ones}(n,1,'gpuArray') \]

\[ \text{result} = 3.1444 \]

Compute the Mandelbrot Set using GPU-Enabled Functions

This example shows how to use GPU-enabled MATLAB functions to compute a well-known mathematical construction: the Mandelbrot Set. Check your GPU using the `gpuDevice` function.

Define the parameters. The Mandelbrot algorithm iterates over a grid of real and imaginary parts. The following code defines the number of iterations, grid size and grid limits.

\[ \text{maxIterations} = 500; \]
\[ \text{gridSize} = 1000; \]
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862,  0.123640851045266];

You can use the `gpuArray` function to transfer data to the GPU and create a `gpuArray`, or you can create an array directly on the GPU. `gpuArray` provides GPU versions of many functions that you can use to create data arrays, such as `linspace`. For more information, see “Create GPU Arrays Directly” on page 9-4.

\[ x = \text{gpuArray.linspace}(\text{xlim}(1), \text{xlim}(2), \text{gridSize}); \]
\[ y = \text{gpuArray.linspace}(\text{ylim}(1), \text{ylim}(2), \text{gridSize}); \]

whos a b

Many MATLAB functions support `gpuArrays`. When you supply a `gpuArray` argument to any GPU-enabled function, the function runs automatically on the GPU. For more information, see “Run MATLAB Functions on a GPU” on page 9-8. Create a complex grid for the algorithm, and create an array for the results, `count`. To create this array directly on the GPU, use the `ones` function, and specify 'gpuArray'.

\[
[xGrid, yGrid] = \text{meshgrid}(x,y);
\]
\[ z0 = \text{complex}(xGrid, yGrid); \]
\[ \text{count} = \text{ones}(	ext{size}(z0), \text{"gpuArray"}); \]

The following code implements the Mandelbrot algorithm using GPU-enabled functions. Because the code uses `gpuArrays`, the calculations happen in the GPU.

\[ z = z0; \]
\[ \text{for} \ n = 0: \text{maxIterations} \]
\[ \quad z = z.*z + z0; \]
\[ \quad \text{inside} = \text{abs}(z) <= 2; \]
\[ \quad \text{count} = \text{count} + \text{inside}; \]
\[ \text{end} \]
\[ \text{count} = \text{log}(	ext{count}); \]

When computations are done, plot the results.

\[ \text{imagesc}(x,y,count) \]
\[ \text{colormap}([\text{jet()}; \text{flipud(jet())}; 0 0 0]); \]
\[ \text{axis off} \]
Input Arguments

X — Array
numeric array | logical array

Array, specified as a numeric or logical array, to transfer to the GPU. The GPU device must have sufficient free memory to store the data. If X is already a gpuArray, `gpuArray` outputs X unchanged.

Example: `G = gpuArray(rand(3));`
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

Output Arguments

**G — GPU array**

`gpuArray`

GPU array, returned as a `gpuArray` object, that represents an array stored in the GPU. Use `gpuArray` objects in hundreds of GPU-enabled MATLAB functions. For more information, see “Run MATLAB Functions on a GPU” on page 9-8. To retrieve the array back from the GPU, use the `gather` function.

The maximum number of elements of a `gpuArray` is limited to `intmax('int32')`.

Example: `G = gpuArray([1 2; 2 4]);`

Data Types: `gpuArray`

See Also

`arrayfun` | `bsxfun` | `existsOnGPU` | `feval` | `gather` | `gpuDevice` | `parallel.gpu.CUDAKernel` | `reset`

Topics

“Establish Arrays on a GPU” on page 9-3
“Run MATLAB Functions on a GPU” on page 9-8
“Identify and Select a GPU Device” on page 9-23

Introduced in R2010b
**gpuDevice**

Query or select GPU device

**Syntax**

D = gpuDevice
D = gpuDevice()
D = gpuDevice(IDX)
gpuDevice([])

**Description**

D = gpuDevice or D = gpuDevice(), if no device is already selected, selects the default GPU device and returns a GPUDevice object representing that device. If a GPU device is already selected, this returns an object representing that device without clearing it.

D = gpuDevice(IDX) selects the GPU device specified by index IDX. A warning or error might occur if the specified GPU device is not supported. This form of the command with a specified index resets the device and clears its memory (even if this device is already currently selected, equivalent to reset); so all workspace variables representing gpuArray or CUDAKernel variables are now invalid, and you should clear them from the workspace or redefine them.

gpuDevice([]), with an empty argument (as opposed to no argument), deselects the GPU device and clears its memory of gpuArray and CUDAKernel variables. This leaves no GPU device selected as the current device.

**Examples**

**Identify and Select a GPU**

This example shows how to identify and select a GPU for your computations.
To determine how many GPU devices are available in your computer, use the `gpuDeviceCount` function.

```matlab
gpuDeviceCount
2
```

When there are multiple devices, the first is the default. You can examine its properties with the `gpuDevice` function to determine if that is the one you want to use.

```matlab
d = gpuDevice
d =
CUDADevice with properties:

    Name: 'GeForce GTX 1080'
    Index: 1
    ComputeCapability: '6.1'
    SupportsDouble: 1
    ToolkitVersion: 9
    MaxThreadsPerBlock: 1024
    MaxShmemPerBlock: 49152
    MaxThreadBlockSize: [1024 1024 64]
    MaxGridSize: [2.1475e+09 65535 65535]
    SIMDWidth: 32
    TotalMemory: 8.5899e+09
    AvailableMemory: 7.0053e+09
    MultiprocessorCount: 20
    ClockRateKHz: 1733500
    ComputeMode: 'Default'
    GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
    CanMapHostMemory: 1
    DeviceSupported: 1
    DeviceSelected: 1
```

If this is the device you want to use, you can proceed. To run computations on the GPU, use `gpuArray` enabled functions. For more information, see “Run MATLAB Functions on a GPU” on page 9-8.

To use another device, call `gpuDevice` with the index of the other device.
gpuDevice(2)

**Query Compute Capabilities**

Create an object representing the default GPU device.

```matlab
g = gpuDevice;
```

Query the compute capabilities of all available GPU devices.

```matlab
for ii = 1:gpuDeviceCount
    g = gpuDevice(ii);
    fprintf(1,'Device %i has ComputeCapability %s \n', ...
            g.Index,g.ComputeCapability)
end
```

Device 1 has ComputeCapability 3.5
Device 2 has ComputeCapability 2.0

**Use Multiple GPUs in a Parallel Pool**

This example shows how to identify and select a GPU for the workers in a parallel pool. If you have access to several GPUs, you can perform your calculations on multiple GPUs in parallel using a parallel pool.

Start a parallel pool with as many workers as GPUs. To determine the number of GPUs available, use the `gpuDeviceCount` function. By default, MATLAB assigns a different GPU to each worker for best performance.

```matlab
parpool(gpuDeviceCount);
```

To identify which GPU each worker is using, call `gpuDevice` inside an `spmd` block. The `spmd` block runs `gpuDevice` on every worker.

```matlab
spmd
    gpuDevice
end
```

Use parallel language features, such as `parfor` or `parfeval`, to distribute your computations to workers in the parallel pool. If you use `gpuArray` enabled functions in your computations, then these functions run on the GPU of the worker. For more
information, see “Run MATLAB Functions on a GPU” on page 9-8. For an example, see “Run MATLAB Functions on Multiple GPUs”.

When you are done with your computations, shut down the parallel pool. You can use the gcp function to obtain the current parallel pool.

delete(gcp);

If you want to use a different choice of GPUs, then you can use gpuDevice to select a particular GPU on each worker. Define an array, for example gpuIndices, that contains the indices of the GPUs to activate on each worker. Then, start a parallel pool with as many workers as GPUs to select, and use an spmd block to run gpuDevice on each worker. The labindex function identifies each worker. Use this function to associate a worker with a GPU index.

gpuIndices = [1 3];
parpool(numel(gpuIndices));
spmd
gpuDevice(gpuIndices(labindex));
end

As a best practice, and for best performance, assign a different GPU to each worker.

**Input Argument**

**IDX — Index of the GPU device**

integer

Index of the GPU device, specified as an integer in the range 1 to gpuDeviceCount.

Example: gpuDevice(1);

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**See Also**

arrayfun | feval | gpuArray | gpuDeviceCount | parallel.gpu.CUDAKernel | reset | wait (GPUDevice)

**Topics**

“GPU Support by Release” on page 9-49
External Websites
Deep Learning with GPUs and MATLAB

Introduced in R2010b
gpuDeviceCount

Number of GPU devices present

Syntax

n = gpuDeviceCount

Description

n = gpuDeviceCount returns the number of GPU devices found in your computer.

Examples

Determine how many GPU devices you have available in your computer and examine the properties of each.

n = gpuDeviceCount;
for ii = 1:n
    gpuDevice(ii)
end

See Also
arrayfun | feval | gpuDevice | parallel.gpu.CUDAKernel

Topics
“GPU Support by Release” on page 9-49

Introduced in R2010b
gpurng

Control random number generation for GPU calculations

Syntax

gpurng(seed)
gpurng('shuffle')
gpurng(seed,generator)
gpurng('shuffle',generator)
gpurng('default')
S = gpurng
S = gpurng(___)  

Description

gpurng(seed) sets the starting point, or seed, of the random number generator used in GPU calculations, so that rand, randi, and randn produce predictable sequences of numbers.

gpurng('shuffle') sets the seed of the random number generator based on the current time so that rand, randi, and randn produce different sequences of numbers after each time you call gpurng.

gpurng(seed,generator) or gpurng('shuffle',generator) selects the type of random number generator used by rand, randi, and randn.

gpurng('default') returns the settings of the random number generator to their default values. The random numbers produced are the same as if you had restarted MATLAB. The default settings are the Combined Multiple Recursive generator with seed 0.

S = gpurng returns the current state of the random number generator as a structure with fields 'Type', 'Seed', and 'State'. Use this structure to restore the random number generator to the captured settings at a later time with gpurng(S).
gpurng(S) restores the state of the random number generator using settings previously captured with S = gpurng.

S = gpurng(___) returns the current state of the random number generator as a structure before changing the settings of the seed or generator type.

Examples

Create Predictable Arrays of Random Numbers on the GPU and CPU

Capture the GPU generator settings, and set the state of the CPU random number generator to match the GPU generator settings. Create predictable arrays of random numbers on the CPU and GPU.

Restore the generator type and seed to their default values on both the CPU and the GPU.

gpurng('default')
rng('default')

Save the default seed and generator type of the GPU random number generator.

GPUdef = gpurng

GPUdef = struct with fields:
   Type: 'combRecursive'
   Seed: 0
   State: [12x1 uint32]

Set the CPU random number generator to match the default GPU settings.

rng(GPUdef)

Create an array of uniformly distributed random numbers on the GPU.

rGPU = rand(1,10, 'gpuArray')

rGPU =

0.7270    0.4522    0.9387    0.2360    0.0277    0.1316    0.2112    0.9324    0.3184    0.3230

Create an array of random numbers on the CPU.
The seed and generator type are the same for both the GPU and the CPU, so the arrays are the same.

```matlab
isequal(rGPU, rCPU)
ans = logical
    1
```

The `gpurng` state does not save the settings for the transformation applied to generate a normally distributed set of random numbers. Even though the seed and the generator type are the same on the GPU and the CPU, the set of normally distributed random numbers is different.

```matlab
nGPU = randn(1,1000, 'gpuArray');
nCPU = randn(1,1000);
figure
hold on
histogram(nGPU)
histogram(nCPU)
legend('GPU', 'CPU')
title('Normally Distributed Random Numbers')
xlabel('Value')
ylabel('Count')
hold off
```
The statistics of the normal distribution of random numbers are the same on the GPU and the CPU.

By default, the CPU uses the 'Ziggurat' transformation, while the GPU uses the 'Inversion' algorithm for the 'CombRecursive' generator. You can set the transformation on the CPU to match that of the GPU using `RandStream` constructor.

**Input Arguments**

- **seed** — Random number seed
  - nonnegative integer
Random number seed, specified as a nonnegative integer. The seed specifies the starting point for the algorithm to generate random numbers. Specify the seed when you want reproducible results.

Example: `gprng(7)`

**generator — Random number generator**

character vector | string

Random number generator, specified as a character vector or string for any valid random number generator that supports multiple streams and substreams. Three random number generator algorithms are supported on the GPU.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Generator</th>
<th>Multiple Stream and Substream Support</th>
<th>Approximate Period In Full Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>'CombRecursive' or 'mrg32k3a'</td>
<td>Combined multiple recursive generator</td>
<td>Yes</td>
<td>$2^{191} (2^{63} \text{ streams of length } 2^{127})$</td>
</tr>
<tr>
<td>'Philox' or 'Philox4x32_10'</td>
<td>Philox 4x32 generator with 10 rounds</td>
<td>Yes</td>
<td>$2^{193} (2^{64} \text{ streams of length } 2^{129})$</td>
</tr>
<tr>
<td>'Threefry' or 'Threefry4x64_20'</td>
<td>Threefry 4x64 generator with 20 rounds</td>
<td>Yes</td>
<td>$2^{514} (2^{256} \text{ streams of length } 2^{258})$</td>
</tr>
</tbody>
</table>

For more information on the differences between generating random numbers on the GPU and CPU, see “Control Random Number Streams” on page 6-40.

Example: `gprng(0,'Philox')`

**S — Previous random number generator state**

structure

Previous random number generator state, specified as a structure previously created using `S = gprng`.

Example: `S = gprng` captures the current state of the random number generator, and `gprng(S)` restores the generator to those settings.

Data Types: `struct`
Output Arguments

S — Random number generator state
structure

Random number generator state, returned as a structure with the fields 'Type', 'Seed', and 'State'.

Example: S = gpurng captures the current state of the random number generator, and gpurng(S) restores the generator to those settings.

Data Types: struct

Compatibility Considerations

Default random number generator change for gpurng

Behavior change in future release

In a future release, the default random number generator for parallel computations will change to Threefry. This generator will offer performance enhancements for parallel calculations over the current default. In releases up to R2018b, the default random number generator for parallel computations is CombRecursive.

With a different default generator, MATLAB will generate different random numbers sequences by default in the context of parallel computations. However, statistics of these calculations will remain unaffected. Therefore, you might want to update any code that relies on the specific random numbers being generated, but most calculations on the random numbers will be unaffected.

To set the generator to the settings used by default in R2018b and earlier on GPU arrays, use the following command.

```
gpurng(0,"CombRecursive")
```

parallel.gpu.rng is renamed to gpurng

Behavior changed in R2018a

Starting in R2018a, the function parallel.gpu.rng is renamed to gpurng. Replace all instances of parallel.gpu.rng with gpurng.
parallel.gpu.rng will continue to work but is not recommended.

See Also
gpuArray | rng

Introduced in R2011b
gputimeit

Time required to run function on GPU

Syntax

t = gputimeit(F)
t = gputimeit(F,N)

Description

t = gputimeit(F) measures the typical time (in seconds) required to run the function specified by the function handle F. The function handle accepts no external input arguments, but can be defined with input arguments to its internal function call.

t = gputimeit(F,N) calls F to return N output arguments. By default, gputimeit calls the function F with one output argument, or no output arguments if F does not return any output.

Examples

Measure the time to calculate sum(A.' .* B, 1) on a GPU, where A is a 12000-by-400 matrix and B is 400-by-12000.

A = rand(12000,400,'gpuArray');
B = rand(400,12000,'gpuArray');
f = @() sum(A.' .* B, 1);
t = gputimeit(f)

0.0026

Compare the time to run svd on a GPU, with one versus three output arguments.

X = rand(1000,'gpuArray');
f = @( ) svd(X);
t3 = gputimeit(f,3)
1.0622

t1 = gputimeit(f,1)
0.2933

**Tips**

gputimeit is preferable to timeit for functions that use the GPU, because it ensures that all operations on the GPU have finished before recording the time and compensates for the overhead. For operations that do not use a GPU, timeit offers greater precision.

Note the following limitations:

- The function F should not call tic or toc.
- You cannot use tic and toc to measure the execution time of gputimeit itself.

**See Also**
gpuArray | wait

*Introduced in R2013b*
help

Help for toolbox functions in Command Window

Syntax

help class/function

Arguments

class A Parallel Computing Toolbox object class, for example, parallel.cluster, parallel.job, or parallel.task.

function A function or property of the specified class. To see what functions or properties are available for a class, see the methods or properties reference page.

Description

help class/function returns command-line help for the specified function of the given class.

If you do not know the class for the function, use class(obj), where function is of the same class as the object obj.

Examples

Get help on functions or properties from Parallel Computing Toolbox object classes.

help parallel.cluster/createJob
help parallel.job/cancel
help parallel.task/wait

c = parcluster();
j1 = createJob(c);
class(j1)

parallel.job.CJSIndependentJob

help parallel.job/createTask
help parallel.job/AdditionalPaths

See Also
methods

Introduced before R2006a
Inf

Array of infinity

Syntax

A = Inf(sz, arraytype)
A = Inf(sz, datatype, arraytype)

A = Inf(sz, 'like', P)
A = Inf(sz, datatype, 'like', P)

C = Inf(sz, codist)
C = Inf(sz, datatype, codist)
C = Inf(sz, ___, codist, 'noCommunication')
C = Inf(sz, ___, codist, 'like', P)

Description

A = Inf(sz, arraytype) creates a matrix with underlying class of double, with Inf values in all elements.

A = Inf(sz, datatype, arraytype) creates a matrix with underlying class of datatype, with Inf values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n,...,k or [m n ... k]</td>
<td>Specifies size as an m-by-n-by-...-by-k array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
<tr>
<td>Argument</td>
<td>Values</td>
<td>Descriptions</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>'codistributed'</td>
<td>Specifies codistributed array, using the default distribution scheme.</td>
<td></td>
</tr>
<tr>
<td>'gpuArray'</td>
<td>Specifies gpuArray.</td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>'double' (default), 'single' Specifies underlying class of the array, i.e., the data type of its elements.</td>
<td></td>
</tr>
</tbody>
</table>

$A = \text{Inf}(sz, 'like', P)$ creates an array of $\text{Inf}$ values with the same type and underlying class (data type) as array $P$.

$A = \text{Inf}(sz, datatype, 'like', P)$ creates an array of $\text{Inf}$ values with the specified underlying class ($datatype$), and the same type as array $P$.

$C = \text{Inf}(sz, \text{codist})$ or $C = \text{Inf}(sz, datatype, \text{codist})$ creates a codistributed array of $\text{Inf}$ values with the specified size and underlying class (the default $datatype$ is 'double'). The codistributor object $\text{codist}$ specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for $\text{codistributor1d}$ and $\text{codistributor2dbc}$. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```plaintext
spmd
    C = \text{Inf}(8, \text{codistributor1d}());
end
```

$C = \text{Inf}(sz, \_, \_, \text{codist}, 'noCommunication')$ specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

$C = \text{Inf}(sz, \_, \_, \text{codist}, 'like', P)$ creates a codistributed array of $\text{Inf}$ values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array $P$.

## Examples

### Create Distributed Inf Matrix

Create a 1000-by-1000 distributed array of $\text{Inf}$s with underlying class double:
D = Inf(1000,'distributed');

**Create Codistributed Inf Matrix**

Create a 1000-by-1000 codistributed double matrix of Inf\$s, distributed by its second dimension (columns).

```matlab
spmd(4)
    C = Inf(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed single matrix of Inf\$s, distributed by its columns.

```matlab
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = Inf(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

**Create gpuArray Inf Matrix**

Create a 1000-by-1000 gpuArray of Inf\$s with underlying class double:

```matlab
G = Inf(1000,'double','gpuArray');
```

**See Also**

Inf | NaN | eye | false | ones | true | zeros
isaUnderlying

True if distributed array's underlying elements are of specified class

Syntax

TF = isaUnderlying(D, 'classname')

Description

TF = isaUnderlying(D, 'classname') returns true if the elements of distributed or codistributed array D are either an instance of classname or an instance of a class derived from classname. isaUnderlying supports the same values for classname as the MATLAB isa function does.

Examples

N = 1000;
D_uint8  = ones(1,N,'uint8','distributed');
D_cell   = distributed.cell(1,N);
isUint8  = isaUnderlying(D_uint8,'uint8') % returns true
isDouble = isaUnderlying(D_cell,'double') % returns false

See Also

isa

Introduced in R2010a
iscodistributed

True for codistributed array

Syntax

tf = iscodistributed(X)

Description

tf = iscodistributed(X) returns true for a codistributed array, or false otherwise. For a description of codistributed arrays, see “Nondistributed Versus Distributed Arrays” on page 5-2.

Examples

With a running parallel pool,

```matlab
spmd
    L = ones(100,1);
    D = ones(100,1,'codistributed');
    iscodistributed(L) % returns false
    iscodistributed(D) % returns true
end
```

See Also

isdistributed

Introduced in R2009b
isComplete

True if codistributor object is complete

Syntax

tf = isComplete(codist)

Description

tf = isComplete(codist) returns true if codist is a completely defined codistributor, or false otherwise. For a description of codistributed arrays, see “Nondistributed Versus Distributed Arrays” on page 5-2.

See Also

codistributed | codistributor

Introduced in R2009b
isdistributed

True for distributed array

Syntax

tf = isdistributed(X)

Description

tf = isdistributed(X) returns true for a distributed array, or false otherwise. For a description of a distributed array, see “Nondistributed Versus Distributed Arrays” on page 5-2.

Examples

With a running parallel pool,

L = ones(100,1);
D = ones(100,1,'distributed');
isdistributed(L) % returns false
isdistributed(D) % returns true

See Also

iscodistributed

Introduced in R2006b
isequal

True if clusters have same property values

Syntax

isequal(C1,C2)
isequal(C1,C2,C3,...)

Description

isequal(C1,C2) returns logical 1 (true) if clusters C1 and C2 have the same property values, or logical 0 (false) otherwise.

isequal(C1,C2,C3,...) returns true if all clusters are equal. isequal can operate on arrays of clusters. In this case, the arrays are compared element by element.

When comparing clusters, isequal does not compare the contents of the clusters’ Jobs property.

Examples

Compare clusters after some properties are modified.

c1 = parcluster('local');
c1.NumWorkers = 2; % Modify cluster
c1.saveAsProfile('local2') % Create new profile
c2 = parcluster('local2'); % Make cluster from new profile
isequal(c1,c2)

1

c0 = parcluster('local') % Use original profile
isequal(c0,c1)

0
See Also
parcluster

Introduced in R2012a
isequal (FevalFuture)

True if futures have same ID

Syntax

eq = isequal(F1,F2)

Description

eq = isequal(F1,F2) returns logical 1 (true) if futures F1 and F2 have the same ID property value, or logical 0 (false) otherwise.

Examples

Compare future object in workspace to queued future object.

p = parpool('local',2);
q = p.FevalQueue;
Fp = parfevalOnAll(p,@pause,0,30);
F1 = parfeval(p,@magic,1,10);
F2 = q.QueuedFutures;
isequal(F1,F2)

1

See Also

cancel | fetchNext | fetchOutputs | parfeval | wait

Introduced in R2014a
**isreplicated**

True for replicated array

**Syntax**

\[
tf = \text{isreplicated}(X)
\]

**Description**

\[
tf = \text{isreplicated}(X)
\]
returns true for a replicated array, or false otherwise. For a description of a replicated array, see “Nondistributed Versus Distributed Arrays” on page 5-2. \text{isreplicated} also returns true for a Composite \(X\) if all its elements are identical.

**Examples**

With an open parallel pool,

\[
\text{spmd}
\begin{align*}
A &= \text{magic}(3); \\
t &= \text{isreplicated}(A) \quad \% \text{returns } t = \text{true} \\
B &= \text{magic(labindex)}; \\
f &= \text{isreplicated}(B) \quad \% \text{returns } f = \text{false}
\end{align*}
\text{end}
\]

**Tips**

\text{isreplicated}(X) requires checking for equality of the array \(X\) across all workers. This might require extensive communication and time. \text{isreplicated} is most useful for debugging or error checking small arrays. A codistributed array is not replicated.

**See Also**

iscodistributed | isdistributed
Introduced in R2006b
**jobStartup**

File for user-defined options to run when job starts

**Syntax**

```matlab
jobStartup(job)
```

**Arguments**

- **job**
  - The job for which this startup is being executed.

**Description**

`jobStartup(job)` runs automatically on a worker the first time that worker evaluates a task for a particular job. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the `jobStartup.m` file to define job initialization actions on the worker. The worker looks for `jobStartup.m` in the following order, executing the one it finds first:

1. Included in the job’s `AttachedFiles` property.
2. In a folder included in the job’s `AdditionalPaths` property.
3. In the worker’s MATLAB installation at the location
   
   ```matlab
   matlabroot/toolbox/distcomp/user/jobStartup.m
   ```

To create a version of `jobStartup.m` for `AttachedFiles` or `AdditionalPaths`, copy the provided file and modify it as required. For further details on `jobStartup` and its implementation, see the text in the installed `jobStartup.m` file.

**See Also**

- `poolStartup`
- `taskFinish`
- `taskStartup`
Introduced before R2006a
**labBarrier**

Block execution until all workers reach this call

**Syntax**

```
labBarrier
```

**Description**

`labBarrier` blocks execution of a parallel algorithm until all workers have reached the call to `labBarrier`. This is useful for coordinating access to shared resources such as file I/O.

**Examples**

**Synchronize Workers for Timing**

When timing code execution on the workers, use `labBarrier` to ensure all workers are synchronized and start their timed work together.

```
labBarrier;
tic
    A = rand(1,1e7,'codistributed');
distTime = toc;
```

**See Also**

`labBroadcast` | `labReceive` | `labSend` | `labSendReceive`

*Introduced before R2006a*
labBroadcast

Send data to all workers or receive data sent to all workers

Syntax

shared_data = labBroadcast(srcWkrIdx, data)
shared_data = labBroadcast(srcWkrIdx)

Arguments

srcWkrIdx
The labindex of the worker sending the broadcast.
data
The data being broadcast. This argument is required only for the worker that is broadcasting. The absence of this argument indicates that a worker is receiving.

shared_data
The broadcast data as it is received on all other workers.

Description

shared_data = labBroadcast(srcWkrIdx, data) sends the specified data to all executing workers. The data is broadcast from the worker with labindex == srcWkrIdx, and is received by all other workers.

shared_data = labBroadcast(srcWkrIdx) receives on each executing worker the specified shared_data that was sent from the worker whose labindex is srcWkrIdx.

If labindex is not srcWkrIdx, then you do not include the data argument. This indicates that the function is to receive data, not broadcast it. The received data, shared_data, is identical on all workers.

This function blocks execution until the worker’s involvement in the collective broadcast operation is complete. Because some workers may complete their call to labBroadcast before others have started, use labBarrier if you need to guarantee that all workers are at the same point in a program.
Examples

In this case, the broadcaster is the worker whose labindex is 1.

```matlab
srcWkrIdx = 1;
if labindex == srcWkrIdx
data = randn(10);
    shared_data = labBroadcast(srcWkrIdx,data);
else
    shared_data = labBroadcast(srcWkrIdx);
end
```

See Also

labBarrier | labSendReceive | labindex

Introduced before R2006a
**labindex**

Index of this worker

**Syntax**

\[ id = \text{labindex} \]

**Description**

\( id = \text{labindex} \) returns the index of the worker currently executing the function. \( \text{labindex} \) is assigned to each worker when a job begins execution, and applies only for the duration of that job. The value of \( \text{labindex} \) spans from 1 to \( n \), where \( n \) is the number of workers running the current job, defined by \text{numlabs}.

**Examples**

View \( \text{labindex} \) in \text{spmd} blocks and \text{parfor}-loops.

\[
p = \text{parpool('local',2)};
spmd
    labindex
end
\]

Lab 1:
    1

Lab 2:
    2

Using the same two-worker pool, \( p \):

\[
\text{parfor} \ a=1:4
    [a,labindex]
end
\]

\[
\text{ans} =
\begin{array}{cc}
3 & 1
\end{array}
\]
Tips

In an `spmd` block, because you have access to all workers individually and control what gets executed on them, each worker has a unique `labindex`.

However, inside a `parfor`-loop, `labindex` always returns a value of 1 on all workers in all iterations.

See Also

`labSendReceive` | `numlabs`

Introduced before R2006a
**labProbe**

Test to see if messages are ready to be received from other worker.

**Syntax**

```matlab
isDataAvail = labProbe
isDataAvail = labProbe(srcWkrIdx)
isDataAvail = labProbe('any',tag)
isDataAvail = labProbe(srcWkrIdx,tag)
[isDataAvail,srcWkrIdx,tag] = labProbe
```

**Arguments**

- **srcWkrIdx**
  - [labindex](#) of a particular worker from which to test for a message.

- **tag**
  - Tag defined by the sending worker's [labSend](#) function to identify particular data.

- **'any'**
  - Character vector to indicate that all workers should be tested for a message.

- **isDataAvail**
  - Logical indicating if a message is ready to be received.

**Description**

- `isDataAvail = labProbe` returns a logical value indicating whether any data is available for this worker to receive with the [labReceive](#) function.

- `isDataAvail = labProbe(srcWkrIdx)` tests for a message only from the specified worker.

- `isDataAvail = labProbe('any',tag)` tests only for a message with the specified tag, from any worker.
isDataAvail = labProbe(srcWkrIdx, tag) tests for a message from the specified worker and tag.

[isDataAvail, srcWkrIdx, tag] = labProbe returns labindex of the workers and tags of ready messages. If no data is available, srcWkrIdx and tag are returned as [].

**See Also**
labReceive | labSend | labSendReceive | labindex

**Introduced before R2006a**
labReceive

Receive data from another worker

Syntax

data = labReceive
data = labReceive(srcWkrIdx)
data = labReceive('any',tag)
data = labReceive(srcWkrIdx,tag)
[data,srcWkrIdx,tag] = labReceive

Arguments

srcWkrIdx  labindex of a particular worker from which to receive data.
tag  Tag defined by the sending worker’s labSend function to identify particular data.
'any'  Character vector to indicate that data can come from any worker.
data  Data sent by the sending worker’s labSend function.

Description

data = labReceive receives data from any worker with any tag.
data = labReceive(srcWkrIdx) receives data from the specified worker with any tag.
data = labReceive('any',tag) receives data from any worker with the specified tag.
data = labReceive(srcWkrIdx,tag) receives data from only the specified worker with the specified tag.
[data,srcWkrIdx,tag] = labReceive returns the source worker labindex and tag with the data.
Tips

This function blocks execution in the worker until the corresponding call to `labSend` occurs in the sending worker.

See Also

`labBarrier` | `labProbe` | `labSend` | `labSendReceive` | `labindex`

Introduced before R2006a
labSend
Send data to another worker

Syntax
labSend(data,rcvWkrIdx)
labSend(data,rcvWkrIdx,tag)

Arguments
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Data sent to the other workers; any MATLAB data type.</td>
</tr>
<tr>
<td>rcvWkrIdx</td>
<td>labindex of receiving worker or workers.</td>
</tr>
<tr>
<td>tag</td>
<td>Nonnegative integer to identify data.</td>
</tr>
</tbody>
</table>

Description
labSend(data,rcvWkrIdx) sends the data to the specified destination. data can be any MATLAB data type. rcvWkrIdx identifies the labindex of the receiving worker, and must be either a scalar or a vector of integers between 1 and numlabs; it cannot be labindex of the current (sending) worker.

labSend(data,rcvWkrIdx,tag) sends the data to the specified destination with the specified tag value. tag can be any integer from 0 to 32767, with a default of 0.

Tips
This function might or might not return before the corresponding labReceive completes in the receiving worker.

See Also
labBarrier | labProbe | labReceive | labSendReceive | labindex | numlabs
Introduced before R2006a
**labSendReceive**

Simultaneously send data to and receive data from another worker

**Syntax**

```
dataReceived = labSendReceive(rcvWkrIdx, srcWkrIdx, dataSent)
dataReceived = labSendReceive(rcvWkrIdx, srcWkrIdx, dataSent, tag)
```

**Arguments**

- **dataSent**: Data on the sending worker that is sent to the receiving worker; any MATLAB data type.
- **dataReceived**: Data accepted on the receiving worker.
- **rcvWkrIdx**: `labindex` of the receiving worker to which data is sent.
- **srcWkrIdx**: `labindex` of the source worker from which data is sent.
- **tag**: Nonnegative integer to identify data.

**Description**

```
dataReceived = labSendReceive(rcvWkrIdx, srcWkrIdx, dataSent) sends dataSent to the worker whose labindex is rcvWkrIdx, and receives dataReceived from the worker whose labindex is srcWkrIdx. The values for arguments rcvWkrIdx and srcWkrIdx must be scalars. This function is conceptually equivalent to the following sequence of calls:

labSend(dataSent, rcvWkrIdx);
dataReceived = labReceive(srcWkrIdx);
```

with the important exception that both the sending and receiving of data happens concurrently. This can eliminate deadlocks that might otherwise occur if the equivalent call to `labSend` would block.
If rcvWkrIdx is an empty array, labSendReceive does not send data, but only receives. If srcWkrIdx is an empty array, labSendReceive does not receive data, but only sends.

dataReceived = labSendReceive(rcvWkrIdx, srcWkrIdx, dataSent, tag) uses the specified tag for the communication. tag can be any integer from 0 to 32767.

**Examples**

Create a unique set of data on each worker, and transfer each worker’s data one worker to the right (to the next higher labindex).

First use the magic function to create a unique value for the variant array mydata on each worker:

\[
\text{mydata} = \text{magic(labindex)}
\]

**Lab 1:**

\[\begin{array}{c}
\text{mydata} = \\
1
\end{array}\]

**Lab 2:**

\[
\begin{array}{cccc}
1 & 3 \\
4 & 2
\end{array}
\]

**Lab 3:**

\[
\begin{array}{cccc}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{array}
\]

Define the worker on either side, so that each worker will receive data from the worker on its “left,” while sending data to the worker on its “right,” cycling data from the end worker back to the beginning worker.

\[
\text{rcvWkrIdx} = \text{mod(labindex, numlabs)} + 1; \quad \% \text{one worker to the right}
\]

\[
\text{srcWkrIdx} = \text{mod(labindex - 2, numlabs)} + 1; \quad \% \text{one worker to the left}
\]

Transfer the data, sending each worker’s mydata into the next worker’s otherdata variable, wrapping the third worker’s data back to the first worker.

\[
\text{otherdata} = \text{labSendReceive(rcvWkrIdx, srcWkrIdx, mydata)}
\]

**Lab 1:**

\[\begin{array}{c}
\text{otherdata} = \\
\end{array}\]

11-212
Transfer data to the next worker without wrapping data from the last worker to the first worker.

```matlab
if labindex < numlabs; rcvWkrIdx = labindex + 1; else rcvWkrIdx = []; end;
if labindex > 1; srcWkrIdx = labindex - 1; else srcWkrIdx = []; end;
otherdata = labSendReceive(rcvWkrIdx,srcWkrIdx,mydata)
```

See Also
`labBarrier` | `labProbe` | `labReceive` | `labSend` | `labindex` | `numlabs`

Introduced in R2006b
length

Length of object array

Syntax

length(obj)

Arguments

obj An object or an array of objects.

Description

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

Examples

Examine how many tasks are in the job j1.

length(j1.Tasks)

ans =
  9

See Also

size

Introduced before R2006a
**listAutoAttachedFiles**

List of files automatically attached to job, task, or parallel pool

**Syntax**

`listAutoAttachedFiles(obj)`

**Description**

`listAutoAttachedFiles(obj)` performs a dependency analysis on all the task functions, or on the batch job script or function. Then it displays a list of the code files that are already or going to be automatically attached to the job or task object `obj`.

If `obj` is a parallel pool, the output lists the files that have already been attached to the parallel pool following an earlier dependency analysis. The dependency analysis runs if a `parfor` or `spmd` block errors due to an undefined function. At that point any files, functions, or scripts needed by the `parfor` or `spmd` block are attached if possible.

**Examples**

**Automatically Attach Files via Cluster Profile**

Employ a cluster profile to automatically attach code files to a job. Set the `AutoAttachFiles` property for a job in the cluster’s profile. If this property value is true, then all jobs you create on that cluster with this profile will have the necessary code files automatically attached. This example assumes that the cluster profile `myAutoCluster` has that setting.

Create batch job, applying your cluster:

```matlab
obj = batch(myScript,'profile','myAutoCluster');
```

Verify attached files by viewing list.
Automatically Attach Files Programmatically

Programmatically set a job to automatically attach code files, and then view a list of those files for one of the tasks in the job.

c = parcluster();  % Use default profile
j = createJob(c);
j.AutoAttachFiles = true;
obj = createTask(j, myFun, OutNum, ArgCell);
listAutoAttachedFiles(obj)  % View attached list

The files returned in the output listing are those that analysis has determined to be required for the workers to evaluate the function myFun, and which automatically attach to the job.

Input Arguments

**obj — Job, task, or pool to which files automatically attach**  
job object | task object | parallel pool object

Job, task, or pool to which code files are automatically attached, specified as a parallel.Job, parallel.Task, or parallel.Pool object. The AutoAttachFiles property of the job object must be true; if the input is a task object, then this applies to its parent job object.

Example: obj = createJob(cluster);
Example: obj = gcp

See Also

batch | createCommunicatingJob | createJob | createTask | parcluster | parpool

Topics

“Add and Modify Cluster Profiles” on page 6-20
Introduced in R2013a
**load**

Load workspace variables from batch job

**Syntax**

\[
\begin{align*}
\text{load}(\text{job}) \\
\text{load}(\text{job},'X') \\
\text{load}(\text{job},'X','Y','Z*') \\
\text{load}(\text{job},'\text{-regexp}','\text{PAT1}','\text{PAT2}') \\
S &= \text{load}(\text{job} \ldots)
\end{align*}
\]

**Arguments**

- **job**
  Job from which to load workspace variables.
- **'X' , 'Y' , 'Z* '**
  Variables to load from the job. Wildcards allow pattern matching in MAT-file style.
- **'-regexp'**
  Indication to use regular expression pattern matching.
- **S**
  Struct containing the variables after loading.

**Description**

\text{load}(\text{job}) \text{ retrieves all variables from a batch job and assigns them into the current workspace.} \text{ load throws an error if the batch runs a function (instead of a script), the job is not finished, or the job encountered an error while running.}

\text{load}(\text{job},'X') \text{ loads only the variable named } X \text{ from the job.}

\text{load}(\text{job},'X','Y','Z*') \text{ loads only the specified variables. The wildcard ' * ' loads variables that match a pattern (MAT-file only).}

\text{load}(\text{job},'\text{-regexp}','\text{PAT1}','\text{PAT2}') \text{ can be used to load all variables matching the specified patterns using regular expressions. For more information on using regular expressions, type } \text{doc regexp} \text{ at the command prompt.}
S = \texttt{load(job ...)}\) returns the contents of \texttt{job} into variable \texttt{S}, which is a struct containing fields matching the variables retrieved.

**Examples**

Run a batch job and load its results into your client workspace.

```matlab
j = \texttt{batch('myScript')};
\texttt{wait(j)}
\texttt{load(j)}
```

Load only variables whose names start with 'a'.

```matlab
\texttt{load(job,'a*')}
```

Load only variables whose names contain any digits.

```matlab
\texttt{load(job,'-regexp','\d')}
```

**See Also**

\texttt{batch} | \texttt{fetchOutputs}

**Introduced in R2008a**
logout

Log out of MJS cluster

**Syntax**

`logout(c)`

**Description**

`logout(c)` logs you out of the MJS cluster specified by cluster object `c`. Any subsequent call to a privileged action requires you to re-authenticate with a valid password. Logging out might be useful when you are finished working on a shared machine.

**See Also**

changePassword

**Introduced in R2012a**
mapreducer

Define parallel execution environment for mapreduce and tall arrays

Syntax

mapreducer
mapreducer(0)
mapreducer(poolobj)
mapreducer(hadoopCluster)
mapreducer(mr)

mr = mapreducer(__)

mr = mapreducer(___,'ObjectVisibility','Off')

Description

mapreducer defines the execution environment for mapreduce or tall arrays. Use the mapreducer function to change the execution environment to use a different cluster or to switch between serial and parallel development.

The default execution environment uses either the local MATLAB session, or a parallel pool if you have Parallel Computing Toolbox. If you have Parallel Computing Toolbox installed, when you use the tall or mapreduce functions, MATLAB automatically starts a parallel pool of workers, unless you have changed the default preferences. By default, a parallel pool uses local workers, typically one worker for each core in your machine. If you turn off the Automatically create a parallel pool option, then you must explicitly start a pool if you want to use parallel resources. See “Specify Your Parallel Preferences” on page 6-12.

When working with tall arrays, use mapreducer to set the execution environment prior to creating the tall array. Tall arrays are bound to the current global execution environment when they are constructed. If you subsequently change the global execution environment, then the tall array is invalid, and you must recreate it.

Note In MATLAB, you do not need to specify configuration settings using mapreducer because mapreduce algorithms and tall array calculations automatically run in the local
MATLAB session only. If you also have Parallel Computing Toolbox, then you can use the additional `mapreducer` configuration options listed on this page for running in parallel. If you have MATLAB Compiler, then you can use separate `mapreducer` configuration options for running in deployed environments.

See: `mapreducer` in the MATLAB documentation, or `mapreducer` in the MATLAB Compiler documentation.

`mapreducer` with no input arguments creates a new `mapreducer` execution environment with all the defaults and sets this to be the current `mapreduce` or tall array execution environment. You can use `gcmr` to get the current `mapreducer` configuration.

- If you have default preferences (**Automatically create a parallel pool** is enabled), and you have not opened a parallel pool, then `mapreducer` opens a pool using the default cluster profile, sets `gcmr` to a `mapreducer` based on this pool and returns this `mapreducer`.
- If you have opened a parallel pool, then `mapreducer` sets `gcmr` to a `mapreducer` based on the current pool and returns this `mapreducer`.
- If you have disabled **Automatically create a parallel pool**, and you have not opened a parallel pool, then `mapreducer` sets `gcmr` to a `mapreducer` based on the local MATLAB session, and `mapreducer` returns this `mapreducer`.

`mapreducer(0)` specifies that `mapreduce` or tall array calculations run in the MATLAB client session without using any parallel resources.

`mapreducer(poolobj)` specifies a parallel pool for parallel execution of `mapreduce` or tall arrays. `poolobj` is a `parallel.Pool` object. The default pool is the current pool that is returned or opened by `gcp`.

`mapreducer(hadoopCluster)` specifies a Hadoop cluster for parallel execution of `mapreduce` or tall arrays. `hadoopCluster` is a `parallel.cluster.Hadoop` object.

`mapreducer(mr)` sets the global execution environment for `mapreduce` or tall arrays, using a previously created `MapReducer` object, `mr`, if its `ObjectVisibility` property is `'On'`.

`mr = mapreducer(____)` returns a `MapReducer` object to specify the execution environment. You can define several `MapReducer` objects, which enables you to swap execution environments by passing one as an input argument to `mapreduce` or `mapreducer`. 
mr = mapreducer(__,'ObjectVisibility','Off') hides the visibility of the MapReducer object, mr, using any of the previous syntaxes. Use this syntax to create new MapReducer objects without affecting the global execution environment of mapreduce.

**Examples**

**Develop in Serial and Then Use Local Workers or Cluster**

If you want to develop in serial and not use local workers or your specified cluster, enter:

mapreducer(0);

If you use mapreducer to change the execution environment after creating a tall array, then the tall array is invalid and you must recreate it. To use local workers or your specified cluster again, enter:

mapreducer(gcp);

**mapreducer with Automatically Create a Parallel Pool Switched Off**

If you have turned off the **Automatically create a parallel pool** option, then you must explicitly start a pool if you want to use parallel resources. See “Specify Your Parallel Preferences” on page 6-12 for details.

The following code shows how you can use mapreducer without input arguments to set the execution environment to your local MATLAB session and then specify a local parallel pool:

```matlab
>> mapreducer
>> parpool('local',1);
Starting parallel pool (parpool) using the 'local' profile ... connected to 1 workers.
>> gather(min(tall(rand(1000,1))))
Evaluating tall expression using the Local MATLAB Session:
Evaluation completed in 0 sec
ans =
```

11-223
5.2238e-04

**Input Arguments**

**poolobj — Pool for parallel execution**
gcp (default) | parallel.Pool object

Pool for parallel execution, specified as a parallel.Pool object.

Example: poolobj = gcp

**hadoopCluster — Hadoop cluster for parallel execution**
parallel.cluster.Hadoop object

Hadoop cluster for parallel execution, specified as a parallel.cluster.Hadoop object.

Example: hadoopCluster = parallel.cluster.Hadoop

**Output Arguments**

**mr — Execution environment for mapreduce and tall arrays**
MapReducer object

Execution environment for mapreduce and tall arrays, returned as a MapReducer object.

If the ObjectVisibility property of mr is set to 'On', then mr defines the default execution environment for all mapreduce algorithms and tall array calculations. If the ObjectVisibility property is 'Off', you can pass mr as an input argument to mapreduce to explicitly specify the execution environment for that particular call.

You can define several MapReducer objects, which enables you to swap execution environments by passing one as an input argument to mapreduce or mapreducer.

**Tips**

One of the benefits of developing your algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, then use mapreducer to scale up and take advantage of the capabilities offered by Parallel Computing Toolbox, MATLAB
Distributed Computing Server, or MATLAB Compiler, without needing to rewrite your algorithm.

See Also

See Also
gcmr | gcp | mapreduce | parallel.cluster.Hadoop | tall

Topics

“Big Data Workflow Using Tall Arrays and Datastores” on page 6-60
“Use Tall Arrays on a Parallel Pool” on page 6-63
“Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
“Run mapreduce on a Parallel Pool” on page 6-71
“Run mapreduce on a Hadoop Cluster” on page 6-75
“Specify Your Parallel Preferences” on page 6-12

Introduced in R2014b
methods

List functions of object class

**Syntax**

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

**Arguments**

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

**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 vectors.

**Examples**

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

```matlab
c = parcluster();
methods(c)

j1 = createJob(c);
methods(j1)

t1 = createTask(j1,@rand,1,{3});
methods(t1)
```
See Also

help

Introduced before R2006a
mexcuda

Compile MEX-function for GPU computation

Syntax

mexcuda filenames
mexcuda
mexcuda option1 ... optionN filenames

Description

mexcuda filenames compiles and links source files into a shared library called a MEX-file, executable from within MATLAB. mexcuda is an extension of the MATLAB mex function. It compiles MEX-files written using the CUDA C++ framework with NVIDIA's nvcc compiler, allowing the files to define and launch GPU kernels. Only a subset of Visual Studio compilers is supported for mexcuda. For details, consult the NVIDIA toolkit documentation. In addition, the mexcuda function exposes the GPU MEX API to allow the MEX-file to read and write gpuArrays.

mexcuda option1 ... optionN filenames builds with the specified build options. The option1 ... optionN arguments supplement or override the default mexcuda build configuration.

Examples

Compile Simple MEX-Function

Compile a simple MEX-function to create the function myMexFunction from a CUDA C++ source file.

mexcuda myMexFunction.cu
An example source file is available at `matlabroot/toolbox/distcomp/gpu/extern/src/mex/mexGPUExample.cu`.

**Display Detailed Build and Troubleshooting Information**

Use verbose mode to display the compile and link commands and other information useful for troubleshooting.

```
mexcuda -v myMexFunction.cu
```

**Compile and Link Multiple Source Files**

Compile and link multiple source files with one command.

```
mexcuda myMexFunction.cu otherSource1.cpp otherSource2.cpp
```

**Compile and Link in Two Stages**

First compile, then link to create a function.

```
mexcuda -c myMexFunction.cu
mexcuda myMexFunction.obj
```

The first line compiles to `myMexFunction.obj` (Windows) or `myMexFunction.o` (UNIX), and the second links to create the function `myMexFunction`.

**Compile with Dynamic Parallelism**

Compile code that uses dynamic parallelism, defining kernels that launch other kernels.

```
mexcuda -dynamic myMexFunction.cu
```
Link to Third-Party Library

Compile a MEX-function that makes use of the CUDA image primitives library, npp, which is installed at C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\lib\x64\nppi.lib.

mexcuda 'LC:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\lib\x64' -lnppi myMexFunction.cu

Input Arguments

filenames — One or more file names
character vector

One or more file names, including name and file extension, specified as a character vector. If the file is not in the current folder, specify the full path to the file. File names can be any combination of:

- C or C++ language source files
- object files
- library files

The first source code file listed in filenames is the name of the binary MEX-file. To override this naming convention, use the '-output' option.

Data Types: char

option1 ... optionN — One or more build options
character vectors corresponding to valid option flags

One or more build options, specified as one of these values. Options can appear in any order on any platform, except where indicated.

In addition to most options available for the mex function, these options are supported:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dynamic</td>
<td>Dynamic parallelism: compiles MEX-files that define kernels that launch other kernels.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-G</td>
<td>Generate debug information for device code. This makes it possible to step</td>
</tr>
<tr>
<td></td>
<td>through kernel code line by line in one of NVIDIA’s debugging applications</td>
</tr>
<tr>
<td></td>
<td>(NSight or cuda-gdb). To enable debugging of host code use -g.</td>
</tr>
</tbody>
</table>

All other `mex` function options are supported, except for the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>-compatibleArrayDims</td>
<td>Use of -largeArrayDims is implicit, and cannot be overridden.</td>
</tr>
</tbody>
</table>

**See Also**

`mex`

**Topics**

Accessing Advanced CUDA Features Using MEX

“Run MEX-Functions Containing CUDA Code” on page 9-36

**Introduced in R2015b**
mpiLibConf

Location of MPI implementation

Syntax

[primaryLib,extras] = mpiLibConf

Arguments

primaryLib
    MPI implementation library used by a communicating job.
extras
    Cell array of other required library names.

Description

[primaryLib,extras] = mpiLibConf returns the MPI implementation library to be used by a communicating job. primaryLib is the name of the shared library file containing the MPI entry points. extras is a cell array of other library names required by the MPI library.

To supply an alternative MPI implementation, create a file named mpiLibConf.m, and place it on the MATLAB path. The recommended location is matlabroot/toolbox/distcomp/user. Your mpiLibConf.m file must be higher on the cluster workers' path than matlabroot/toolbox/distcomp/mpi. (Sending mpiLibConf.m as a file dependency for this purpose does not work.) After your mpiLibConf.m file is in place, update the toolbox path caching with the following command in MATLAB:

rehash toolboxcache

Examples

Use the mpiLibConf function to view the current MPI implementation library:

mpiLibConf
mpich2.dll

**Tips**

Under all circumstances, the MPI library must support all MPI-1 functions. Additionally, the MPI library must support null arguments to `MPI_Init` as defined in section 4.2 of the MPI-2 standard. The library must also use an `mpi.h` header file that is fully compatible with MPICH2.

When used with the MATLAB job scheduler or the local cluster, the MPI library must support the following additional MPI-2 functions:

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

When used with any third-party scheduler, it is important to launch the workers using the version of `mpiexec` corresponding to the MPI library being used. Also, you might need to launch the corresponding process management daemons on the cluster before invoking `mpiexec`.

**See Also**

rehash

**Topics**

“Use Different MPI Builds on UNIX Systems” (MATLAB Distributed Computing Server)
“Toolbox Path Caching in MATLAB” (MATLAB)

**Introduced before R2006a**
**mpiprofile**

Profile parallel communication and execution times

**Syntax**

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

**Description**

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

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

`mpiprofile` takes the following options.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-detail mmex</td>
<td>This option specifies the set of functions for which profiling statistics are gathered. -detail mmex (the default) records information about functions, local functions, and MEX-functions. -detail builtin additionally records information about built-in functions such as eig or labReceive.</td>
</tr>
<tr>
<td>-message detail default</td>
<td>This option specifies the detail at which communication information is stored. -message detail default collects information on a per-lab instance.</td>
</tr>
<tr>
<td>-message detail simplified</td>
<td>-message detail simplified turns off collection for *PerLab data fields, which reduces the profiling overhead. If you have a very large cluster, you might want to use this option; however, you will not get all the detailed inter-lab communication plots in the viewer. For information about the structure of returned data, see mpiprofile info below.</td>
</tr>
<tr>
<td>-history</td>
<td>mpiprofile supports these options in the same way as the standard profile.</td>
</tr>
<tr>
<td>-nohistory</td>
<td>No other profile options are supported by mpiprofile. These three options have no effect on the data displayed by mpiprofile viewer.</td>
</tr>
<tr>
<td>-historysize &lt;size&gt;</td>
<td></td>
</tr>
</tbody>
</table>

mpiprofile off stops the parallel profiler. To reset the state of the profiler and disable collecting communication information, you should also call mpiprofile reset.

mpiprofile resume restarts the profiler without clearing previously recorded function statistics. This works only in pmode or in the same MATLAB worker session.

mpiprofile clear clears the profile information.
mpiprofile status returns a valid status when it runs on the worker.

mpiprofile reset turns off the parallel profiler and resets the data collection back to the standard profiler. If you do not call reset, subsequent profile commands will collect MPI information.

mpiprofile info returns a profiling data structure with additional fields to the one provided by the standard profile info in the FunctionTable entry. All these fields are recorded on a per-function and per-line basis, except for the *PerLab fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesSent</td>
<td>Records the quantity of data sent</td>
</tr>
<tr>
<td>BytesReceived</td>
<td>Records the quantity of data received</td>
</tr>
<tr>
<td>TimeWasted</td>
<td>Records communication waiting time</td>
</tr>
<tr>
<td>CommTime</td>
<td>Records the communication time</td>
</tr>
<tr>
<td>CommTimePerLab</td>
<td>Vector of communication receive time for each lab</td>
</tr>
<tr>
<td>TimeWastedPerLab</td>
<td>Vector of communication waiting time for each lab</td>
</tr>
<tr>
<td>BytesReceivedPerLab</td>
<td>Vector of data received from each lab</td>
</tr>
</tbody>
</table>

The three *PerLab fields are collected only on a per-function basis, and can be turned off by typing the following command in pmode:

mpiprofile on -messagedetail simplified

mpiprofile viewer is used in pmode after running user code with mpiprofile on. Calling the viewer stops the profiler and opens the graphical profile browser with parallel options. The output is an HTML report displayed in the profiler window. The file listing at the bottom of the function profile page shows several columns to the left of each line of code. In the summary page:

- Column 1 indicates the number of calls to that line.
- Column 2 indicates total time spent on the line in seconds.
- Columns 3-6 contain the communication information specific to the parallel profiler

mpiprofile('viewer', <profinfoarray>) in function form can be used from the client. A structure <profinfoarray> needs be passed in as the second argument, which is an array of mpiprofile info structures. See pInfoVector in the Examples section below.
mpiprofile does not accept -timer clock options, because the communication timer clock must be real.

For more information and examples on using the parallel profiler, see “Profiling Parallel Code” on page 6-45.

Examples

In pmode, turn on the parallel profiler, run your function in parallel, and call the viewer:

```matlab
mpiprofile on;
% call your function;
mpiprofile viewer;
```

If you want to obtain the profiler information from a communicating job outside of pmode (i.e., in the MATLAB client), you need to return output arguments of `mpiprofile info` by using the functional form of the command. Define your function `foo()`, and make it the task function in a communicating job:

```matlab
function [pInfo,yourResults] = foo
mpiprofile on
initData = (rand(100, codistributor()) ...
    * rand(100,codistributor()));
pInfo = mpiprofile('info');
yourResults = gather(initData,1)

After the job runs and `foo()` is evaluated on your cluster, get the data on the client:

```matlab
A = fetchOutputs(yourJob);
```

Then view parallel profile information:

```matlab
pInfoVector = [A{:,1}];
mpiprofile('viewer',pInfoVector);
```

See Also

`mpiSettings` | `pmode` | `profile`

Introduced in R2007b
mpiSettings

Configure options for MPI communication

Syntax

mpiSettings('DeadlockDetection','on')
mpiSettings('MessageLogging','on')
mpiSettings('MessageLoggingDestination','CommandWindow')
mpiSettings('MessageLoggingDestination','stdout')
mpiSettings('MessageLoggingDestination','File','filename')

Description

mpiSettings('DeadlockDetection','on') turns on deadlock detection during calls to labSend and labReceive. If deadlock is detected, a call to labReceive might cause an error. Although it is not necessary to enable deadlock detection on all workers, this is the most useful option. The default value is 'off' for communicating jobs, and 'on' inside pmode sessions or spmd statements. Once the setting has been changed within a pmode session or an spmd statement, the setting stays in effect until either the pmode session ends or the parallel pool is closed.

mpiSettings('MessageLogging','on') turns on MPI message logging. The default is 'off'. The default destination is the MATLAB Command Window.

mpiSettings('MessageLoggingDestination','CommandWindow') sends MPI logging information to the MATLAB Command Window. If the task within a communicating job is set to capture Command Window output, the MPI logging information will be present in the task's CommandWindowOutput property.

mpiSettings('MessageLoggingDestination','stdout') sends MPI logging information to the standard output for the MATLAB process. If you are using a MATLAB job scheduler (MJS), this is the mdce service log file.

mpiSettings('MessageLoggingDestination','File','filename') sends MPI logging information to the specified file.
Examples

Set deadlock detection for a communicating job inside the jobStartup.m file for that job:

```matlab
% Inside jobStartup.m for the communicating job
mpiSettings('DeadlockDetection','on');
myLogFname = sprintf('%s_%d.log',tempname,labindex);
mpiSettings('MessageLoggingDestination','File',myLogFname);
mpiSettings('MessageLogging','on');
```

Turn off deadlock detection for all subsequent spmd statements that use the same parallel pool:

```matlab
spmd;mpiSettings('DeadlockDetection','off');end
```

Tips

Setting the MessageLoggingDestination does not automatically enable message logging. A separate call is required to enable message logging.

`mpiSettings` has to be called on the worker, not the client. That is, it should be called within the task function, within `jobStartup.m`, or within `taskStartup.m`.

Introduced before R2006a
mxGPUCopyFromMxArray (C)

Copy mxArray to mxGPUArray

C Syntax

#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyFromMxArray(mxArray const * const mp)

Arguments

mp
   Pointer to an mxArray that contains either GPU or CPU data.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyFromMxArray produces a new mxGPUArray object with the same characteristics as the input mxArray.

- If the input mxArray contains a gpuArray, the output is a new copy of the data on the GPU.
- If the input mxArray contains numeric or logical CPU data, the output is copied to the GPU.

Either way, this function always allocates memory on the GPU and allocates a new mxGPUArray object on the CPU. Use mxGPUDestroyGPUArray to delete the result when you are done with it.
See Also
mxGPUCopyGPUArray | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCopyGPUArray (C)

Duplicate (deep copy) mxGPUArray object

C Syntax

```c
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyGPUArray(mxGPUArray const * const mgp)
```

Arguments

mgp

Pointer to an mxGPUArray.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyGPUArray produces a new array on the GPU and copies the data, and then returns a new mxGPUArray that refers to the copy. Use mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also

mxGPUCopyFromMxArray | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCopyImag (C)

Copy imaginary part of mxGPUArray

C Syntax

#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyImag(mxGPUArray const * const mgp)

Arguments

mgp

Pointer to an mxGPUArray. The target gpuArray must be full, not sparse.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyImag copies the imaginary part of GPU data, and returns a new mxGPUArray object that refers to the copy. The returned array is real, with element values equal to the imaginary values of the input, similar to how the MATLAB `imag` function behaves. If the input is real rather than complex, the function returns an array of zeros.

Use `mxGPUDestroyGPUArray` to delete the result when you are done with it.

See Also

mxGPUCopyReal | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCopyReal (C)

Copy real part of mxGPUArray

C Syntax

#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCopyReal(mxGPUArray const * const mgp)

Arguments

mgp

Pointer to an mxGPUArray. The target gpuArray must be full, not sparse.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCopyReal copies the real part of GPU data, and returns a new mxGPUArray object that refers to the copy. If the input is real rather than complex, the function returns a copy of the input.

Use mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also

mxGPUCopyImag | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCreateComplexGPUArray (C)

Create complex GPU array from two real gpuArrays

C Syntax

#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCreateComplexGPUArray(mxGPUArray const * const mgpR,
                                           mxGPUArray const * const mgpI)

Arguments

mgpRmgpI

Pointers to mxGPUArray data containing real and imaginary coefficients. The target
gpuArrays must be full, not sparse.

Returns

Pointer to an mxGPUArray.

Description

mxGPUCreateComplexGPUArray creates a new complex mxGPUArray from two real
mxGPUArray objects. The function allocates memory on the GPU and copies the data. The
inputs must both be real, and have matching sizes and classes. Use
mxGPUDestroyGPUArray to delete the result when you are done with it.

See Also

mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCreateFromMxArray (C)

Create read-only mxGPUArray object from input mxArray

C Syntax

#include "gpu/mxGPUArray.h"
mxGPUArray const * mxGPUCreateFromMxArray(mxArray const * const mp)

Arguments

mp
   Pointer to an mxArray that contains either GPU or CPU data.

Returns

Pointer to a read-only mxGPUArray object.

Description

mxGPUCreateFromMxArray produces a read-only mxGPUArray object from an mxArray.

- If the input mxArray contains a gpuArray, this function extracts a reference to the
  GPU data from an mxArray passed as an input to the function.
- If the input mxArray contains CPU data, the data is copied to the GPU, but the
  returned object is still read-only.

If you need a writable copy of the array, use mxGPUCopyFromMxArray instead.

This function allocates a new mxGPUArray object on the CPU. Use
mxGPUDestroyGPUArray to delete the result when you are done with it.
See Also
mxGPUCopyFromMxArray | mxGPUCreateGPUArray | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCreateGPUArray (C)

Create mxGPUArray object, allocating memory on GPU

C Syntax

```
#include "gpu/mxGPUArray.h"
mxGPUArray* mxGPUCreateGPUArray(mwSize const ndims,
    mwSize const * const dims,
    mxClassID const cid,
    mxComplexity const ccx,
    mxGPUIinitialize const init0)
```

Arguments

ndims
- mwSize type specifying the number of dimensions in the created mxGPUArray.
dims
- Pointer to an mwSize vector specifying the sizes of each dimension in the created mxGPUArray.
cid
- mxClassID type specifying the element class of the created mxGPUArray.
ccx
- mxComplexity type specifying the complexity of the created mxGPUArray.
init0
- mxGPUIinitialize type specifying whether to initialize elements values to 0 in the created mxGPUArray.

- A value of MX_GPU_INITIALIZE_VALUES specifies that elements are to be initialized to 0.
- A value of MX_GPU_DO_NOT_INITIALIZE specifies that elements are not to be initialized.
Returns

Pointer to an mxGPUArray.

Description

mxGPUCREATEGPUARRAY creates a new mxGPUArray object with the specified size, type, and complexity. It also allocates the required memory on the GPU, and initializes the memory if requested.

This function allocates a new mxGPUArray object on the CPU. Use mxGPUDestroyGPUArray to delete the object when you are done with it.

See Also

mxGPUCREATEFROMMARRAY | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCreateMxArrayOnCPU (C)

Create mxArray for returning CPU data to MATLAB with data from GPU

C Syntax

```
#include "gpu/mxGPUArray.h"
mxArray* mxGPUCreateMxArrayOnCPU(mxGPUArray const * const mgp)
```

Arguments

`mgp`
Pointer to an mxGPUArray.

Returns

Pointer to an mxArray object containing CPU data that is a copy of the GPU data.

Description

mxGPUCreateMxArrayOnCPU copies the GPU data from the specified mxGPUArray into an mxArray on the CPU for return to MATLAB. This is similar to the `gather` function. After calling this function, the input mxGPUArray object is no longer needed and you can delete it with mxGPUDestroyGPUArray.

See Also

mxGPUCreateMxArrayOnGPU | mxGPUDestroyGPUArray

Introduced in R2013a
mxGPUCreateMxArrayOnGPU (C)

Create mxArray for returning GPU data to MATLAB

C Syntax

```
#include "gpu/mxGPUArray.h"
mxArray* mxGPUCreateMxArrayOnGPU(mxGPUArray const * const mgp)
```

Arguments

`mgp`

Pointer to an `mxGPUArray`.

Returns

Pointer to an `mxArray` object containing GPU data.

Description

`mxGPUCreateMxArrayOnGPU` puts the `mxGPUArray` into an `mxArray` for return to MATLAB. The data remains on the GPU and the returned class in MATLAB is `gpuArray`. After this call, the `mxGPUArray` object is no longer needed and can be destroyed.

See Also

`mxGPUCreateMxArrayOnCPU` | `mxGPUDestroyGPUArray`

Introduced in R2013a
mxGPUDestroyGPUArray (C)

Delete mxGPUArray object

C Syntax

#include "gpu/mxGPUArray.h"
mxGPUDestroyGPUArray(mxGPUArray const * const mgp)

Arguments

mgp

    Pointer to an mxGPUArray.

Description

mxGPUDestroyGPUArray deletes an mxGPUArray object on the CPU. Use this function to delete an mxGPUArray object you created with:

- mxGPUCreateGPUArray
- mxGPUCreateFromMxArray
- mxGPUCopyFromMxArray
- mxGPUCopyReal
- mxGPUCopyImag, or
- mxGPUCreateComplexGPUArray.

This function clears memory on the GPU, unless some other mxArray holds a reference to the same data. For example, if the mxGPUArray was extracted from an input mxArray, or wrapped in an mxArray for an output, then the data remains on the GPU.
See Also
mxGPUCopyFromMxArray | mxGPUCopyImag | mxGPUCopyReal |
mxGPUCreateComplexGPUArray | mxGPUCreateFromMxArray |
mxGPUCreateGPUArray

Introduced in R2013a
mxGPUGetClassID (C)

mxClassID associated with data on GPU

C Syntax

```c
#include "gpu/mxGPUArray.h"
mxClassID mxGPUGetClassID(mxGPUArray const * const mgp)
```

Arguments

mgp

Pointer to an mxGPUArray.

Returns

mxClassID type.

Description

mxGPUGetClassID returns an mxClassID type indicating the underlying class of the input data.

See Also

mxGPUGetComplexity

Introduced in R2013a
mxGPUGetComplexity (C)

Complexity of data on GPU

C Syntax

#include "gpu/mxGPUArray.h"
mxComplexity mxGPUGetComplexity(mxGPUArray const * const mgp)

Arguments

mgp
    Pointer to an mxGPUArray.

Returns

mxComplexity type.

Description

mxGPUGetComplexity returns an mxComplexity type indicating the complexity of the GPU data.

See Also

mxGPUGetClassID

Introduced in R2013a
**mxGPUGetData (C)**

Raw pointer to underlying data

**C Syntax**

```c
#include "gpu/mxGPUArray.h"
void* mxGPUGetData(mxGPUArray const * const mgp)
```

**Arguments**

`mgp`

Pointer to an `mxGPUArray` on the GPU. The target `gpuArray` must be full, not sparse.

**Returns**

Pointer to data.

**Description**

`mxGPUGetData` returns a raw pointer to the underlying data. Cast this pointer to the type of data that you want to use on the device. It is your responsibility to check that the data inside the array has the appropriate type, for which you can use `mxGPUGetClassID`.

**See Also**

`mxGPUGetClassID`, `mxGPUGetDataReadOnly`

*Introduced in R2013a*
mxGPUGetDataReadOnly (C)

Read-only raw pointer to underlying data

C Syntax

```c
#include "gpu/mxGPUArray.h"
void const* mxGPUGetDataReadOnly(mxGPUArray const * const mgp)
```

Arguments

**mgp**

Pointer to an `mxGPUArray` on the GPU. The target gpuArray must be full, not sparse.

Returns

Read-only pointer to data.

Description

`mxGPUGetDataReadOnly` returns a read-only raw pointer to the underlying data. Cast it to the type of data that you want to use on the device. It is your responsibility to check that the data inside the array has the appropriate type, for which you can use `mxGPUGetClassID`.

See Also

`mxGPUGetClassID` | `mxGPUGetData`

Introduced in R2013a
mxGPUGetDimensions (C)

mxGPUArray dimensions

C Syntax

#include "gpu/mxGPUArray.h"
mwSize const * mxGPUGetDimensions(mxGPUArray const * const mgp)

Arguments

mgp

    Pointer to an mxGPUArray.

Returns

Pointer to a read-only array of mwSize type.

Description

mxGPUGetDimensions returns a pointer to an array of mwSize indicating the dimensions of the input argument. Use mxFree to delete the output.

See Also

mxGPUGetComplexity | mxGPUGetNumberOfDimensions

Introduced in R2013a
mxGPUGetNumberOfDimensions (C)

Size of dimension array for mxGPUArray

C Syntax

#include "gpu/mxGPUArray.h"
mwSize mxGPUGetNumberOfDimensions(mxGPUArray const * const mgp)

Arguments

mgp

  Pointer to an mxGPUArray.

Returns

mwSize type.

Description

mxGPUGetNumberOfDimensions returns the size of the dimension array for the mxGPUArray input argument, indicating the number of its dimensions.

See Also

mxGPUGetComplexity | mxGPUGetDimensions

Introduced in R2013a
**mxGPUGetNumberOfElements (C)**

Number of elements on GPU for array

**C Syntax**

```c
#include "gpu/mxGPUArray.h"
mwSize mxGPUGetNumberOfElements(mxGPUArray const * const mgp)
```

**Arguments**

*mgp*

Pointer to an `mxGPUArray`.

**Returns**

`mwSize` type.

**Description**

`mxGPUGetNumberOfElements` returns the total number of elements on the GPU for this array.

**See Also**

`mxGPUGetComplexity` | `mxGPUGetDimensions` | `mxGPUGetNumberOfDimensions`

*Introduced in R2013a*
mxGPUIsSame (C)
Determine if two mxGPUArrays refer to same GPU data

C Syntax
#include "gpu/mxGPUArray.h"
int mxGPUIsSame(mxGPUArray const * const mgp1,
    mxGPUArray const * const mgp2)

Arguments
mgp1mgp2
    Pointers to mxGPUArray.

Returns
int type.

Description
mxGPUIsSame returns an integer indicating if two mxGPUArray pointers refer to the same GPU data:
    • 1 (true) indicates that the inputs refer to the same data.
    • 0 (false) indicates that the inputs do not refer to the same data.

See Also
mxGPUIsValidGPUData

Introduced in R2013a
**mxGPUIsSparse (C)**

Determine if mxGPUArray contains sparse GPU data

**C Syntax**

```
#include "gpu/mxGPUArray.h"
int mxGPUIsSparse(mxGPUArray const * mp);
```

**Arguments**

*mp*

- Pointer to an `mxGPUArray` to be queried for sparse data.

**Returns**

Integer indicating true result:

- 1 indicates the input is a sparse `gpuArray`.
- 0 indicates the input is not a sparse `gpuArray`.

**See Also**

`mxGPUIsValidGPUData | mxIsGPUArray`

**Introduced in R2015a**
**mxGPUIsValidGPUData (C)**

Determine if mxArray is pointer to valid GPU data

**C Syntax**

```c
#include "gpu/mxGPUArray.h"
int mxGPUIsValidGPUData(mxArray const * const mp)
```

**Arguments**

- **mgp**
  - Pointer to an mxArray.

**Returns**

- int type.

**Description**

`mxGPUIsValidGPUData` indicates if the mxArray is a pointer to valid GPU data.

If the GPU device is reinitialized in MATLAB with `gpuDevice`, all data on the device becomes invalid, but the CPU data structures that refer to the GPU data still exist. This function checks whether the mxArray is a container of valid GPU data, and returns one of the following values:

- 0 (false) for CPU data or for invalid GPU data.
- 1 (true) for valid GPU data.

**See Also**

- `mxIsGPUArray`
Introduced in R2013a
**mxGPUSetDimensions (C)**

Modify number of dimensions and size of each dimension

**C Syntax**

```c
#include "gpu/mxGPUArray.h"
void mxGPUSetDimensions(mxGPUArray * const mgp, mwSize const * const dims, mwSize const ndims);
```

**Arguments**

- **mgp**
  Pointer to an mxGPUArray

- **dims**
  Dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to 5 and `dims[1]` to 7 establishes a 5-by-7 mxGPUArray.

  The `dims` array must not increase the overall size of the mxGPUArray. This array must contain at least `ndims` elements.

- **ndims**
  Number of dimensions.

**Description**

Call `mxGPUSetDimensions` to reshape an existing mxGPUArray. `mxGPUSetDimensions` does not reallocate memory.

**See Also**

`mxGPUGetDimensions (C)`
Introduced in R2018b
**mxInitGPU (C)**

Initialize MATLAB GPU library on currently selected device

**C Syntax**

```c
#include "gpu/mxGPUArray.h"
int mxInitGPU()
```

**Returns**

int type with one of the following values:

- **MX_GPU_SUCCESS** if the MATLAB GPU library is successfully initialized.
- **MX_GPU_FAILURE** if not successfully initialized.

**Description**

Before using any CUDA code in your MEX file, initialize the MATLAB GPU library if you intend to use any `mxGPUArray` functionality in MEX or any GPU calls in MATLAB. There are many ways to initialize the MATLAB GPU API, including:

- Call `mxInitGPU` at the beginning of your MEX file before any CUDA code.
- Call `gpuDevice(deviceIndex)` in MATLAB before running any MEX code.
- Create a `gpuArray` in MATLAB before running any MEX code.

You should call `mxInitGPU` at the beginning of your MEX file, unless you have an alternate way of guaranteeing that the MATLAB GPU library is initialized at the start of your MEX file.

If the library is initialized, this function returns without doing any work. If the library is not initialized, the function initializes the default device. Note: At present, a MATLAB MEX file can work with only one GPU device at a time.
See Also
gpuArray | gpuDevice

Introduced in R2013a
**mxIsGPUArray (C)**

Determine if mxArray contains GPU data

**C Syntax**

```c
#include "gpu/mxGPUArray.h"
int mxIsGPUArray(mxArray const * const mp);
```

**Arguments**

*mp*

Pointer to an mxArray that might contain gpuArray data.

**Returns**

Integer indicating true result:

- 1 indicates the input is a gpuArray.
- 0 indicates the input is not a gpuArray.

**See Also**

mxGPUIsSparse | mxGPUIsValidGPUData

**Introduced in R2013a**
**NaN**

Array of Not-a-Numbers

**Syntax**

\[
A = \text{NaN}(sz,\text{arraytype}) \\
A = \text{NaN}(sz,\text{datatype},\text{arraytype}) \\
A = \text{NaN}(sz,\text{like}',P) \\
A = \text{NaN}(sz,\text{datatype},\text{like}',P) \\
C = \text{NaN}(sz,\text{codist}) \\
C = \text{NaN}(sz,\text{datatype},\text{codist}) \\
C = \text{NaN}(sz,\text{like}',P) \\
C = \text{NaN}(sz,\text{like}',P)
\]

**Description**

\( A = \text{NaN}(sz,\text{arraytype}) \) creates a matrix with underlying class of double, with NaN values in all elements.

\( A = \text{NaN}(sz,\text{datatype},\text{arraytype}) \) creates a matrix with underlying class of datatype, with NaN values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n,...,k or [m n ... k]</td>
<td>Specifies size as an m-by-n-by-...-by-k array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
</tbody>
</table>
A = NaN(sz,'like',P) creates an array of NaN values with the same type and underlying class (data type) as array P.

A = NaN(sz,datatype,'like',P) creates an array of NaN values with the specified underlying class (datatype), and the same type as array P.

C = NaN(sz,codist) or C = NaN(sz,datatype,codist) creates a codistributed array of NaN values with the specified size and underlying class (the default datatype is 'double'). The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

spmd
    C = NaN(8,codistributor1d());
end

C = NaN(sz,___,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = NaN(sz,___,codist,'like',P) creates a codistributed array of NaN values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

Examples

Create Distributed NaN Matrix

Create a 1000-by-1000 distributed matrix of NaNs with underlying class double:
D = NaN(1000,'distributed');

**Create Codistributed NaN Matrix**

Create a 1000-by-1000 codistributed double matrix of NaNs, distributed by its second dimension (columns).

```matlab
spmd(4)
    C = NaN(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed single matrix of NaNs, distributed by its columns.

```matlab
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = NaN(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

**Create gpuArray NaN Matrix**

Create a 1000-by-1000 gpuArray of NaNs with underlying class double:

```matlab
G = NaN(1000,'double','gpuArray');
```

**See Also**

Inf | NaN | eye | false | ones | true | zeros

**Introduced in R2006b**
**numlabs**

Total number of workers operating in parallel on current job

**Syntax**

\[ n = \text{numlabs} \]

**Description**

\[ n = \text{numlabs} \] returns the total number of workers currently operating on the current job. This value is the maximum value that can be used with `labSend` and `labReceive`.

**Tips**

In an `spmd` block, `numlabs` on each worker returns the parallel pool size.

However, inside a `parfor`-loop, `numlabs` always returns a value of 1.

**See Also**

`labSendReceive` | `labindex`

**Introduced before R2006a**
ones

Array of ones

Syntax

N = ones(sz, arraytype)
N = ones(sz, datatype, arraytype)

N = ones(sz, 'like', P)
N = ones(sz, datatype, 'like', P)

C = ones(sz, codist)
C = ones(sz, datatype, codist)
C = ones(sz, ____ , codist, 'noCommunication')
C = ones(sz, ____ , codist, 'like', P)

Description

N = ones(sz, arraytype) creates a matrix with underlying class of double, with ones in all elements.

N = ones(sz, datatype, arraytype) creates a matrix with underlying class of datatype, with ones in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n,...,k or [m n ... k]</td>
<td>Specifies size as an m-by-n-by-...-by-k array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
</tbody>
</table>
### Argument | Values | Descriptions
--- | --- | ---
'codistributed' | Specifies codistributed array, using the default distribution scheme.
'gpuArray' | Specifies gpuArray.

datatype | 'double' (default), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64' | Specifies underlying class of the array, i.e., the data type of its elements.

\[ N = \text{ones}(sz, 'like', P) \] creates an array of ones with the same type and underlying class (data type) as array \( P \).

\[ N = \text{ones}(sz, \text{datatype}, 'like', P) \] creates an array of ones with the specified underlying class (\( \text{datatype} \)), and the same type as array \( P \).

\[ C = \text{ones}(sz, \text{codist}) \text{ or } C = \text{ones}(sz, \text{datatype}, \text{codist}) \] creates a codistributed array of ones with the specified size and underlying class (the default \( \text{datatype} \) is 'double'). The codistributor object \( \text{codist} \) specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```matlab
spmd
    C = ones(8, codistributor1d());
end
```

\[ C = \text{ones}(sz, \_, \_, \text{codist}, 'noCommunication') \] specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

\[ C = \text{ones}(sz, \_, \_, \text{codist}, 'like', P) \] creates a codistributed array of ones with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array \( P \).
Examples

Create Distributed Ones Matrix

Create a 1000-by-1000 distributed array of ones with underlying class double:

\[
D = \text{ones}(1000,'\text{distributed}');
\]

Create Codistributed Ones Matrix

Create a 1000-by-1000 codistributed double matrix of ones, distributed by its second dimension (columns).

\[
\text{spmd}(4) \\
C = \text{ones}(1000,'\text{codistributed}'); \\
\text{end}
\]

With four workers, each worker contains a 1000-by-250 local piece of \(C\).

Create a 1000-by-1000 codistributed uint16 matrix of ones, distributed by its columns.

\[
\text{spmd}(4) \\
\text{codist} = \text{codistributor('1d',2,100*[1:numlabs])}; \\
C = \text{ones}(1000,1000,'\text{uint16}',\text{codist}); \\
\text{end}
\]

Each worker contains a 100-by-\text{labindex} local piece of \(C\).

Create gpuArray Ones Matrix

Create a 1000-by-1000 gpuArray of ones with underlying class uint32:

\[
G = \text{ones}(1000,'\text{uint32}',\text{gpuArray}');
\]

See Also

Inf | NaN | eye | false | ones | true | zeros

Introduced in R2006b
pagefun

Apply function to each page of array on GPU

Syntax

A = pagefun(FUN,B)
A = pagefun(FUN,B,C,...)
[A,B,...] = pagefun(FUN,C,...)

Description

pagefun iterates over the pages of a gpuArray, applying the same function to each page.

A = pagefun(FUN,B) applies the function specified by FUN to each page of the
gpuArray B, and returns the results in gpuArray A, such that A(:,,:I,J,...) =
FUN(B(:,,:I,J,...)). FUN is a handle to a function that takes a two-dimensional input
argument.

You can use gather to retrieve the array from the GPU back to the MATLAB workspace.

A = pagefun(FUN,B,C,...) evaluates FUN using pages of the arrays B, C, etc., as
input arguments with scalar expansion enabled. Any of the input page dimensions that
are scalar are virtually replicated to match the size of the other arrays in that dimension
so that A(:,,:I,J,...) = FUN(B(:,,:I,J,...), C(:,,:I,J,...),...). At least
one of the inputs B, C, etc. must be a gpuArray. Any other inputs held in CPU memory are
converted to a gpuArray before calling the function on the GPU. If an array is to be used
in several different pagefun calls, it is more efficient to convert that array to a gpuArray
before your series of pagefun calls. The input pages B(:,,:I,J,...), C(:,,:I,J,...), etc., must satisfy all of the input and output requirements of FUN.

[A,B,...] = pagefun(FUN,C,...), where FUN is a handle to a function that returns
multiple outputs, returns gpuArrays A, B, etc., each corresponding to one of the output
arguments of FUN. pagefun invokes FUN with as many outputs as there are in the call to
pagefun. All elements of A must be the same class; B can be a different class from A, but
all elements of B must be of the same class; etc.
FUN must return values of the same class each time it is called. The order in which `pagefun` computes pages is not specified and should not be relied on.

FUN must be a handle to a function that is written in the MATLAB language (i.e., not a built-in function or a MEX-function).

Currently the supported values for FUN are:

- Most element-wise gpuArray functions, listed in “Run MATLAB Functions on a GPU” on page 9-8, and the following:
  - `@ctranspose`
  - `@fliplr`
  - `@flipud`
  - `@inv`
  - `@mldivide`
  - `@mrdivide`
  - `@mtimes`
  - `@rot90`
  - `@transpose`

**Examples**

```matlab
M = 3;  % output number of rows
K = 6;  % matrix multiply inner dimension
N = 2;  % output number of columns
P1 = 10; % size of first page dimension
P2 = 17; % size of second page dimension
P3 = 4;  % size of third page dimension
P4 = 12; % size of fourth page dimension
A = rand(M,K,P1,1,P3,'gpuArray');
B = rand(K,N,1,P2,P3,P4,'gpuArray');
C = pagefun(@mtimes,A,B);
s = size(C) % M-by-N-by-P1-by-P2-by-P3-by-P4
s =
    3    2   10   17     4    12
```

```matlab
M = 300; % output number of rows
K = 500; % matrix multiply inner dimension
```
N = 1000;      % output number of columns
P = 200;       % number of pages
A = rand(M,K,'gpuArray');
B = rand(K,N,P,'gpuArray');
C = pagefun(@mtimes,A,B);
s = size(C)    % returns M-by-N-by-P

s =
     300     1000      200

See Also
arrayfun | bsxfun | gather | gpuArray

Introduced in R2013b
parallel.cluster.Hadoop

Create Hadoop cluster object

Syntax

hadoopCluster = parallel.cluster.Hadoop
hadoopCluster = parallel.cluster.Hadoop(Name,Value)

Description


You use the resulting object as input to the mapreduce and mapreducer functions, for specifying the Hadoop cluster as the parallel execution environment for tall arrays and mapreduce.

hadoopCluster = parallel.cluster.Hadoop(Name,Value) uses the specified names and values to set properties on the created parallel.cluster.Hadoop object.

Examples

Set Hadoop Cluster as Execution Environment for mapreduce and mapreducer

This example shows how to create and use a parallel.cluster.Hadoop object to set a Hadoop cluster as the mapreduce parallel execution environment.

hadoopCluster = parallel.cluster.Hadoop('HadoopInstallFolder', '/host/hadoop-install');
mr = mapreducer(hadoopCluster);
Set Hadoop Cluster as Execution Environment for tall arrays

This example shows how to create and use a parallel.cluster.Hadoop object to set a Hadoop cluster as the tall array parallel execution environment.

```matlab
hadoopCluster = parallel.cluster.Hadoop(...
   'HadoopInstallFolder','/host/hadoop-install', ...
   'SparkInstallFolder','/host/spark-install');
mr = mapreducer(hadoopCluster);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'HadoopInstallFolder','/share/hadoop/a1.2.1'

ClusterMatlabRoot — Path to MATLAB for workers
character vector

Path to MATLAB for workers, specified as the comma-separated pair consisting of 'ClusterMatlabRoot' and a character vector. This points to the installation of MATLAB Distributed Computing Server for the workers, whether local to each machine or on a network share.

HadoopConfigurationFile — Path to Hadoop application configuration file
character vector

Path to Hadoop application configuration file, specified as the comma-separated pair consisting of 'HadoopConfigurationFile' and a character vector.

HadoopInstallFolder — Path to Hadoop installation on the local machine
character vector

Path to Hadoop installation on the local machine, specified as the comma-separated pair consisting of 'HadoopInstallFolder' and a character vector. If this property is not
set, the default is the value specified by the environment variable HADOOP_PREFIX, or if that is not set, then HADOOP_HOME.

**SparkInstallFolder — Path to Spark enabled Hadoop installation on worker machines**
character vector

Path to Spark enabled Hadoop installation on worker machines, specified as the comma-separated pair consisting of 'SparkInstallFolder' and a character vector. If this property is not set, the default is the value specified by the environment variable SPARK_PREFIX, or if that is not set, then SPARK_HOME.

**Output Arguments**

**hadoopCluster — Hadoop cluster**
parallel.cluster.Hadoop object

Hadoop cluster, returned as a parallel.cluster.Hadoop object.

**See Also**
mapreduce | mapreducer

**Topics**
“Use Tall Arrays on a Spark Enabled Hadoop Cluster” on page 6-67
“Run mapreduce on a Hadoop Cluster” on page 6-75
“Read and Analyze Hadoop Sequence File” (MATLAB)

**Introduced in R2014b**
parallel.clusterProfiles

Names of all available cluster profiles

Syntax

ALLPROFILES = parallel.clusterProfiles
[ALLPROFILES, DEFAULTPROFILE] = parallel.clusterProfiles

Description

ALLPROFILES = parallel.clusterProfiles returns a cell array containing the names of all available profiles.

[ALLPROFILES, DEFAULTPROFILE] = parallel.clusterProfiles returns a cell array containing the names of all available profiles, and separately the name of the default profile.

The cell array ALLPROFILES always contains a profile called local for the local cluster, and always contains the default profile. If the default profile has been deleted, or if it has never been set, parallel.clusterProfiles returns local as the default profile.

You can create and change profiles using the saveProfile or saveAsProfile methods on a cluster object. Also, you can create, delete, and change profiles through the Cluster Profile Manager.

Examples

Display the names of all the available profiles and set the first in the list to be the default profile.

allNames = parallel.clusterProfiles()
parallel.defaultClusterProfile(allNames{1});

Display the names of all the available profiles and get the cluster identified by the last profile name in the list.
allNames = parallel.clusterProfiles()
lastCluster = parcluster(allNames{end});

See Also
parallel.defaultClusterProfile | parallel.exportProfile | parallel.importProfile

Introduced in R2012a
parallel.defaultClusterProfile

Examine or set default cluster profile

Syntax

\[
p = \text{parallel.defaultClusterProfile} \\
\text{oldprofile} = \text{parallel.defaultClusterProfile}(\text{newprofile})
\]

Description

\( p = \text{parallel.defaultClusterProfile} \) returns the name of the default cluster profile.

\( \text{oldprofile} = \text{parallel.defaultClusterProfile}(\text{newprofile}) \) sets the default profile to be \text{newprofile} and returns the previous default profile. It might be useful to keep the old profile so that you can reset the default later.

If the default profile has been deleted, or if it has never been set, \text{parallel.defaultClusterProfile} returns 'local' as the default profile.

You can save modified profiles with the \text{saveProfile} or \text{saveAsProfile} method on a cluster object. You can create, delete, import, and modify profiles with the Cluster Profile Manager, accessible from the MATLAB desktop \text{Home} tab \text{Environment} area by selecting \text{Parallel > Create and Manage Clusters}.

Examples

Display the names of all available profiles and set the first in the list to be the default.

\[
\text{allProfiles} = \text{parallel.clusterProfiles} \\
\text{parallel.defaultClusterProfile(allProfiles\{1\})}
\]

First set the profile named 'MyProfile' to be the default, and then set the profile named 'Profile2' to be the default.
parallel.defaultClusterProfile('MyProfile');
oldDefault = parallel.defaultClusterProfile('Profile2');
strcmp(oldDefault,'MyProfile') % returns true

See Also
parallel.clusterProfiles | parallel.importProfile

Introduced in R2012a
**parallel.exportProfile**

Export one or more profiles to file

**Syntax**

```plaintext
parallel.exportProfile(profileName, filename)
parallel.exportProfile({profileName1, profileName2,...,profileNameN}, filename)
```

**Description**

`parallel.exportProfile(profileName, filename)` exports the profile with the name `profileName` to specified filename. The extension `.settings` is appended to the filename, unless already there.

`parallel.exportProfile({profileName1,profileName2,...,profileNameN}, filename)` exports the profiles with the specified names to filename.

To import a profile, use `parallel.importProfile` or the Cluster Profile Manager.

**Examples**

Export the profile named `MyProfile` to the file `MyExportedProfile.settings`.

```plaintext
parallel.exportProfile('MyProfile','MyExportedProfile')
```

Export the default profile to the file `MyDefaultProfile.settings`.

```plaintext
def_profile = parallel.defaultClusterProfile();
parallel.exportProfile(def_profile,'MyDefaultProfile')
```

Export all profiles except for `local` to the file `AllProfiles.settings`.

```plaintext
allProfiles = parallel.clusterProfiles();
% Remove 'local' from allProfiles
notLocal = ~strcmp(allProfiles,'local');
```
profilesToExport = allProfiles(notLocal);
if ~isempty(profilesToExport)
    parallel.exportProfile(profilesToExport,'AllProfiles');
end

**See Also**

parallel.clusterProfiles | parallel.importProfile

**Introduced in R2012a**
parallel.gpu.CUDAKernel

Create GPU CUDA kernel object from PTX and CU code

Syntax

KERN = parallel.gpu.CUDAKernel(PTXFILE, CPROTO)
KERN = parallel.gpu.CUDAKernel(PTXFILE, CPROTO, FUNC)
KERN = parallel.gpu.CUDAKernel(PTXFILE, CUFILE)
KERN = parallel.gpu.CUDAKernel(PTXFILE, CUFILE, FUNC)

Description

KERN = parallel.gpu.CUDAKernel(PTXFILE, CPROTO) and KERN = parallel.gpu.CUDAKernel(PTXFILE, CPROTO, FUNC) create a CUDAKernel object that you can use to call a CUDA kernel on the GPU. PTXFILE is the name of the file that contains the PTX code, or the contents of a PTX file as a character vector; and CPROTO is the C prototype for the kernel call that KERN represents. If specified, FUNC must be a character vector that unambiguously defines the appropriate kernel entry name in the PTX file. If FUNC is omitted, the PTX file must contain only a single entry point.

KERN = parallel.gpu.CUDAKernel(PTXFILE, CUFILE) and KERN = parallel.gpu.CUDAKernel(PTXFILE, CUFILE, FUNC) create a kernel object that you can use to call a CUDA kernel on the GPU. In addition, they read the CUDA source file CUFILE, and look for a kernel definition starting with '__global__' to find the function prototype for the CUDA kernel that is defined in PTXFILE.

For information on executing your kernel object, see “Run a CUDAKernel” on page 9-31.

Examples

If simpleEx.cu contains the following:

/*
 * Add a constant to a vector.
 */
__global__ void addToVector(float * pi, float c, int vecLen) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < vecLen) {
        pi[idx] += c;
    }
}

and simpleEx.ptx contains the PTX resulting from compiling simpleEx.cu into PTX, both of the following statements return a kernel object that you can use to call the addToVector CUDA kernel.

kern = parallel.gpu.CUDAKernel('simpleEx.ptx', ...
                      'simpleEx.cu');
kern = parallel.gpu.CUDAKernel('simpleEx.ptx', ...
                      'float *,float,int');

See Also
arrayfun | existsOnGPU | feval | gpuArray | reset

Introduced in R2010b
parallel.importProfile

Import cluster profiles from file

Syntax

prof = parallel.importProfile(filename)

Description

prof = parallel.importProfile(filename) imports the profiles stored in the specified file and returns the names of the imported profiles. If filename has no extension, .settings is assumed; configuration files must be specified with the .mat extension. Configuration .mat files contain only one profile, but profile .settings files can contain one or more profiles. If only one profile is defined in the file, then prof is a character vector reflecting the name of the profile; if multiple profiles are defined in the file, then prof is a cell array of character vectors. If a profile with the same name as an imported profile already exists, an extension is added to the name of the imported profile.

You can use the imported profile with any functions that support profiles. parallel.importProfile does not set any of the imported profiles as the default; you can set the default profile by using the parallel.defaultClusterProfile function.

Profiles that were exported in a previous release are upgraded during import. Configurations are automatically converted to cluster profiles.

Imported profiles are saved as a part of your MATLAB settings, so these profiles are available in subsequent MATLAB sessions without importing again.

Examples

Import a profile from file ProfileMaster.settings and set it as the default cluster profile.

profile_master = parallel.importProfile('ProfileMaster');
parallel.defaultClusterProfile(profile_master)
Import all the profiles from the file `ManyProfiles.settings`, and use the first one to open a parallel pool.

```matlab
profs = parallel.importProfile('ManyProfiles');
parpool(profs{1})
```

Import a configuration from the file `OldConfiguration.mat`, and set it as the default parallel profile.

```matlab
old_conf = parallel.importProfile('OldConfiguration.mat')
parallel.defaultClusterProfile(old_conf)
```

See Also

- `parallel.clusterProfiles`
- `parallel.defaultClusterProfile`
- `parallel.exportProfile`

Introduced in R2012a
parallel.pool.Constant

Build parallel.pool.Constant from data or function handle

Syntax

C = parallel.pool.Constant(X)
C = parallel.pool.Constant(FH)
C = parallel.pool.Constant(FH,CLEANUP)
C = parallel.pool.Constant(COMP)

Description

C = parallel.pool.Constant(X) copies the value X to each worker and returns a parallel.pool.Constant object, C, which allows each worker to access the value X within a parallel language construct (parfor, spmd, parfeval) using the property C.Value. This can improve performance when you have multiple parfor-loops accessing the same constant set of data, because X is transferred only once to the workers.

C = parallel.pool.Constant(FH) evaluates function handle FH on each worker and stores the result in C.Value. This is also useful for creating and using any handle-type resources on the workers, such as file handles and database connections.

C = parallel.pool.Constant(FH,CLEANUP) evaluates function handle FH on each worker and stores the result in C.Value. When C is cleared, the function handle CLEANUP is evaluated with a single argument C.Value on each worker.

C = parallel.pool.Constant(COMP) uses the values stored in the Composite COMP, and stores them in C.Value on each worker. This is especially useful when the data that you need to use inside a parfor-loop can be constructed only on the workers, such as when the data is too large to conveniently fit in the client, or when it is being loaded from a file that only the workers can access. If COMP does not have a defined value on every worker, an error results.
**Tips**

`parallel.pool.Constant` must be called in the MATLAB client session.

`parallel.pool.Constant` automatically starts a parallel pool, if possible. If no pool is running, and one cannot be started automatically, `parallel.pool.Constant` generates an error.

**Examples**

**Make `parallel.pool.Constant` from array in client**

This example shows how to create a numeric `parallel.pool.Constant`, and use it in multiple `parfor`-loops on the same pool.

First, create some large data on the client, then build a `parallel.pool.Constant`, transferring the data to the pool only once.

```matlab
data = rand(1000);
c = parallel.pool.Constant(data);
for ii = 1:10
    % Run multiple PARFOR loops accessing the data.
    parfor jj = 1:10
        x(ii,jj) = c.Value(ii,jj);
    end
end
```

**Make `parallel.pool.Constant` from function handle**

This example shows how to create a `parallel.pool.Constant` with a function handle and a cleanup function.

Create a temporary file on each worker. By passing `@fclose` as the second argument, the file is automatically closed when `c` goes out of scope.

```matlab
c = parallel.pool.Constant(@() fopen(tempname(pwd),'wt'),@fclose);
spmd
    disp(fopen(c.Value)); % Displays the temporary filenames.
end
```
parfor idx = 1:1000
    fprintf(c.Value,'Iteration: %d\n',idx);
end
clear c; % Closes the temporary files.

Make parallel.pool.Constant from Composite

This example shows how to build large data sets as a Composite on pool workers inside an spmd block, and then use that data as a parallel.pool.Constant inside a parfor-loop.

spmd
    if labindex == 1
        x = labBroadcast(1,rand(5000));
    else
        x = labBroadcast(1);
    end
end
xc = parallel.pool.Constant(x);
parfor idx = 1:10
    s(idx) = sum(xc.Value(:,idx));
end
s

s =
    1.0e+03 *
    2.5108   2.5031   2.5123   2.4909   2.4957   2.5462   2.4859   2.5320   2.5076   2.5432

See Also
parcluster|parfeval|parfor|parpool|spmd

Introduced in R2015b
parcluster

Create cluster object

Syntax

c = parcluster

c = parcluster(profile)

Description

c = parcluster returns a cluster object representing the cluster identified by the default cluster profile, with the cluster object properties set to the values defined in that profile. Use a cluster object in functions such as parpool or batch.

c = parcluster(profile) returns a cluster object representing the cluster identified by the specified cluster profile, with the cluster object properties set to the values defined in that profile.

You can save modified profiles with the saveProfile or saveAsProfile method on a cluster object. You can create, delete, import, and modify profiles with the Cluster Profile Manager, accessible from the MATLAB desktop Home tab Environment area by selecting Parallel > Create and Manage Clusters. For more information, see “Discover Clusters and Use Cluster Profiles” on page 6-15.

Examples

Find the cluster identified by the default parallel computing cluster profile, with the cluster object properties set to the values defined in that profile.

myCluster = parcluster;

View the name of the default profile and find the cluster identified by it. Open a parallel pool on the cluster.
defaultProfile = parallel.defaultClusterProfile
myCluster = parcluster(defaultProfile);
parpool(myCluster);

Find a particular cluster using the profile named 'MyProfile', and create an independent job on the cluster.

myCluster = parcluster('MyProfile');
j = createJob(myCluster);

See Also
createJob | parallel.clusterProfiles | parallel.defaultClusterProfile | parpool

Introduced in R2012a
parfeval

Execute function asynchronously on parallel pool worker

Syntax

\[ F = \text{parfeval}(p, \text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \]
\[ F = \text{parfeval}(\text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \]

Description

\[ F = \text{parfeval}(p, \text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \] requests asynchronous execution of the function \( \text{fcn} \) on a worker contained in the parallel pool \( p \), expecting \( \text{numout} \) output arguments and supplying as input arguments \( \text{in1}, \text{in2}, \ldots \). The asynchronous evaluation of \( \text{fcn} \) does not block MATLAB. \( F \) is a parallel.FevalFuture object, from which the results can be obtained when the worker has completed evaluating \( \text{fcn} \). The evaluation of \( \text{fcn} \) always proceeds unless you explicitly cancel execution by calling \( \text{cancel}(F) \). To request multiple function evaluations, you must call \( \text{parfeval} \) multiple times. (However, \( \text{parfevalOnAll} \) can run the same function on all workers.)

\[ F = \text{parfeval}(\text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \] requests asynchronous execution on the current parallel pool. If no pool exists, it starts a new parallel pool, unless your parallel preferences disable automatic creation of pools.

Examples

Submit a single request to the parallel pool and retrieve the outputs.

\begin{verbatim}
p = gcp(); \% Get the current parallel pool
f = parfeval(p,@magic,1,10);
value = fetchOutputs(f); \% Block until f completes
\end{verbatim}

Submit a vector of multiple future requests in a \texttt{for}-loop and retrieve the individual future outputs as they become available.

\begin{verbatim}
p = gcp();
\% To request multiple evaluations, use a loop.
\end{verbatim}
for idx = 1:10
    f(idx) = parfeval(p, @magic, 1, idx); % Square size determined by idx
end
% Collect the results as they become available.
magicResults = cell(1, 10);
for idx = 1:10
    % fetchNext blocks until next results are available.
    [completedIdx, value] = fetchNext(f);
    magicResults{completedIdx} = value;
    fprintf('Got result with index: %d.
', completedIdx);
end

See Also
afterAll | afterEach | cancel | fetchNext | fetchOutputs | isequal |
parallel.FevalFuture | parallel.pool.Constant | parfevalOnAll | parpool |
ticBytes | tocBytes | wait

Introduced in R2013b
parfevalOnAll

Execute function asynchronously on all workers in parallel pool

Syntax

\[
F = \text{parfevalOnAll}(p, \text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots)
\]
\[
F = \text{parfevalOnAll}(\text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots)
\]

Description

\( F = \text{parfevalOnAll}(p, \text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \) requests the asynchronous execution of the function \( \text{fcn} \) on all workers in the parallel pool \( p \). \text{parfevalOnAll} evaluates \( \text{fcn} \) on each worker with input arguments \( \text{in1}, \text{in2}, \ldots \), and expects \( \text{numout} \) output arguments. \( F \) is a parallel.FevalOnAllFuture object, from which you can obtain the results when all workers have completed executing \( \text{fcn} \).

\( F = \text{parfevalOnAll}(\text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \) requests asynchronous execution on all workers in the current parallel pool. If no pool exists, it starts a new parallel pool, unless your parallel preferences disable automatic creation of pools.

**Note** Use \text{parfevalOnAll} instead of \text{parfor} or \text{spmd} if you want to use \text{clear}. This preserves workspace transparency. See “Ensure Transparency in parfor-Loops” on page 2-58.

Examples

Unload a \text{mex} file before deleting temporary folders for distributing simulations, using the \text{clear} function. Because \text{clear} has 0 output arguments, specify 0 in the \text{numout} input argument of \text{parfevalOnAll}.

\[
\text{parfevalOnAll}(@\text{clear}, 0, '\text{mex}')
\]

Close all Simulink models on all workers:
p = gcp(); % Get the current parallel pool
f = parfevalOnAll(p,@bdclose,0,'all');
% No output arguments, but you might want to wait for completion
wait(f);

See Also
cancel | fetchNext | fetchOutputs | parallel.pool.Constant | parfeval | parpool | wait

Introduced in R2013b
parfor

Execute for-loop iterations in parallel on workers in parallel pool

Syntax

parfor loopVar = initVal:endVal; statements; end
parfor (loopVar = initVal:endVal,M); statements; end

Description

parfor loopVar = initVal:endVal; statements; end executes for-loop iterations in parallel on workers in a parallel pool.

MATLAB executes the loop body commands in statements for values of loopVar between initVal and endVal. loopVar specifies a vector of integer values increasing by 1. If you have Parallel Computing Toolbox, the iterations of statements can execute on a parallel pool of workers on your multi-core computer or cluster. As with a for-loop, you can include a single line or multiple lines in statements.

To find out how parfor can help increase your throughput, see “Decide When to Use parfor” on page 2-2.

parfor differs from a traditional for-loop in the following ways:

• Loop iterations are executed in parallel in a nondeterministic order. This means that you might need to modify your code to use parfor. For more help, see “Convert for-Loops Into parfor-Loops” on page 2-8.
• Loop iterations must be consecutive, increasing integer values.
• The body of the parfor-loop must be independent. One loop iteration cannot depend on a previous iteration, because the iterations are executed in a nondeterministic order. For more help, see “Ensure That parfor-Loop Iterations are Independent” on page 2-12.
• You cannot use a parfor-loop inside another parfor-loop. For more help, see “Nested parfor-Loops and for-Loops” on page 2-15.
parfor (loopVar = initVal:endVal,M); statements; end uses M to specify the maximum number of workers from the parallel pool to use in evaluating statements in the loop body. M must be a nonnegative integer.

By default, MATLAB uses the available workers in your parallel pool. You can change the number of workers on the Home tab in the Environment section, select Parallel > Parallel Preferences. You can override the default number of workers in a parallel pool by using parpool. When no workers are available in the pool or M is zero, MATLAB still executes the loop body in a nondeterministic order, but not in parallel. Use this syntax to switch between parallel and serial execution when testing your code.

To execute the iterations in parallel, you must have a parallel pool of workers. By default, if you execute parfor, you automatically create a parallel pool of workers on the cluster defined by your default cluster profile. The default cluster is local. You can change your cluster in Parallel Preferences. For more details, see “Specify Your Parallel Preferences” on page 6-12.

**Examples**

**Convert a for-Loop Into a parfor-Loop**

Create a parfor-loop for a computationally intensive task and measure the resulting speedup.

In the MATLAB Editor, enter the following for-loop. To measure the time elapsed, add tic and toc.

```matlab
 tic
n = 200;
A = 500;
a = zeros(1,n);
for i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc
```

Run the script, and note the elapsed time.

Elapsed time is 31.935373 seconds.

In the script, replace the for-loop with a parfor-loop.
tic
n = 200;
A = 500;
a = zeros(1,n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
toc

Run the new script, and run it again. The first run is slower than the second run, because the parallel pool has to be started, and you have to make the code available to the workers. Note the elapsed time for the second run.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

Elapsed time is 10.760068 seconds.

Observe that you speed up your calculation by converting the for-loop into a parfor-loop on four workers. You might reduce the elapsed time further by increasing the number of workers in your parallel pool. For more information, see “Convert for-Loops Into parfor-Loops” on page 2-8 and “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23.

Test parfor-Loops by Switching Between Parallel and Serial Execution

You can specify the maximum number of workers \( M \) for a parfor-loop. Set \( M = 0 \) to run the body of the loop in the desktop MATLAB, without using workers, even if a pool is open. When \( M = 0 \), MATLAB still executes the loop body in a nondeterministic order, but not in parallel, so that you can check whether your parfor-loops are independent and suitable to run on workers. This is the simplest way to allow you to debug the contents of a parfor-loop. You cannot set breakpoints directly in the body of the parfor-loop, but you can set breakpoints in functions called from the body of the parfor-loop.

Specify \( M = 0 \) to run the body of a parfor-loop in the desktop MATLAB, even if a pool is open.

\[
M = 0;
\]
\[
% M specifies maximum number of workers
\]
\[
y = ones(1,100);
\]
\[
parfor (i = 1:100,M)
\]
\[
y(i) = i;
\]
\[
end\]
To control the number of workers in your parallel pool, see “Specify Your Parallel Preferences” on page 6-12 and parpool.

**Measure Data Transferred to Workers Using a parfor-Loop**

To measure how much data is transferred to and from the workers in your current parallel pool, add ticBytes(gcp) and tocBytes(gcp) before and after the parfor-loop. Use gcp as an argument to get the current parallel pool.

Delete your current parallel pool if you still have one.

delete(gcp('nocreate'))

tic
ticBytes(gcp);
n = 200;
A = 500;
a = zeros(1,n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A))));
end
tocBytes(gcp)
toc

Run the new script, and run it again. The first run is slower than the second run, because the parallel pool has to be started, and you have to make the code available to the workers.

By default, MATLAB automatically opens a parallel pool of workers on your local machine.

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

<table>
<thead>
<tr>
<th></th>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15340</td>
<td>7024</td>
</tr>
<tr>
<td>2</td>
<td>13328</td>
<td>5712</td>
</tr>
<tr>
<td>3</td>
<td>13328</td>
<td>5704</td>
</tr>
<tr>
<td>4</td>
<td>13328</td>
<td>5728</td>
</tr>
<tr>
<td>Total</td>
<td>55324</td>
<td>24168</td>
</tr>
</tbody>
</table>

You can use the ticBytes and tocBytes results to examine the amount of data transferred to and from the workers in a parallel pool. In this example, the data transfer...
is small. For more information about parfor-loops, see “Decide When to Use parfor” on page 2-2 and “Convert for-Loops Into parfor-Loops” on page 2-8.

**Input Arguments**

loopVar — Loop index

integer

Loop index variable with initial value initVal and final value endVal. The variable can be any numeric type and the value must be an integer.

Make sure that your parfor-loop variables are consecutive increasing integers. For more help, see “Troubleshoot Variables in parfor-Loops” on page 2-32.

The range of the parfor-loop variable must not exceed the supported range. For more help, see “Avoid Overflows in parfor-Loops” on page 2-32.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

initVal — Initial value of loop index

integer

Initial value loop index variable, loopVar. The variable can be any numeric type and the value must be an integer. With endVal, specifies the parfor range vector, which must be of the form M:N.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

endVal — Final value of loop index

integer

Final value loop index variable, loopVar. The variable can be any numeric type and the value must be an integer. With initVal, specifies the parfor range vector, which must be of the form M:N.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

statements — Loop body

text
Loop body, specified as text. The series of MATLAB commands to execute in the `parfor`-loop.

You might need to modify your code to use `parfor`-loops. For more help, see “Convert for-Loops Into parfor-Loops” on page 2-8.

Do not nest `parfor`-loops, see “Nested parfor-Loops and for-Loops” on page 2-15.

**M — Maximum number of workers running in parallel**

number of workers in the parallel pool (default) | nonnegative integer

Maximum number of workers running in parallel, specified as a nonnegative integer. If you specify an upper limit, MATLAB uses no more than this number, even if additional workers are available. If you request more workers than the number of available workers, then MATLAB uses the maximum number of workers available at the time of the call. If the loop iterations are fewer than the number of workers, some workers perform no work.

If `parfor` cannot run on multiple workers (for example, if only one core is available or `M` is 0), MATLAB executes the loop in a serial manner. In this case, MATLAB still executes the loop body in a nondeterministic order. Use this syntax to switch between parallel and serial when testing your code.

**Tips**

- Use a `parfor`-loop when:
  - You have many loop iterations of a simple calculation. `parfor` divides the loop iterations into groups so that each thread can execute one group of iterations.
  - You have some loop iterations that take a long time to execute.
  - Do not use a `parfor`-loop when an iteration in your loop depends on the results of other iterations.

  Reductions are one exception to this rule. A reduction variable accumulates a value that depends on all the iterations together, but is independent of the iteration order. For more information, see “Reduction Variables” on page 2-47.

- When you use `parfor`, you have to wait for the loop to complete to obtain your results. Your client MATLAB is blocked and you cannot break out of the loop early. If you want to obtain intermediate results, or break out of a `for`-loop early, try `parfeval` instead.
- A `parfor`-loop runs on the existing parallel pool. If no pool exists, `parfor` starts a new parallel pool, unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `parfor` cannot start one, the loop runs serially in the client session.

- If the `AutoAttachFiles` property in the cluster profile for the parallel pool is set to `true`, MATLAB performs an analysis on a `parfor`-loop to determine what code files are necessary for its execution, see `listAutoAttachedFiles`. Then MATLAB automatically attaches those files to the parallel pool so that the code is available to the workers.

- You cannot call scripts directly in a `parfor`-loop. However, you can call functions that call scripts.

- Do not use `clear` inside a `parfor` loop because it violates workspace transparency. See “Ensure Transparency in parfor-Loops” on page 2-58.

- You can run Simulink models in parallel with the `parsim` command instead of using `parfor`-loops. For more information and examples of using Simulink in parallel, see “Run Multiple Simulations” (Simulink).

### See Also

- `afterEach`
- `for`
- `gcp`
- `listAutoAttachedFiles`
- `parfeval`
- `parpool`
- `send`
- `ticBytes`
- `tocBytes`

### Topics

- “Decide When to Use parfor” on page 2-2
- “Convert for-Loops Into parfor-Loops” on page 2-8
- “Ensure That parfor-Loop Iterations are Independent” on page 2-12
- “Nested parfor-Loops and for-Loops” on page 2-15
- “Troubleshoot Variables in parfor-Loops” on page 2-32
- “Scale Up parfor-Loops to Cluster and Cloud” on page 2-23
- “Specify Your Parallel Preferences” on page 6-12
- “Run Parallel Simulations” (Simulink)

### Introduced in R2008a
parpool

Create parallel pool on cluster

Syntax

parpool
parpool(poolsize)
parpool(profilename)
parpool(profilename,poolsize)
parpool(cluster)
parpool(cluster,poolsize)
parpool(___,Name,Value)
poolobj = parpool(__)

Description

parpool starts a parallel pool of workers using the default cluster profile, with the pool size specified by your parallel preferences and the default profile. parpool creates a pool on the default cluster with its NumWorkers in the range [1, preferredNumWorkers] for running parallel language features. preferredNumWorkers is the value defined in your parallel preferences.

parpool enables the full functionality of the parallel language features in MATLAB by creating a special job on a pool of workers, and connecting the MATLAB client to the parallel pool. Parallel language features include parfor, parfeval, parfevalOnAll, spmd, and distributed. If possible, the working folder on the workers is set to match that of the MATLAB client session.

parpool(poolsize) creates and returns a pool with the specified number of workers. poolsize can be a positive integer or a range specified as a 2-element vector of integers. If poolsize is a range, the resulting pool has size as large as possible in the range requested.

Specifying the poolsize overrides the number of workers specified in the preferences or profile, and starts a pool of exactly that number of workers, even if it has to wait for them
to be available. Most clusters have a maximum number of workers they can start. If the profile specifies a MATLAB job scheduler (MJS) cluster, parpool reserves its workers from among those already running and available under that MJS. If the profile specifies a local or third-party scheduler, parpool instructs the scheduler to start the workers for the pool.

parpool(profilename) or parpool(profilename,poolsize) starts a worker pool using the cluster profile identified by profilename.

parpool(cluster) or parpool(cluster,poolsize) starts a worker pool on the cluster specified by the cluster object cluster.

parpool(___,Name,Value) applies the specified values for certain properties when starting the pool.

poolobj = parpool(___) returns a parallel.Pool object to the client workspace representing the pool on the cluster. You can use the pool object to programmatically delete the pool or to access its properties. Use delete(pool) to shut down the parallel pool.

Examples

Create Pool from Default Profile

Start a parallel pool using the default profile to define the number of workers.

parpool

Create Pool from Specified Profile

Start a parallel pool of 16 workers using a profile called myProf.

parpool('myProf',16)

Create Pool from Local Profile

Start a parallel pool of 2 workers using the local profile.
parpool('local',2)

**Create Pool on Specified Cluster**

Create an object representing the cluster identified by the default profile, and use that cluster object to start a parallel pool. The pool size is determined by the default profile.

```matlab
c = parcluster
parpool(c)
```

**Create Pool and Attach Files**

Start a parallel pool with the default profile, and pass two code files to the workers.

```matlab
parpool('AttachedFiles', {'mod1.m', 'mod2.m'})
```

**Return Pool Object and Delete Pool**

Create a parallel pool with the default profile, and later delete the pool.

```matlab
poolobj = parpool;
delete(poolobj)
```

**Determine Size of Current Pool**

Find the number of workers in the current parallel pool.

```matlab
poolobj = gcp('nocreate'); % If no pool, do not create new one.
if isempty(poolobj)
    poolsize = 0;
else
```
poolsize = poolobj.NumWorkers
end

Input Arguments

poolsize — Size of parallel pool
positive integer | 2-element vector of integers

Size of the parallel pool, specified as a positive integer or a range specified as a 2-element vector of integers. If poolsize is a range, the resulting pool has size as large as possible in the range requested. Set the default preferred number of workers in the parallel preferences or parallel profile.

Example: parpool('local',2)

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |
uint32 | uint64

profilename — Profile that defines cluster and properties
character vector

Profile that defines cluster and properties, specified as a character vector.

Example: parpool('myClusterProfile',16)

Data Types: char

cluster — Cluster to start pool on
cluster object

Cluster to start pool on, specified as a cluster object. Use parcluster to get a cluster object.

Example: c = parcluster; parpool(c)

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'AttachedFiles',{'myFun.m'}
AttachedFiles — Files to attach to pool
character vector | string scalar | string array | cell array of character vectors

Files to attach to pool, specified as a character vector, string or string array, or cell array of character vectors.

With this argument pair, `parpool` starts a parallel pool and passes the identified files to the workers in the pool. The files specified here are appended to the `AttachedFiles` property specified in the applicable parallel profile to form the complete list of attached files. The 'AttachedFiles' property name is case sensitive, and must appear as shown. Example: `{'myFun.m','myFun2.m'}`

Data Types: `char` | `cell`

AutoAddClientPath — Specifies if client path is added to worker path
ture (default) | false

A logical value (true or false) that controls whether user-added-entries on the client's path are added to each worker's path at startup. By default 'AutoAddClientPath' is set to true.

Data Types: `logical`

EnvironmentVariables — Environment variables copied to workers
character vector | string scalar | string array | cell array of character vectors

Names of environment variables to copy from the client session to the workers, specified as a character vector, string or string array, or cell array of character vectors. The names specified here are appended to the 'EnvironmentVariables' property specified in the applicable parallel profile to form the complete list of environment variables. Any variables listed which are not set are not copied to the workers. These environment variables are set on the workers for the duration of the parallel pool.

Data Types: `char` | `cell`

SpmdEnabled — Indication if pool is enabled to support SPMD
true (default) | false

Indication if pool is enabled to support SPMD, specified as a logical. You can disable support only on a local or MJS cluster. Because `parfor` iterations do not involve interworker communication, disabling SPMD support this way allows the parallel pool to keep evaluating a `parfor`-loop even if one or more workers aborts during loop execution.
Data Types: logical

Output Arguments

poolobj — Access to parallel pool from client
parallel.Pool object

Access to parallel pool from client, returned as a parallel.Pool object.

Tips

**Note** Remove any *startup.m* from your MATLAB path if you want to run any parallel code including `parpool`. If you have trouble starting the parallel pool, see this MATLAB Answers page: https://uk.mathworks.com/matlabcentral/answers/92124-why-am-i-unable-to-use-parpool-with-the-local-scheduler-or-validate-my-local-configuration-of-parall

- The pool status indicator in the lower-left corner of the desktop shows the client session connection to the pool and the pool status. Click the icon for a menu of supported pool actions.

With a pool running: ![Start parallel pool](#) With no pool running: ![Start parallel pool](#)

- If you set your parallel preferences to automatically create a parallel pool when necessary, you do not need to explicitly call the `parpool` command. You might explicitly create a pool to control when you incur the overhead time of setting it up, so the pool is ready for subsequent parallel language constructs.

- `delete(poolobj)` shuts down the parallel pool. Without a parallel pool, `spmd` and `parfor` run as a single thread in the client, unless your parallel preferences are set to automatically start a parallel pool for them.

- When you use the MATLAB editor to update files on the client that are attached to a parallel pool, those updates automatically propagate to the workers in the pool. (This automatic updating does not apply to Simulink model files. To propagate updated model files to the workers, use the `updateAttachedFiles` function.)
• If possible, the working folder on the workers is initially set to match that of the MATLAB client session. Subsequently, the following commands entered in the client Command Window also execute on all the workers in the pool:

```plaintext
cd
addpath
rmpath
```

This behavior allows you to set the working folder and the command search path on all the workers, so that subsequent pool activities such as `parfor`-loops execute in the proper context.

When changing folders or adding a path with `cd` or `addpath` on clients with Windows operating systems, the value sent to the workers is the UNC path for the folder if possible. For clients with Linux® operating systems, it is the absolute folder location.

If any of these commands does not work on the client, it is not executed on the workers either. For example, if `addpath` specifies a folder that the client cannot access, the `addpath` command is not executed on the workers. However, if the working folder can be set on the client, but cannot be set as specified on any of the workers, you do not get an error message returned to the client Command Window.

Be careful of this slight difference in behavior in a mixed-platform environment where the client is not the same platform as the workers, where folders local to or mapped from the client are not available in the same way to the workers, or where folders are in a nonshared file system. For example, if you have a MATLAB client running on a Microsoft Windows operating system while the MATLAB workers are all running on Linux operating systems, the same argument to `addpath` cannot work on both. In this situation, you can use the function `pctRunOnAll` to assure that a command runs on all the workers.

Another difference between client and workers is that any `addpath` arguments that are part of the `matlabroot` folder are not set on the workers. The assumption is that the MATLAB install base is already included in the workers’ paths. The rules for `addpath` regarding workers in the pool are:

- Subfolders of the `matlabroot` folder are not sent to the workers.
- Any folders that appear before the first occurrence of a `matlabroot` folder are added to the top of the path on the workers.
- Any folders that appear after the first occurrence of a `matlabroot` folder are added after the `matlabroot` group of folders on the workers’ paths.
For example, suppose that \texttt{matlabroot} on the client is \texttt{C:\Applications\matlab}. With an open parallel pool, execute the following to set the path on the client and all workers:

\begin{verbatim}
addpath('P1',
    'P2',
    'C:\Applications\matlab\T3',
    'C:\Applications\matlab\T4',
    'P5',
    'C:\Applications\matlab\T6',
    'P7',
    'P8');
\end{verbatim}

Because \texttt{T3}, \texttt{T4}, and \texttt{T6} are subfolders of \texttt{matlabroot}, they are not set on the workers’ paths. So on the workers, the pertinent part of the path resulting from this command is:

\begin{verbatim}
P1
P2
<worker original matlabroot folders...>
P5
P7
P8
\end{verbatim}

\textbf{See Also}

\texttt{Composite}\mid\texttt{delete}\mid\texttt{distributed}\mid\texttt{gcp}\mid\texttt{parallel.defaultClusterProfile}\mid\texttt{parallel.pool.Constant}\mid\texttt{parcluster}\mid\texttt{parfeval}\mid\texttt{parfevalOnAll}\mid\texttt{parfor}\mid\texttt{pctRunOnAll}\mid\texttt{spmd}

\textbf{Topics}

“Specify Your Parallel Preferences” on page 6-12
“Discover Clusters and Use Cluster Profiles” on page 6-15
“Pass Data to and from Worker Sessions” on page 7-17

\textbf{Introduced in R2013b}
**pause**

Pause MATLAB job scheduler queue

**Syntax**

```matlab
pause(mjs)
```

**Arguments**

- **mjs** MATLAB job scheduler object whose queue is paused.

**Description**

`pause(mjs)` pauses the MATLAB job scheduler’s queue so that jobs waiting in the queued state will not run. Jobs that are already running also pause, after completion of tasks that are already running. No further jobs or tasks will run until the `resume` function is called for the MJS.

The `pause` function does nothing if the MJS is already paused.

**See Also**

`resume` | `wait`

**Topics**

“Program Independent Jobs for a Supported Scheduler” on page 7-8

**Introduced before R2006a**
**pctconfig**

Configure settings for Parallel Computing Toolbox client session

**Syntax**

```
pctconfig('p1',v1,...)
cfg = pctconfig('p1',v1,...)
cfg = pctconfig()
```

**Arguments**

- `p1` Property to configure. Supported properties are 'portrange', 'hostname'.
- `v1` Value for corresponding property.
- `cfg` Structure of configuration value.

**Description**

```
pctconfig('p1',v1,...) sets the client configuration property p1 with the value v1.
```

Note that the property value pairs can be in any format supported by the `set` function, i.e., param-value character vector pairs, structures, and param-value cell array pairs. If a structure is used, the structure field names are the property names and the field values specify the property values.

If the property is 'portrange', the specified value is used to set the range of ports to be used by the client session of Parallel Computing Toolbox software. This is useful in environments with a limited choice of ports. The value of 'portrange' should either be a 2-element vector `[minport, maxport]` specifying the range, or 0 to specify that the client session should use ephemeral ports. By default, the client session searches for available ports to communicate with the other sessions of MATLAB Distributed Computing Server software.
If the property is `'hostname'`, the specified value is used to set the hostname for the client session of Parallel Computing Toolbox software. This is useful when the client computer is known by more than one hostname. The value you should use is the hostname by which the cluster nodes can contact the client computer. The toolbox supports both short hostnames and fully qualified domain names.

`config = pctconfig('p1',v1,...)` returns a structure to `config`. The field names of the structure reflect the property names, while the field values are set to the property values.

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

**Examples**

View the current settings for hostname and ports.

```matlab
config = pctconfig()
```

```matlab
cfg =
    portrange: [27370 27470]
    hostname: 'machine32'
```

Set the current client session port range to 21000-22000 with hostname `fdm4`.

```matlab
pctconfig('hostname','fdm4','portrange',[21000 22000]);
```

Set the client hostname to a fully qualified domain name.

```matlab
pctconfig('hostname','desktop24.subnet6.companydomain.com');
```

**Tips**

The values set by this function do not persist between MATLAB sessions. To guarantee its effect, call `pctconfig` before calling any other Parallel Computing Toolbox functions.

**Introduced in R2008a**
**pctRunDeployedCleanup**

Clean up after deployed parallel applications

**Syntax**

pctRunDeployedCleanup

**Description**

pctRunDeployedCleanup performs necessary cleanup so that the client JVM can properly terminate when the deployed application exits. All deployed applications that use Parallel Computing Toolbox functionality need to call `pctRunDeployedCleanup` after the last call to Parallel Computing Toolbox functionality.

After calling `pctRunDeployedCleanup`, you should not use any further Parallel Computing Toolbox functionality in the current MATLAB session.

**Introduced in R2010a**
**pctRunOnAll**

Run command on client and all workers in parallel pool

**Syntax**

pctRunOnAll command

**Description**

pctRunOnAll command runs the specified command on all the workers of the parallel pool as well as the client, and prints any command-line output back to the client Command Window. The specified command runs in the base workspace of the workers and does not have any return variables. This is useful if there are setup changes that need to be performed on all the workers and the client.

**Note** If you use pctRunOnAll to run a command such as addpath in a mixed-platform environment, it can generate a warning on the client while executing properly on the workers. For example, if your workers are all running on Linux operating systems and your client is running on a Microsoft Windows operating system, an addpath argument with Linux-based paths will warn on the Windows-based client.

**Examples**

Clear all loaded functions on all workers:

pctRunOnAll clear functions

Change the directory on all workers to the project directory:

pctRunOnAll cd /opt/projects/c1456

Add some directories to the paths of all the workers:

pctRunOnAll addpath({'/usr/share/path1' '/usr/share/path2'})
See Also
parpool

Introduced in R2008a
pload

Load file into parallel session

Syntax

pload(fileroot)

Arguments

fileroot Part of filename common to all saved files being loaded.

Description

pload(fileroot) loads the data from the files named [fileroot num2str(labindex)] into the workers running a communicating job. The files should have been created by the psave command. The number of workers should be the same as the number of files. The files should be accessible to all the workers. Any codistributed arrays are reconstructed by this function. If fileroot contains an extension, the character representation of the labindex will be inserted before the extension. Thus, pload('abc') attempts to load the file abc1.mat on worker 1, abc2.mat on worker 2, and so on.

Examples

Create three variables — one replicated, one variant, and one codistributed. Then save the data. (This example works in a communicating job or in pmode, but not in a parfor or spmd block.)

```matlab
clear all;
rep = speye(numlabs);
var = magic(labindex);
D = eye(numlabs,codistributor());
psave('threeThings');
```
This creates three files (threeThings1.mat, threeThings2.mat, threeThings3.mat) in the current working directory.

Clear the workspace on all the workers and confirm there are no variables.

clear all
whos

Load the previously saved data into the workers. Confirm its presence.

pload('threeThings');
whos
isreplicated(rep)
iscodistributed(D)

See Also
labindex | load | numlabs | pmode | psave | save

Introduced in R2006b
pmode

Interactive Parallel Command Window

Syntax

pmode start
pmode start numworkers
pmode start prof numworkers
pmode quit
pmode exit
pmode client2lab clientvar workers workervar
pmode lab2client workervar worker clientvar
pmode cleanup prof

Description

pmode allows the interactive parallel execution of MATLAB commands. pmode achieves this by defining and submitting a communicating job, and opening a Parallel Command Window connected to the workers running the job. The workers then receive commands entered in the Parallel Command Window, process them, and send the command output back to the Parallel Command Window. Variables can be transferred between the MATLAB client and the workers. For more details about program development in pmode and its interactive features, see “Run Communicating Jobs Interactively Using pmode” on page 4-3.

pmode start starts pmode, using the default profile to define the cluster and number of workers. (The initial default profile is local; you can change it by using the function parallel.defaultClusterProfile.) You can also specify the number of workers using pmode start numworkers.

pmode start prof numworkers starts pmode using the Parallel Computing Toolbox profile prof to locate the cluster, submits a communicating job with the number of workers identified by numworkers, and connects the Parallel Command Window with the workers. If the number of workers is specified, it overrides the minimum and maximum number of workers specified in the profile.
pmode quit or pmode exit stops the pmode job, deletes it, and closes the Parallel Command Window. You can enter this command at the MATLAB prompt or the pmode prompt.

pmode client2lab clientvar workers workervar copies the variable clientvar from the MATLAB client to the variable workervar on the workers identified by workers. If workervar is omitted, the copy is named clientvar. workers can be either a single index or a vector of indices. You can enter this command at the MATLAB prompt or the pmode prompt.

pmode lab2client workervar worker clientvar copies the variable workervar from the worker identified by worker, to the variable clientvar on the MATLAB client. If clientvar is omitted, the copy is named workervar. You can enter this command at the MATLAB prompt or the pmode prompt. Note: If you use this command in an attempt to transfer a codistributed array to the client, you get a warning, and only the local portion of the array on the specified worker is transferred. To transfer an entire codistributed array, first use the gather function to assemble the whole array into the worker workspaces.

pmode cleanup prof deletes all communicating jobs created by pmode for the current user running on the cluster specified in the profile prof, including jobs that are currently running. The profile is optional; the default profile is used if none is specified. You can enter this command at the MATLAB prompt or the pmode prompt.

You can invoke pmode as either a command or a function, so the following are equivalent.

```
  pmode start prof 4
  pmode('start','prof',4)
```

**Examples**

In the following examples, the pmode prompt (P>>) indicates commands entered in the Parallel Command Window. Other commands are entered in the MATLAB Command Window.

Start pmode using the default profile to identify the cluster and number of workers.

<code>pmode start</code>

Start pmode using the local profile with four local workers.
pmode start local 4
Start pmode using the profile myProfile and eight workers on the cluster.

pmode start myProfile 8
Execute a command on all workers.

P>> x = 2*labindex;
Copy the variable x from worker 7 to the MATLAB client.

pmode lab2client x 7
Copy the variable y from the MATLAB client to workers 1 through 8.

pmode client2lab y 1:8
Display the current working directory of each worker.

P>> pwd

See Also
createCommunicatingJob | parallel.defaultClusterProfile | parcluster

Introduced in R2006b
poll

Retrieve data sent from a worker

Syntax

poll(pollablequeue)
[data, OK] = poll(pollablequeue, timeout)

Description

poll(pollablequeue) retrieves the result of a message or data sent from a worker to the parallel.pool.PollableDataQueue specified by pollablequeue. You can use poll only in the process where you created the data queue.

[data, OK] = poll(pollablequeue, timeout) returns data, and OK as a boolean true to indicate that data has been returned. If there is no data in the queue, then an empty array is returned with a boolean false for OK. Specify timeout in seconds as an optional second parameter. In that case, the method might block for the time specified by timeout before returning. If any data arrives in the queue during that period, that data is returned.

Examples

Send a Message in a parfor-loop, and Poll for the Result

Construct a PollableDataQueue.

p = parallel.pool.PollableDataQueue;

Start a parfor-loop, and send a message, such as data with the value 1.

parfor i = 1
    send(p, i);
end
Poll for the result.

```
poll(p)
```

1

For more details on sending data using a `PollableDataQueue`, see `send`.

**Send and Poll for Data while Using parfeval**

This example shows how to return intermediate results from a worker to the client and to display the result on the client.

Construct a `PollableDataQueue`. A `PollableDataQueue` is most useful for sending and polling for data during asynchronous function evaluations using `parfeval` or `parfevalOnAll`.

```
q = parallel.pool.PollableDataQueue;
```

Start a timer and send the data queue as input to the function for `parfeval` execution on the pool. Display the time elapsed and the data returned.

```
f = parfeval(@workerFcn, 0, q);
msgsReceived = 0;
starttime = tic;
while msgsReceived < 2
    [data, gotMsg] = poll(q, 1);
    if gotMsg
        fprintf('Got message: %s after %.3g seconds\n', ...
            data, toc(starttime));
        msgsReceived = msgsReceived + 1;
    else
        fprintf('No message available at %.3g seconds\n', ...
            toc(starttime));
    end
end
```

```
function workerFcn(q)
    send(q, 'start');
pause(3);
    send(q, 'stop');
end
```
The first message is returned in 0.39 s after you have executed `parfeval`. In that time the data and function for `parfeval` have been serialized, sent over to the workers, deserialized and set running. When you start the code, the worker sends some data, which is serialized, sent over the network back to the client and put on a data queue. `poll` notes this operation and returns the value to the client function. Then the time taken since `parfeval` was called is displayed. Note a delay of 3 s while the worker is computing something (in this case a long pause).

**Input Arguments**

`pollablequeue — Pollable data queue`  
`parallel.pool.PollableDataQueue`  
Pollable data queue, specified as a `parallel.pool.PollableDataQueue` object.  
Example: `[data, OK] = poll(pollablequeue, optionalTimeout);`

`timeout — Optional timeout`  
`scalar`  
Optional timeout interval (in seconds) used to block `poll` before returning, specified as a scalar.  
Example: `[data, OK] = poll(pollablequeue, timeout);`

**Output Arguments**

`data — Message or data`  
`scalar | vector | matrix | array | string | character vector`  
Message or data from workers to a data queue, specified as any serializable value.  
Example: `[data, OK] = poll(pollablequeue, timeout);`

`OK — Check if data has been returned`  
`Boolean`
Check if data has been returned, returned as a Boolean value. If data has been returned, then OK is assigned the value of a boolean true. If there is no data in the queue pollablequeue, then an empty array is returned and a boolean false for OK.

Example: [data, OK] = poll(pollablequeue, timeout);

See Also
afterEach | parallel.pool.DataQueue | parallel.pool.PollableDataQueue | parfeval | parfevalOnAll | parfor | parpool | send

Introduced in R2017a
**poolStartup**

File for user-defined options to run on each worker when parallel pool starts

**Syntax**

`poolStartup`

**Description**

`poolStartup` runs automatically on a worker each time the worker forms part of a parallel pool. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the `poolStartup.m` file to define pool initialization on the worker. The worker looks for `poolStartup.m` in the following order, executing the one it finds first:

1. Included in the job's `AttachedFiles` property.
2. In a folder included in the job's `AdditionalPaths` property.
3. In the worker's MATLAB installation at the location
   
   \[ \text{matlabroot}/toolbox/distcomp/user/poolStartup.m \]

To create a version of `poolStartup.m` for `AttachedFiles` or `AdditionalPaths`, copy the provided file and modify it as required.

`poolStartup` is the ideal location for startup code required for parallel execution on the parallel pool. For example, you might want to include code for using `mpiSettings`. Because `jobStartup` and `taskStartup` execute before `poolStartup`, they are not suited to pool-specific code. In other words, you should use `taskStartup` for setup code on your worker regardless of whether the task is from an independent job, communicating job, or using a parallel pool; while `poolStartup` is for setup code for pool usage only.

For further details on `poolStartup` and its implementation, see the text in the installed `poolStartup.m` file.
See Also
jobStartup | taskFinish | taskStartup

Introduced in R2010a
**promote**

Promote job in MJS cluster queue

**Syntax**

promote(c,job)

**Arguments**

- **c** The MJS cluster object that contains the job.
- **job** Job object promoted in the queue.

**Description**

promote(c, job) promotes the job object **job**, that is queued in the MJS cluster **c**.

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

**Examples**

Create and submit multiple jobs to the cluster identified by the default cluster profile, assuming that the default cluster profile uses an MJS:

```matlab
c = parcluster();
pause(c) % Prevent submissions from running.
j1 = createJob(c,'Name','Job A');
j2 = createJob(c,'Name','Job B');
j3 = createJob(c,'Name','Job C');
submit(j1);submit(j2);submit(j3);
```

Promote **Job C** by one position in its queue:
promote(c,j3)

Examine the new queue sequence:

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

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

**Tips**

After a call to `promote` or `demote`, there is no change in the order of job objects contained in the `Jobs` property of the MJS cluster object. To see the scheduled order of execution for jobs in the queue, use the `findJob` function in the form `[pending queued running finished] = findJob(c)`.

**See Also**

createJob | demote | findJob | submit

*Introduced before R2006a*
**psave**

Save data from communicating job session

**Syntax**

```matlab
psave(fileroott)
```

**Arguments**

fileroott  Part of filename common to all saved files.

**Description**

`psave(fileroott)` saves the data from the workers' workspace into the files named `[fileroott num2str(labindex)]`. The files can be loaded by using the `pload` command with the same `fileroott`, which should point to a folder accessible to all the workers. If `fileroott` contains an extension, the character representation of the `labindex` is inserted before the extension. Thus, `psave('abc')` creates the files 'abc1.mat', 'abc2.mat', etc., one for each worker.

**Examples**

Create three arrays — one replicated, one variant, and one codistributed. Then save the data. (This example works in a communicating job or in pmode, but not in a `parfor` or `spmd` block.)

```matlab
clear all;
rep = speye(numlabs);
var = magic(labindex);
D = eye(numlabs,codistributor());
psave('threeThings');
```

This creates three files (threeThings1.mat, threeThings2.mat, threeThings3.mat) in the current working folder.
Clear the workspace on all the workers and confirm there are no variables.

```matlab
clear all
whos
```

Load the previously saved data into the workers. Confirm its presence.

```matlab
pload('threeThings');
whos
isreplicated(rep)
iscodistributed(D)
```

**See Also**

labindex | load | numlabs | pload | pmode | save

**Introduced in R2006b**
rand

Array of rand values

Syntax

\[
R = \text{rand}(sz, \text{arraytype})
\]

\[
R = \text{rand}(sz, \text{datatype}, \text{arraytype})
\]

\[
R = \text{rand}(sz, 'like', P)
\]

\[
R = \text{rand}(sz, \text{datatype}, 'like', P)
\]

\[
C = \text{rand}(sz, \text{codist})
\]

\[
C = \text{rand}(sz, \text{datatype}, \text{codist})
\]

\[
C = \text{rand}(sz, \_\_\_, \text{codist}, '\text{noCommunication}')
\]

\[
C = \text{rand}(sz, \_\_\_, \text{codist}, 'like', P)
\]

Description

\[
R = \text{rand}(sz, \text{arraytype})
\] creates a matrix with underlying class of double, with rand values in all elements.

\[
R = \text{rand}(sz, \text{datatype}, \text{arraytype})
\] creates a matrix with underlying class of \text{datatype}, with rand values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( sz )</td>
<td>( n )</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>( m,n ) or ([m \ n])</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>( m,n,\ldots,k ) or ([m \ n \ \ldots \ k])</td>
<td>Specifies size as an m-by-n-by-\ldots-by-k array.</td>
</tr>
<tr>
<td>( arraytype )</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
</tbody>
</table>
R = rand(sz,'like',P) creates an array of rand values with the same type and underlying class (data type) as array P.

R = rand(sz,datatype,'like',P) creates an array of rand values with the specified underlying class (datatype), and the same type as array P.

C = rand(sz,codist) or C = rand(sz,datatype,codist) creates a codistributed array of rand values with the specified size and underlying class (the default datatype is 'double'). The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = rand(8,codistributor1d());
end
```

C = rand(sz,___,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = rand(sz,___,codist,'like',P) creates a codistributed array of rand values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

## Examples

### Create Distributed Rand Matrix

Create a 1000-by-1000 distributed array of rands with underlying class double:
D = rand(1000,'distributed');

Create Codistributed Rand Matrix

Create a 1000-by-1000 codistributed double matrix of rands, distributed by its second dimension (columns).

```
spmd(4)
    C = rand(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed single matrix of rands, distributed by its columns.

```
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = rand(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray Rand Matrix

Create a 1000-by-1000 gpuArray of rands with underlying class double:

```
G = rand(1000,'double','gpuArray');
```

See Also
codistributed.sprand | distributed.sprand | rand | randi | randn

Introduced in R2006b
randi

Array of random integers

Syntax

\[
R = \text{randi}(\text{valrange}, \text{sz}, \text{arraytype}) \\
R = \text{randi}(\text{valrange}, \text{sz}, \text{datatype}, \text{arraytype}) \\
R = \text{randi}(\text{valrange}, \text{sz}, \text{'like'}, P) \\
R = \text{randi}(\text{valrange}, \text{sz}, \text{datatype}, \text{'like'}, P) \\
C = \text{randi}(\text{valrange}, \text{sz}, \text{codist}) \\
C = \text{randi}(\text{valrange}, \text{sz}, \text{datatype}, \text{codist}) \\
C = \text{randi}(\text{valrange}, \text{sz}, \text{'like'}, P) \\
C = \text{randi}(\text{valrange}, \text{sz}, \text{'like'}, P)
\]

Description

\(R = \text{randi}(\text{valrange}, \text{sz}, \text{arraytype})\) creates a matrix with underlying class of double, with \text{randi} integer values in all elements.

\(R = \text{randi}(\text{valrange}, \text{sz}, \text{datatype}, \text{arraytype})\) creates a matrix with underlying class of \text{datatype}, with \text{randi} values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{valrange}</td>
<td>max or [min max]</td>
<td>Specifies integer value range from 1 to max, or from min to max.</td>
</tr>
<tr>
<td>\text{sz}</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m, n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
</tbody>
</table>
### Argument | Values | Descriptions
--- | --- | ---
m,n,...,k or [m \ n \ ... \ k] | Specifies size as an m-by-n-by-...-by-k array.

*arraytype* | 'distributed' | Specifies distributed array.
| 'codistributed' | Specifies codistributed array, using the default distribution scheme.
| 'gpuArray' | Specifies gpuArray.

datatype | 'double' (default), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64' | Specifies underlying class of the array, i.e., the data type of its elements.

R = randi(valrange,sz,'like',P) creates an array of randi values with the same type and underlying class (data type) as array P.

R = randi(valrange,sz,datatype,'like',P) creates an array of randi values with the specified underlying class (datatype), and the same type as array P.

C = randi(valrange,sz,codist) or C = randi(valrange,sz,datatype,codist) creates a codistributed array of randi values with the specified size and underlying class (the default datatype is 'double'). The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```matlab
spmd
    C = randi(8,codistributor1d());
end
```

C = randi(valrange,sz,____,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.
C = randi(valrange,sz,___,codist,'like',P) creates a codistributed array of randi values with the specified range, size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

Examples

Create Distributed Randi Matrix

Create a 1000-by-1000 distributed array of randi values from 1 to 100, with underlying class double:

D = randi(100,1000,'distributed');

Create Codistributed Randi Matrix

Create a 1000-by-1000 codistributed double matrix of randi values from 0 to 12, distributed by its second dimension (columns).

spmd(4)
    C = randi([0 12],1000,'codistributed');
end

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed single matrix of randi values from 1 to 4, distributed by its columns.

spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = randi(4,1000,1000,'single',codist);
end

Each worker contains a 100-by-labindex local piece of C.

Create gpuArray Randi Matrix

Create a 1000-by-1000 gpuArray of randi values from —50 to 50, with underlying class double:

G = randi([-50 50],1000,'double','gpuArray');
See Also
rand | randi | randn

Introduced in R2014a
randn

Array of randn values

Syntax

R = randn(sz,arraytype)
R = randn(sz,datatype,arraytype)

R = randn(sz,'like',P)
R = randn(sz,datatype,'like',P)

C = randn(sz,codist)
C = rand(sz,datatype,codist)
C = randn(sz,___,codist,'noCommunication')
C = randn(sz,___,codist,'like',P)

Description

R = randn(sz,arraytype) creates a matrix with underlying class of double, with randn values in all elements.

R = randn(sz,datatype,arraytype) creates a matrix with underlying class of datatype, with randn values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n,...,k or [m n ... k]</td>
<td>Specifies size as an m-by-n-by-...-by-k array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
<tr>
<td>Argument</td>
<td>Values</td>
<td>Descriptions</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>'codistributed'</td>
<td>Specifies codistributed array, using the default distribution scheme.</td>
</tr>
<tr>
<td></td>
<td>'gpuArray'</td>
<td>Specifies gpuArray.</td>
</tr>
<tr>
<td>datatype</td>
<td>'double' (default),</td>
<td>Specifies underlying class of the array, i.e., the data type of its elements.</td>
</tr>
</tbody>
</table>

R = randn(sz,'like',P) creates an array of randn values with the same type and underlying class (data type) as array P.

R = randn(sz,datatype,'like',P) creates an array of randn values with the specified underlying class (datatype), and the same type as array P.

C = randn(sz,codist) or C = rand(sz,datatype,codist) creates a codistributed array of randn values with the specified size and underlying class (the default datatype is 'double'). The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```
spmd
    C = randn(8,codistributor1d());
end
```

C = randn(sz,___,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = randn(sz,___,codist,'like',P) creates a codistributed array of randn values with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array P.

**Examples**

**Create Distributed Randn Matrix**

Create a 1000-by-1000 distributed array of randn values with underlying class double:
D = randn(1000,'distributed');

**Create Codistributed Randn Matrix**

Create a 1000-by-1000 codistributed double matrix of randn values, distributed by its second dimension (columns).

```matlab
spmd(4)
    C = randn(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed single matrix of randn values, distributed by its columns.

```matlab
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = randn(1000,1000,'single',codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

**Create gpuArray Rand Matrix**

Create a 1000-by-1000 gpuArray of randn values with underlying class double:

```matlab
G = randn(1000,'double','gpuArray');
```

**See Also**

codistributed.sprandn | distributed.sprandn | rand | randi | randn

**Introduced in R2006b**
recreate

Create new job from existing job

Syntax

newjob = recreate(oldjob)
newjob = recreate(oldjob,'Tasks',tasksToRecreate)
newjob = recreate(oldjob,'TaskState',states)
newjob = recreate(oldjob,'TaskID',ids)

Description

newjob = recreate(oldjob) creates a new job object based on an existing job, containing the same tasks and options as oldjob. The old job can be in any state; the new job state is pending. If oldjob was created using batch, then MATLAB automatically submits the new job.

newjob = recreate(oldjob,'Tasks',tasksToRecreate) creates a job object with tasks that correspond to tasksToRecreate. Because communicating jobs have only one task, this option only supports independent jobs.

newjob = recreate(oldjob,'TaskState',states) creates a job object with tasks that correspond to the tasks with State specified by states. Because communicating jobs have only one task, this option only supports independent jobs.

newjob = recreate(oldjob,'TaskID',ids) creates a job object containing the tasks from oldjob that correspond to the tasks with IDs specified by ids. Because communicating jobs have only one task, this option only supports independent jobs.

Examples

Use recreate to Resubmit Tasks with Errors

This approach is useful when tasks depend on a file that is not present anymore.
Create a new job using the default cluster profile. In this example, it is the local parallel pool.

```matlab
cluster = parcluster;
job = createJob(cluster);
```

Create several tasks. In particular, create a task that depends on a MAT-file that does not exist.

```matlab
createTask(job,@() 'Task1',1);
createTask(job,@() load('myData.mat'),1);
```

Submit the job, and wait for it to finish. Because the MAT-file in the second task does not exist, the job fails. If you call `fetchOutputs` on `job` to retrieve the results, you get an error. Check the error using the `Error` property of the corresponding task.

```matlab
submit(job);
wait(job);
job.Tasks(2).Error
```

```matlab
ans =
    ParallelException with properties:
        identifier: 'MATLAB:load:couldNotReadFile'
        message: 'Unable to read file 'myData.mat'. No such file or directory.'
        cause: {}
        remotecause: {
            [1×1 MException]
        }
        stack: [1×1 struct]
```

Create the MAT-file referenced from the second task using the `save` function. To create a new job with the tasks that resulted in an error, use the `Tasks` name-value pair in `recreate`, and provide the `hasError` function. If you want to select a different set of tasks, you can define your own function.

```matlab
str = 'Task2';
save myData str
newjob = recreate(job,'Tasks',@hasError);
```

Submit the new job, wait for its completion, and fetch the outputs. Because the MAT-file now exists, the job does not fail.

```matlab
submit(newjob);
wait(newjob);
```
Recreate an Entire Job

This example shows how to recreate the entire job `myJob`.

```matlab
ewJob = recreate(myJob)
```

Recreate a Job with Only Pending Tasks

This example shows how to recreate an independent job, which has only pending tasks from the job `oldIndependentJob`.

```matlab
ewJob = recreate(oldIndependentJob, 'TaskState', 'pending');
```

Recreate a Job with Specified Tasks

This example shows how to recreate an independent job, which has only the tasks with IDs 21 to 32 from the job `oldIndependentJob`.

```matlab
ewJob = recreate(oldIndependentJob, 'TaskID', [21:32]);
```

Recreate Jobs of a Specific User

This example shows how to find and recreate all failed jobs submitted by user Mary. Assume the default cluster is the one Mary had submitted her jobs to.

```matlab
c = parcluster();
failedjobs = findJob(c, 'Username', 'Mary', 'State', 'failed');
for m = 1:length(failedjobs)
```
newJob(m) = recreate(failedjobs(m));
end

Input Arguments

oldjob — Original job
parallel.Job

Original job to be duplicated, specified as a parallel.Job object.
Example: newJob = recreate(oldjob); submit(newJob);
Data Types: parallel.Job

tasksToRecreate — Tasks to duplicate
parallel.Task array | logical array | function handle

Tasks to duplicate from oldjob, specified as:

- An array of parallel.Task belonging to oldjob.
- A 1 x N logical array, where N is the size of oldjob.Tasks, indicating the tasks in oldjob to be recreated.
- A function handle that accepts oldjob.Tasks as an input argument. This function must return a 1 x N logical array indicating the tasks in oldjob to be recreated, where N is the size of oldjob.Tasks.

To rerun tasks containing errors or warnings, use this syntax with the predefined functions @hasError and hasWarnings.
Example: newJob = recreate(oldjob,'Tasks',@hasError | @hasWarnings);
Data Types: parallel.Task | logical | function_handle

states — State of the tasks to duplicate
'pending' | 'running' | 'finished' | 'failed' | cell array with any of the valid states

State of the tasks to duplicate, specified as a string or cell array of strings. states represents the State of the required tasks to recreate from oldjob. Valid states are 'pending', 'running', 'finished', and 'failed'.
Example: newJob = recreate(oldJob,'TaskState','failed');
Data Types: char | string | cell

**ids — IDs of the tasks to duplicate**
vector of integers

IDs of the tasks to duplicate from `oldjob`, specified as a vector of integers.

Example:
```
newJob = recreate(oldIndependentJob,'TaskID',[1 5]);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**See Also**
- `createCommunicatingJob`
- `createJob`
- `createTask`
- `findJob`
- `submit`

**Introduced in R2014a**
redistribute

Redistribute codistributed array with another distribution scheme

Syntax

D2 = redistribute(D1,codist)

Description

D2 = redistribute(D1,codist) redistributes a codistributed array D1 and returns D2 using the distribution scheme defined by the codistributor object codist.

Examples

Redistribute an array according to the distribution scheme of another array.

spmd
    % First, create a magic square distributed by columns:
    M = codistributed(magic(10),codistributor1d(2,[1 2 3 4]));

    % Create a pascal matrix distributed by rows (first dimension):
    P = codistributed(pascal(10),codistributor1d(1));

    % Redistribute the pascal matrix according to the
    % distribution (partition) scheme of the magic square:
    R = redistribute(P,getCodistributor(M));
end

See Also
codistributed | codistributor | codistributor1d.defaultPartition

Introduced in R2006b
reset

Reset GPU device and clear its memory

Syntax

reset(gpudev)

Description

reset(gpudev) resets the GPU device and clears its memory of gpuArray and CUDAKernel data. The GPU device identified by gpudev remains the selected device, but all gpuArray and CUDAKernel objects in MATLAB representing data on that device are invalid.

Arguments

gpudev GPUDevice object representing the currently selected device

Tips

After you reset a GPU device, any variables representing arrays or kernels on the device are invalid; you should clear or redefine them.

Examples

Reset GPU Device

Create a gpuArray on the selected GPU device, then reset the device.
g = gpuDevice(1);
M = gpuArray(magic(4));
M  % Display gpuArray

16   2   3  13
 5  11  10   8
 9   7   6  12
 4  14  15   1

reset(g);
g   % Show that the device is still selected

g =

CUDADevice with properties:

Name: 'Tesla K20c'
Index: 1
ComputeCapability: '3.5'
SupportsDouble: 1
DriverVersion: 7.5
ToolkitVersion: 7.5
MaxThreadsPerBlock: 1024
MaxShmemPerBlock: 49152
MaxThreadBlockSize: [1024 1024 64]
MaxGridSize: [2.1475e+09 65535 65535]
SIMDWidth: 32
TotalMemory: 5.0327e+09
AvailableMemory: 4.9190e+09
MultiprocessorCount: 13
ClockRateKHz: 705500
ComputeMode: 'Default'
GPUOverlapsTransfers: 1
KernelExecutionTimeout: 0
CanMapHostMemory: 1
DeviceSupported: 1
DeviceSelected: 1

whos  % Show that the gpuArray variable name
      % is still in the MATLAB workspace

Name      Size      Bytes  Class     Attributes
---       ----      ------  --------
g        1x1          112  parallel.gpu.CUDADevice
M        1x1          108  gpuArray

M  % Try to display gpuArray
Data no longer exists on the GPU.

clear M

See Also
gpuArray | gpuDevice | parallel.gpu.CUDAKernel

Introduced in R2012a
resume
Resume processing queue in MATLAB job scheduler

Syntax
resume(mjs)

Arguments
mjs MATLAB job scheduler object whose queue is resumed.

Description
resume(mjs) resumes processing of the specified MATLAB job scheduler's queue so that jobs waiting in the queued state will be run. This call will do nothing if the MJS is not paused.

See Also
pause | wait

Introduced before R2006a
saveAsProfile

Save cluster properties to specified profile

Description

saveAsProfile(cluster,profileName) saves the properties of the cluster object to the specified profile, and updates the cluster Profile property value to indicate the new profile name.

Examples

Create a cluster, then modify a property and save the properties to a new profile.

```matlab
myCluster = parcluster('local');
myCluster.NumWorkers = 3;
saveAsProfile(myCluster,'local2'); New profile now specifies 3 workers
```

See Also

parcluster|saveProfile

Introduced in R2012a
saveProfile

Save modified cluster properties to its current profile

Description

saveProfile(cluster) saves the modified properties on the cluster object to the profile specified by the cluster’s Profile property, and sets the Modified property to false. If the cluster’s Profile property is empty, an error is thrown.

Examples

Create a cluster, then modify a property and save the change to the profile.

myCluster = parcluster('local')

myCluster =
Local Cluster
Properties:
     Profile: local
     Modified: false
     Host: HOSTNAME
     NumWorkers: 4

myCluster.NumWorkers = 3

myCluster =
Local Cluster
Properties:
     Profile: local
     Modified: true
     Host: HOSTNAME
     NumWorkers: 3

The myCluster.Modified property is now true.

saveProfile(myCluster);
myCluster
myCluster =
Local Cluster
  Properties:
    Profile: local
    Modified: false
    Host: HOSTNAME
    NumWorkers: 3

After saving, the local profile now matches the current property settings, so the myCluster.Modified property is false.

**See Also**
parcluster|saveAsProfile

**Introduced in R2012a**
setConstantMemory

Set some constant memory on GPU

Syntax

setConstantMemory(kern, sym, val)
setConstantMemory(kern, sym1, val1, sym2, val2, ...)

Description

setConstantMemory(kern, sym, val) sets the constant memory in the CUDA kernel kern with symbol name sym to contain the data in val. val can be any numeric array, including a gpuArray. The command errors if the named symbol does not exist or if it is not big enough to contain the specified data. Partially filling a constant is allowed.

There is no automatic data-type conversion for constant memory, so it is important to make sure that the supplied data is of the correct type for the constant memory symbol being filled.

setConstantMemory(kern, sym1, val1, sym2, val2, ...) sets multiple constant symbols.

Examples

If KERN represents a CUDA kernel whose CU file contains the following includes and constant definitions:

```
#include "tmwtypes.h"
__constant__ int32_t N1;
__constant__ int N2; // Assume 'int' is 32 bits
__constant__ double CONST_DATA[256];
```

you can fill these with MATLAB data as follows:

```
KERN = parallel.gpu.CUDAKernel(ptxFile, cudaFile);
```
setConstantMemory(KERN,'N1',int32(10));
setConstantMemory(KERN,'N2',int32(10));
setConstantMemory(KERN,'CONST_DATA',1:10);

or

setConstantMemory(KERN,'N1',int32(10),'N2',int32(10),'CONST_DATA',1:10);

See Also

gpuArray | parallel.gpu.CUDAKernel

Introduced in R2012a
setJobClusterData

Set specific user data for job on generic cluster

Syntax

setJobClusterData(cluster,job,userdata)

Arguments

- cluster Cluster object identifying the generic third-party cluster running the job
- job Job object identifying the job for which to store data
- userdata Information to store for this job

Description

setJobClusterData(cluster,job,userdata) stores data for the job job that is running on the generic cluster cluster. You can later retrieve the information with the function getJobClusterData. For example, it might be useful to store the third-party scheduler’s external ID for this job, so that the function specified in GetJobStateFcn can later query the scheduler about the state of the job. Or the stored data might be an array with the scheduler’s ID for each task in the job.

For more information and examples on using these functions and properties, see “Integration Scripts for Generic Schedulers” on page 7-21.

See Also

getJobClusterData

Introduced in R2012a
shutdown

Shut down cloud cluster

Syntax

shutdown(cluster)
shutdown(MJScluster,'At',D)
shutdown(MJScluster,'After',event)
shutdown(MJScluster,'After',numhours)

Description

shutdown(cluster) shuts down the cluster immediately.

shutdown(MJScluster,'At',D) shuts down the cluster at the time specified by the datenum, datevec, or datetime D. D is interpreted in the local time zone unless D is a datetime with TimeZone specified.

shutdown(MJScluster,'After',event) shuts down the cluster after the specified event event has occurred. event can be 'never' or 'idle'. A cluster is 'idle' immediately when there are no running jobs, queued jobs, or running pools. The cluster is eligible for shutdown if 'idle' for more than 5 minutes, and is guaranteed to shut down within 60 minutes.

shutdown(MJScluster,'After',numhours) shuts down the cluster after numhours hours, as measured from the time the method is called.

Examples

Shut Down a Cloud Cluster Immediately

shutdown(cluster);
Shut Down Cluster at Date and Time Specified

Specify date and time to terminate a cluster using a datenum, datevec, or datetime. The datetime can (optionally) have a TimeZone specified.

```matlab
shutdown(MJSCluster,'At',datenum('2017-02-22 19:00'));
shutdown(MJSCluster,'At',datevec('2017-02-22 19:00'));
shutdown(MJSCluster,'At',datetime(2017, 2, 22, 19, 0, 0,'TimeZone','local'));
```

Enable Cluster to Run Indefinitely

```matlab
shutdown(MJSCluster,'After','never');
```

Shut Down When Cluster Is Idle

```matlab
shutdown(MJSCluster,'After','idle');
```

Shut Down Cluster After a Number of Hours

```matlab
shutdown(MJSCluster,'After',10);
```

Input Arguments

`cluster` — MATLAB Distributed Computing Server for Amazon EC2 cloud cluster
cluster object (default)

MATLAB Distributed Computing Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`.

Example: `shutdown(cluster);`

`MJScluster` — MATLAB Distributed Computing Server for Amazon EC2 cloud cluster
cluster object (default)

MATLAB Distributed Computing Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`. 
Example: `shutdown(MJScluster);`

**D — Date and time**
datenum | datevec | datetime

Date and time, specified as a datenum, datevec, or datetime. D is interpreted in the local time zone unless D is a datetime with TimeZone specified.

Example: `shutdown(MJSCluster,'At',datenum('2017-02-22 19:00'));`
Example: `shutdown(MJSCluster,'At',datevec('2017-02-22 19:00'));`
Example: `shutdown(MJSCluster,'At',datetime(2017, 2, 22, 19, 0, 0,'TimeZone', 'local'));`

**event — Event to shut down the cluster**
'never' | 'idle'

Event to shut down the cluster, specified as 'never' or 'idle'. A cluster is 'idle' immediately when there are no running jobs, queued jobs, or running pools. The cluster is eligible for shutdown if 'idle' for more than 5 minutes, and is guaranteed to shut down within 60 minutes.

Example: `shutdown(MJSCluster,'After','idle');`

**numhours — Number of hours**
scalar

Number of hours after which the cluster shuts down, specified as scalar, measured from the time you call `shutdown`.

Example: `shutdown(MJSCluster,'After',10);`

**See Also**
datenum | datetime | datevec | parallel.Cluster | parcluster | parpool | start | wait (cluster)

Introduced in R2017a
size

Size of object array

Syntax

d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)

Arguments

obj An object or an array of objects.
dim The dimension of obj.
d The number of rows and columns in obj.
m The number of rows in obj, or the length of the dimension specified by dim.
n The number of columns in obj.
m1,m2,m3,...,mn The lengths of the first n dimensions of obj.

Description

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

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

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

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

length

Introduced before R2006a
sparse

Create sparse distributed or codistributed matrix

Syntax

SD = sparse(FD)
SC = sparse(m,n,codist)
SC = sparse(m,n,codist,'noCommunication')
SC = sparse(i,j,v,m,n,nzmax)
SC = sparse(i,j,v,m,n)
SC = sparse(i,j,v)

Description

SD = sparse(FD) converts a full distributed or codistributed array FD to a sparse distributed or codistributed (respectively) array SD.

SC = sparse(m,n,codist) creates an m-by-n sparse codistributed array of underlying class double, distributed according to the scheme defined by the codistributor codist. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. This form of the syntax is most useful inside spmd, pmode, or a communicating job.

SC = sparse(m,n,codist,'noCommunication') creates an m-by-n sparse codistributed array in the manner specified above, but does not perform any global communication for error checking when constructing the array. This form of the syntax is most useful inside spmd, pmode, or a communicating job.

SC = sparse(i,j,v,m,n,nzmax) uses vectors i and j to specify indices, and v to specify element values, for generating an m-by-n sparse matrix such that SC(i(k),j(k)) = v(k), with space allocated for nzmax nonzeros. If any of the input vectors i, j, or v is codistributed, the output sparse matrix SC is codistributed. Vectors i, j, and v must be the same length. Any elements of v that are zero are ignored, along with the corresponding values of i and j. Any elements of v that have duplicate values of i and j are added together.
To simplify this six-argument call, you can pass scalars for the argument \( v \) and one of the arguments \( i \) or \( j \), in which case they are expanded so that \( i, j, \) and \( v \) all have the same length.

\[
\text{SC} = \text{sparse}(i,j,v,m,n) \text{ uses } \text{nzmax} = \max([\text{length}(i) \text{ length}(j)]).
\]

\[
\text{SC} = \text{sparse}(i,j,v) \text{ uses } m = \max(i) \text{ and } n = \max(j).\text{ The maxima are computed before any zeros in } v \text{ are removed, so one of the rows of } [i \ j \ v] \text{ might be } [m \ n \ 0], \text{ assuring the matrix size satisfies the requirements of } m \text{ and } n.
\]

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

```matlab
spmd
    SC = logical(sparse(m,n,codistributor1d()));
end
```

### Examples

With four workers,

```matlab
spmd(4)
    C = sparse(1000,1000,codistributor1d())
end
```

creates a 1000-by-1000 codistributed sparse double array \( C \). \( C \) is distributed by its second dimension (columns), and each worker contains a 1000-by-250 local piece of \( C \).

```matlab
spmd(4)
    codist = codistributor1d(2,1:numlabs)
    C = sparse(10,10,codist);
end
```

creates a 10-by-10 codistributed sparse double array \( C \), distributed by its columns. Each worker contains a 10-by-\text{labindex} local piece of \( C \).

Convert a distributed array into a sparse distributed array:

```matlab
R = rand(1000,'distributed');
D = floor(2*R); % D also is distributed
SD = sparse(D); % SD is sparse distributed
```
Create a sparse codistributed array from vectors of indices and a distributed array of element values:

```matlab
r = [1 1 4 4 8];
c = [1 4 1 4 8];
v = [10 20 30 40 0];
V = distributed(v);
spmd
    SC = sparse(r,c,V);
end
```

In this example, even though the fifth element of the value array \( v \) is 0, the size of the result is an 8-by-8 matrix because of the corresponding maximum indices in \( r \) and \( c \). Matrix \( SC \) is considered codistributed when viewed inside an \texttt{spmd} block, and distributed when viewed from the client workspace. To view a full version of the matrix, the \texttt{full} function converts this distributed sparse array to a full distributed array:

```matlab
S = full(SC)
```

```
10  0  0  20  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
30  0  0  40  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
```

**See Also**

codistributed.spalloc | distributed.spalloc | sparse

**Introduced in R2006b**


**spmd**

Execute code in parallel on workers of parallel pool

**Syntax**

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

**Description**

The general form of an `spmd` (single program, multiple data) statement is:

```
spmd
    statements
end
```

`spmd, statements, end` defines an `spmd` statement on a single line. MATLAB executes the `spmd` body denoted by `statements` on several MATLAB workers simultaneously. The `spmd` statement can be used only if you have Parallel Computing Toolbox. To execute the statements in parallel, you must first open a pool of MATLAB workers using `parpool` or have your parallel preferences allow the automatic start of a pool.

Inside the body of the `spmd` statement, each MATLAB worker has a unique value of `labindex`, while `numlabs` denotes the total number of workers executing the block in parallel. Within the body of the `spmd` statement, communication functions for communicating jobs (such as `labSend` and `labReceive`) can transfer data between the workers.

Values returning from the body of an `spmd` statement are converted to Composite objects on the MATLAB client. A Composite object contains references to the values stored on the remote MATLAB workers, and those values can be retrieved using cell-array indexing. The actual data on the workers remains available on the workers for subsequent `spmd` execution, so long as the Composite exists on the client and the parallel pool remains open.
By default, MATLAB uses all workers in the pool. When there is no pool active, MATLAB will create a pool and use all the workers from that pool. If your preferences do not allow automatic pool creation, MATLAB executes the block body locally and creates Composite objects as necessary. You cannot execute an `spmd` block if any worker is busy executing a `parfeval` request, unless you use `spmd(0)`.

`spmd(n), statements, end` uses `n` to specify the exact number of MATLAB workers to evaluate `statements`, provided that `n` workers are available from the parallel pool. If there are not enough workers available, an error is thrown. If `n` is zero, MATLAB executes the block body locally and creates Composite objects, the same as if there is no pool available.

`spmd(m,n), statements, end` uses a minimum of `m` and a maximum of `n` workers to evaluate `statements`. If there are not enough workers available, an error is thrown. `m` can be zero, which allows the block to run locally if no workers are available.

For more information about `spmd` and Composite objects, see “Distribute Arrays and Run SPMD” on page 1-16.

**Note** Use `parfevalOnAll` instead of `parfor` or `spmd` if you want to use `clear`. This preserves workspace transparency. See “Ensure Transparency in parfor-Loops” on page 2-58.

**Examples**

Perform a simple calculation in parallel, and plot the results:

```matlab
parpool(3)
spmd
    % build magic squares in parallel
    q = magic(labindex + 2);
end
for ii=1:length(q)
    % plot each magic square
    figure, imagesc(q{ii});
end
delete(gcp)
```
**Tips**

- An `spmd` block runs on the workers of the existing parallel pool. If no pool exists, `spmd` will start a new parallel pool, unless the automatic starting of pools is disabled in your parallel preferences. If there is no parallel pool and `spmd` cannot start one, the code runs serially in the client session.
- If the `AutoAttachFiles` property in the cluster profile for the parallel pool is set to `true`, MATLAB performs an analysis on an `spmd` block to determine what code files are necessary for its execution, then automatically attaches those files to the parallel pool job so that the code is available to the workers.
- For information about restrictions and limitations when using `spmd`, see “Run Single Programs on Multiple Data Sets” on page 3-2.

**See Also**

`Composite | batch | gop | labindex | numlabs | parallel.pool.Constant | parpool`

*Introduced in R2008b*
**start**

Start cloud cluster

**Syntax**

```matlab
start(cluster)
```

**Description**

`start(cluster)` starts the specified MATLAB Distributed Computing Server for Amazon EC2 cluster, if it is not already running. If the cluster is already running or in the process of shutting down, then `start(cluster)` returns immediately, and the state of the cluster is not changed.

**Examples**

**Start Cloud Cluster**

Obtain your cluster profile using one of the following ways:

- From the MATLAB **Parallel > Discover Clusters** user interface. For more information, see “Discover Clusters and Use Cluster Profiles” on page 6-15.
- By downloading it from Cloud Center. For more information, see MathWorks Cloud Center documentation.

Create a cluster using the default profile.

```matlab
myCluster = parcluster;
```

Start the cluster:

```matlab
start(myCluster);
```

Wait for the cluster to be ready to accept job submissions.
wait(myCluster);

### Input Arguments

**cluster** — MATLAB Distributed Computing Server for Amazon EC2 cluster  
cluster object (default)

MATLAB Distributed Computing Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`.

Example: `start(cluster)`

### See Also

`parallel.Cluster` | `parcluster` | `parpool` | `shutdown` | `wait (cluster)`

Introduced in R2017a
**submit**

Queue job in scheduler

**Syntax**

`submit(j)`

**Arguments**

- **j**: Job object to be queued.

**Description**

`submit(j)` queues the job object `j` in its cluster queue. The cluster used for this job was determined when the job was created.

**Examples**

Find the MJS cluster identified by the cluster profile `Profile1`.

```matlab
c1 = parcluster('Profile1');
```

Create a job object in this cluster.

```matlab
j1 = createJob(c1);
```

Add a task object to be evaluated for the job.

```matlab
t1 = createTask(j1,@rand,1,{8,4});
```

Queue the job object in the cluster for execution.

```matlab
submit(j1);
```
**Tips**

When a job is submitted to a cluster queue, the job’s State property is set to queued, and the job is added to the list of jobs waiting to be executed.

The jobs in the waiting list are executed in a first in, first out manner; that is, the order in which they were submitted, except when the sequence is altered by promote, demote, cancel, or delete.

**See Also**
createCommunicatingJob | createJob | findJob | parcluster | promote | recreate

*Introduced before R2006a*
subsasgn

Subscripted assignment for Composite

Syntax

C(i) = {B}
C(1:end) = {B}
C([i1,i2]) = {B1,B2}
C{i} = B

Description

subsasgn assigns remote values to Composite objects. The values reside on the workers in the current parallel pool.

C(i) = {B} sets the entry of C on worker i to the value B.

C(1:end) = {B} sets all entries of C to the value B.

C([i1,i2]) = {B1,B2} assigns different values on workers i1 and i2.

C{i} = B sets the entry of C on worker i to the value B.

See Also

Composite|subsasgn|subsref

Introduced in R2008b
**subsref**

Subscripted reference for Composite

**Syntax**

\[
B = C(i)
\]

\[
B = C([i_1,i_2,\ldots])
\]

\[
B = C\{i\}
\]

\[
[B_1,B_2,\ldots] = C\{[i_1,i_2,\ldots]\}
\]

**Description**

`subsref` retrieves remote values of a Composite object from the workers in the current parallel pool.

- \( B = C(i) \) returns the entry of Composite \( C \) from worker \( i \) as a cell array.
- \( B = C([i_1,i_2,\ldots]) \) returns multiple entries as a cell array.
- \( B = C\{i\} \) returns the value of Composite \( C \) from worker \( i \) as a single entry.
- \( [B_1,B_2,\ldots] = C\{[i_1,i_2,\ldots]\} \) returns multiple entries.

**See Also**

Composite | subsasgn | subsref

**Introduced in R2008b**
**taskFinish**

User-defined options to run on worker when task finishes

**Syntax**

taskFinish(task)

**Arguments**

task  The task being evaluated by the worker

**Description**

taskFinish(task) runs automatically on a worker each time the worker finishes evaluating a task for a particular job. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the taskFinish.m file to define anything you want executed on the worker when a task is finished. The worker looks for taskFinish.m in the following order, executing the one it finds first:

1. Included in the job’s AttachedFiles property.
2. In a folder included in the job’s AdditionalPaths property.
3. In the worker’s MATLAB installation at the location `matlabroot/toolbox/distcomp/user/taskFinish.m`

To create a version of taskFinish.m for AttachedFiles or AdditionalPaths, copy the provided file and modify it as required. For further details on taskFinish and its implementation, see the text in the installed taskFinish.m file.

**See Also**

jobStartup | poolStartup | taskStartup
Introduced before R2006a
**taskStartup**

User-defined options to run on worker when task starts

**Syntax**

taskStartup(task)

**Arguments**

task The task being evaluated by the worker.

**Description**

taskStartup(task) runs automatically on a worker each time the worker evaluates a task for a particular job. You do not call this function from the client session, nor explicitly as part of a task function.

You add MATLAB code to the `taskStartup.m` file to define task initialization on the worker. The worker looks for `taskStartup.m` in the following order, executing the one it finds first:

1. Included in the job’s `AttachedFiles` property.
2. In a folder included in the job’s `AdditionalPaths` property.
3. In the worker’s MATLAB installation at the location `matlabroot/toolbox/distcomp/user/taskStartup.m`

To create a version of `taskStartup.m` for `AttachedFiles` or `AdditionalPaths`, copy the provided file and modify it as required. For further details on `taskStartup` and its implementation, see the text in the installed `taskStartup.m` file.

**See Also**

`jobStartup` | `poolStartup` | `taskFinish`
Introduced before R2006a
**send**

Send data from worker to client using a data queue

**Syntax**

```matlab
send(queue, data)
send(pollablequeue, data)
```

**Description**

`send(queue, data)` sends a message or data with the value `data` to the `parallel.pool.DataQueue` specified by `queue`. Call `afterEach` to pass each of the pending messages to the function specified by `afterEach`.

`send(pollablequeue, data)` sends a message or data with the value `data` to the `parallel.pool.PollableDataQueue` specified by `pollablequeue`. Retrieve the result using `poll(pollablequeue)`, and return `data` as the answer.

Use the `send` and `poll` functions together using a pollable data queue to transfer and retrieve messages or data from different workers.

**Examples**

**Send a Message in a parfor-Loop, and Dispatch the Message on the Queue**

Construct a `DataQueue`, and call `afterEach`.

```matlab
q = parallel.pool.DataQueue;
afterEach(q, @disp);
```

Start a `parfor`-loop, and send a message. The pending message is passed to the `afterEach` function, in this example `@disp`. 
\begin{verbatim}
parfor i = 1:3
    send(q, i);
end;

For more details on listening for data using a DataQueue, see afterEach.
\end{verbatim}

**Send a Message in a parfor-loop, and Poll for the Result**

Construct a PollableDataQueue.

```matlab
p = parallel.pool.PollableDataQueue;
```

Start a parfor-loop, and send a message, such as data with the value 1.

```matlab
parfor i = 1
    send(p, i);
end
```

Poll for the result.

```matlab
poll(p)
```

For more details on retrieving data using a PollableDataQueue, see poll.

**Construct a Simple parfor Wait Bar Using a Data Queue**

This example shows a function that creates a parfor wait bar. Create a DataQueue, and use afterEach to specify the function to execute each time the queue receives data. This example calls a subfunction that updates the wait bar.

Create a parfor-loop to carry out a computationally demanding task in MATLAB. Use send to send some dummy data on each iteration of the parfor-loop. When the queue receives the data, afterEach calls nUpdateWaitbar in the client MATLAB, and you can observe the wait bar progress.
function a = parforWaitbar

D = parallel.pool.DataQueue;

h = waitbar(0, 'Please wait ...');

afterEach(D, @nUpdateWaitbar);

N = 200;
P = 1;

parfor i = 1:N
    a(i) = max(abs(eig(rand(400))));
    send(D, i);
end

function nUpdateWaitbar(~)
    waitbar(P/N, h);
    P = P + 1;
end
end

Input Arguments

queue — Data queue
parallel.pool.DataQueue

Data queue, specified as a parallel.pool.DataQueue object.

Example: q = parallel.pool.DataQueue;

data — Message or data
scalar | vector | matrix | array | string | character vector | serializable object

Message or data from workers to a data queue, specified as any data type that can be serialized.
Example: send(queue, data);

pollablequeue — Pollable data queue
parallel.pool.PollableDataQueue

Pollable data queue, specified as a parallel.pool.PollableDataQueue object.
Example: p = parallel.pool.PollableDataQueue;

See Also
afterEach | parallel.pool.DataQueue | parallel.pool.PollableDataQueue | parfor | parpool | poll

Introduced in R2017a
ticBytes

Start counting bytes transferred within parallel pool

Syntax

ticBytes(pool)
startState = ticBytes(pool)

Description

ticBytes(pool) starts counting the number of bytes transferred to each worker in the pool, so that later tocBytes(pool) can measure the amount of data transferred to each worker between the two calls.

Use the ticBytes(pool) and tocBytes(pool) functions together to measure how much data is transferred to and from the workers in a parallel pool. You can use ticBytes and tocBytes while executing parallel language constructs and functions, such as parfor, spmd, or parfeval. Use ticBytes and tocBytes to pass around less data and optimize your code.

startState = ticBytes(pool) saves the state to an output argument, startState, so that you can simultaneously record the number of bytes transferred for multiple pairs of ticBytes and tocBytes calls. Use the value of startState as an input argument for a subsequent call to tocBytes.

Examples

Measure Amount of Data Transferred While Running a Simple parfor-loop

a = 0;
b = rand(100);
ticBytes(gcp);
parfor i = 1:100
    a = a + sum(b(:, i));
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

<table>
<thead>
<tr>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42948</td>
</tr>
<tr>
<td>2</td>
<td>36548</td>
</tr>
<tr>
<td>3</td>
<td>27500</td>
</tr>
<tr>
<td>4</td>
<td>27500</td>
</tr>
<tr>
<td>Total</td>
<td>1.345e+05</td>
</tr>
<tr>
<td></td>
<td>23312</td>
</tr>
</tbody>
</table>

Workers might transfer different numbers of bytes, because each worker might carry out different numbers of loop iterations.

**Simultaneously Measure Multiple Amounts of Data Transferred, Using Two Pairs of ticBytes and tocBytes Calls**

Measure the minimum and average number of bytes transferred while running a parfor loop nested in a for loop.

REPS = 10;
minBytes = Inf;
ticBytes(gcp); % ticBytes, pair 1

for ii=1:REPS
    a = 0;
    b = rand(100);
    startS = ticBytes(gcp) % ticBytes, pair 2
    parfor i = 1:100
        a = a + sum(b(:, i));
    end
    bytes = tocBytes(gcp, startS) % tocBytes, pair 2
    minBytes = min(bytes, minBytes)
end

averageBytes = tocBytes(gcp)/REPS % tocBytes, pair 1
Note that nesting a parfor-loop in a for-loop can be slow due to overhead, see “Convert Nested for-Loops to parfor” on page 2-16.

**Input Arguments**

`pool` — parallel pool

*parallel.Pool*

Parallel pool, typically specified by `gcp`, if you want the current parallel pool. Otherwise, use `parpool` to create a new pool.

Example: `ticBytes(gcp);`

**Output Arguments**

`startState` — Starting state

*TicBytesResult*

Starting state returned as an input argument for a subsequent call to `tocBytes`.

Example: `startState = ticBytes(gcp);`

**See Also**

`gcp` | `parfeval` | `parfor` | `parpool` | `spmd` | `tocBytes`

*Introduced in R2016b*
**tocBytes**

Read how many bytes have been transferred since calling tocBytes.

**Syntax**

```
tocBytes(pool)
bytes = tocBytes(pool)
tocBytes(pool,startState)
bytes = tocBytes(pool,startState)
```

**Description**

tocBytes(pool) reads how many bytes have been transferred since calling ticBytes. The function displays the total number of bytes transferred to and from each of the workers in a parallel pool after the most recent execution of ticBytes.

Use the ticBytes (pool) and tocBytes (pool) functions together to measure how much data is transferred to and from the workers in a parallel pool. You can use ticBytes and tocBytes while executing parallel language constructs and functions, such as parfor, spmd, or parfeval. Use ticBytes and tocBytes to pass around less data and optimize your code.

```
bytes = tocBytes(pool) returns the number of bytes transferred to and from each of the workers in the parallel pool.

tocBytes(pool,startState) displays the total number of bytes transferred in the parallel pool after the ticBytes command that generated startState.

bytes = tocBytes(pool,startState) returns the number of bytes transferred to and from each of the workers in the parallel pool after the ticBytes command that generated startState.
```

**Examples**
Measure Amount of Data Transferred While Running a Simple parfor-loop

Use `tocBytes(gcp,startS)` to measure the amount of data transferred.

```matlab
a = 0;
b = rand(100);
startS = ticBytes(gcp);
parfor i = 1:100
    a = a + sum(b(:, i));
end
tocBytes(gcp,startS)
```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

<table>
<thead>
<tr>
<th>BytesSentToWorkers</th>
<th>BytesReceivedFromWorkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42948</td>
</tr>
<tr>
<td>2</td>
<td>36548</td>
</tr>
<tr>
<td>3</td>
<td>27500</td>
</tr>
<tr>
<td>4</td>
<td>27500</td>
</tr>
<tr>
<td>Total</td>
<td>1.345e+05</td>
</tr>
<tr>
<td></td>
<td>23312</td>
</tr>
</tbody>
</table>

Workers might transfer different numbers of bytes, because each worker might carry out different numbers of loop iterations.

Measure Amount of Data Transferred While Running a Simple spmd Block

Use `bytes = tocBytes(gcp)` to measure the amount of data transferred.

```matlab
ticBytes(gcp);
spmd
    rand(100);
end
bytes = tocBytes(gcp)
```

`bytes =

```
13448  1208
13448  1208
13448  1208
13448  1208
```
Workers transfer the same number of bytes, because each worker carries out the same number of loop iterations.

**Simultaneously Measure Multiple Amounts of Data Transferred, Using Two Pairs of ticBytes and tocBytes Calls**

Measure the minimum and average number of bytes transferred while running a parfor loop nested in a for loop.

```matlab
REPS = 10;
minBytes = Inf;
ticBytes(gcp);  % ticBytes, pair 1
for ii=1:REPS
    a = 0;
    b = rand(100);
    startS = ticBytes(gcp)  % ticBytes, pair 2
    parfor i = 1:100
        a = a + sum(b(:, i));
    end
    bytes = tocBytes(gcp, startS)  % tocBytes, pair 2
    minBytes = min(bytes, minBytes)
end
averageBytes = tocBytes(gcp)/REPS  % tocBytes, pair 1
```

Note that nesting a parfor-loop in a for-loop can be slow due to overhead, see “Convert Nested for-Loops to parfor” on page 2-16.

**Input Arguments**

- **pool — parallel pool**
  `parallel.Pool`

  Parallel pool, typically specified by `gcp`, if you want the current parallel pool. Otherwise, use `parpool` to create a new pool.

  Example: tocBytes(gcp);
startState — Starting state
TicBytesResult

Starting state returned by ticBytes(pool).
Example: startState = ticBytes(gcp);

Output Arguments

bytes — Bytes transferred
tocBytes(pool)

Bytes transferred, returned as a matrix of size numWorkers x 2. This matrix contains the number of bytes transferred to and from each of the workers in the parallel pool. bytes returns values in bytes without headings. Use tocBytes(pool) without an output argument to get Sent and Received headings, worker numbers, and values in bytes in the Command Window output.
Example: bytes = tocBytes(pool);

See Also
gcp | parfeval | parfor | parpool | spmd | ticBytes

Introduced in R2016b
true

Array of logical 1 (true)

Syntax

T = true(sz,arraytype)

T = true(sz,'like',P)

C = true(sz,codist)
C = true(sz,___,codist,'noCommunication')
C = true(sz,___,codist,'like',P)

Description

T = true(sz,arraytype) creates a matrix with true values in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>n</td>
<td>Specifies size as an n-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n or [m n]</td>
<td>Specifies size as an m-by-n matrix.</td>
</tr>
<tr>
<td></td>
<td>m,n,...,k or [m n ... k]</td>
<td>Specifies size as an m-by-n-by-...-by-k array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
<tr>
<td></td>
<td>'codistributed'</td>
<td>Specifies codistributed array, using the default distribution scheme.</td>
</tr>
<tr>
<td></td>
<td>'gpuArray'</td>
<td>Specifies gpuArray.</td>
</tr>
</tbody>
</table>

T = true(sz,'like',P) creates an array of true values with the same type as array P.
C = true(sz,codist) creates a codistributed array of true values with the specified size. The codistributor object codist specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```matlab
spmd
  C = true(8,codistributor1d());
end
```

C = true(sz,___,codist,'noCommunication') specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

C = true(sz,___,codist,'like',P) creates a codistributed array of true values with the specified size and distribution scheme. If the codistributor argument is omitted, the distribution scheme is taken from the codistributed array P.

### Examples

#### Create Distributed True Matrix

Create a 1000-by-1000 distributed array of true values with underlying class double:

```matlab
D = true(1000,'distributed');
```

#### Create Codistributed True Matrix

Create a 1000-by-1000 codistributed matrix of true values, distributed by its second dimension (columns).

```matlab
spmd(4)
  C = true(1000,'codistributed');
end
```

With four workers, each worker contains a 1000-by-250 local piece of C.

Create a 1000-by-1000 codistributed matrix of true values, distributed by its columns.
Example:

```matlab
spmd(4)
    codist = codistributor('1d',2,100*[1:numlabs]);
    C = true(1000,1000,codist);
end
```

Each worker contains a 100-by-labindex local piece of C.

**Create gpuArray True Matrix**

Create a 1000-by-1000 gpuArray of trues:

```matlab
G = true(1000,'gpuArray');
```

**See Also**

Inf | NaN | eye | false | ones | true | zeros

**Introduced in R2006b**
updateAttachedFiles

Update attached files or folders on parallel pool

Syntax

updateAttachedFiles(poolobj)

Description

updateAttachedFiles(poolobj) checks all the attached files of the specified parallel pool to see if they have changed, and replicates any changes to each of the workers in the pool. This checks files that were attached (by a profile or parpool argument) when the pool was started and those subsequently attached with the addAttachedFiles command.

Examples

Update Attached Files on Current Parallel Pool

Update all attached files on the current parallel pool.

```matlab
poolobj = gcp;
updateAttachedFiles(poolobj)
```

Input Arguments

poolobj — Pool with attached files
pool object

Pool with attached files, specified as a pool object.

Example: poolobj = gcp;
See Also
addAttachedFiles | gcp | listAutoAttachedFiles | parpool

Topics
“Add and Modify Cluster Profiles” on page 6-20

Introduced in R2013b
**wait**

Wait for job to change state

**Syntax**

```matlab
wait(j)
wait(j,state)
OK = wait(j,state,timeout)
```

**Arguments**

- **j**  
  Job object whose change in state to wait for.
- **state**  
  Value of the job object’s *State* property to wait for.
- **timeout**  
  Maximum time to wait, in seconds.

**Description**

`wait(j)` blocks execution in the client session until the job identified by the object `j` reaches the 'finished' state or fails. This occurs when all the job’s tasks are finished processing on the workers.

`wait(j,state)` blocks execution in the client session until the specified job object changes state to the value of `state`. The valid states to wait for are 'queued', 'running', and 'finished'.

If the object is currently or has already been in the specified state, a wait is not performed and execution returns immediately. For example, if you execute `wait(j,'queued')` for a job already in the 'finished' state, the call returns immediately.

`OK = wait(j,state,timeout)` blocks execution until either the job reaches the specified state, or `timeout` seconds elapse, whichever happens first. `OK` is `false` if `timeout` is exceeded before `state` is reached.
Note Simulink models cannot run while a MATLAB session is blocked by `wait`. If you must run Simulink from the MATLAB client while also running jobs, you cannot use `wait`.

Examples

Submit a job to the queue, and wait for it to finish running before retrieving its results.

```matlab
submit(j);
wait(j,'finished')
results = fetchOutputs(j)
```

Submit a batch job and wait for it to finish before retrieving its variables.

```matlab
j = batch('myScript');
wait(j)
load(j)
```

See Also

`pause` | `resume` | `wait (FevalFuture)` | `wait (GPUDevice)`

Introduced in R2008a
**wait (cluster)**

Wait for cloud cluster to change state

**Syntax**

```matlab
wait(cluster)
wait(cluster,state)
OK = wait(cluster,state,timeout)
```

**Description**

`wait(cluster)` blocks execution in the client MATLAB session until `cluster` reaches the 'online' state. The 'online' state indicates that the cluster is running and you can use all requested workers to run jobs.

`wait(cluster,state)` blocks execution in the client session until `cluster` changes state. For a cluster object, the valid states are:

- 'online': The cluster is running and you can use all requested workers to run jobs.
- 'waitingforworkers': The cluster is running, and you can use some but not all of the requested workers to run jobs. You can still use the cluster in this state with the workers that are available.
- 'offline': The cluster is not running, but you can restart using the `start()` command or via https://cloudcenter.mathworks.com. If the cluster has shared persisted storage, then any previous jobs in the queue are still present when you restart the cluster.

`OK = wait(cluster,state,timeout)` blocks execution in the client session until `cluster` changes state, or until `timeout` seconds have elapsed, whichever happens first. `OK` is **true** if `state` has been reached or a terminal state such as 'error' occurs. `OK` is **false** in case of a timeout.

**Examples**
Wait Until the Cluster Is Running

In Cluster Profile Manager, select MATLAB Distributed Computing Server for Amazon EC2 as your default cluster profile.

Create and start a cloud cluster using the default profile.

```matlab
cluster = parcluster;
start(cluster);
```

Wait until the cluster is running. Use all requested workers to run jobs.

```matlab
wait(cluster,'online');
```

Wait for Specified Time for Cluster to Start

In Cluster Profile Manager, select MATLAB Distributed Computing Server for Amazon EC2 as your default cluster profile.

Create and start a cloud cluster using the default profile.

```matlab
cluster = parcluster;
start(cluster);
```

Wait 100 seconds for the head node and all workers to start.

```matlab
OK = wait(cluster,'online',100);
```

Input Arguments

- `cluster` — MATLAB Distributed Computing Server for Amazon EC2 cloud cluster
  
  A cluster object (default)

  MATLAB Distributed Computing Server for Amazon EC2 cluster, specified as cluster object created using `parcluster`.

  Example: `wait(cluster);`

- `state` — cloud cluster state
  
  'online' | 'waitingforworkers' | 'offline'
Cloud cluster state, specified as a cluster object, for which the valid states are 'online', 'waitingforworkers', and 'offline'.

Example: wait(cluster,'online');

**timeout** — time elapsed before cloud cluster changes state
seconds

Time elapsed before cloud cluster changes state, specified in seconds.
Example: wait(cluster,'online',100);

**Output Arguments**

**OK** — check if state has been reached

Boolean

Check if state has been reached, specified as a Boolean. OK is true if state has been reached or a terminal state such as 'error' occurs. OK is false in case of a timeout.
Example: OK = wait(cluster,'waitingforworkers',10);

**See Also**

parallel.Cluster | parcluster | parpool | shutdown | start

Introduced in R2017a
**wait (FevalFuture)**

Wait for futures to complete

**Syntax**

```matlab
wait(F)
wait(F,state)
OK = wait(F,state,timeout)
```

**Description**

`wait(F)` blocks execution until each element of the array of futures `F` has reached the 'finished' state.

`wait(F,state)` blocks execution until each element of the array of futures `F` has reached the state `state`. Valid values for `state` are 'running' or 'finished'.

`OK = wait(F,state,timeout)` blocks execution for a maximum of `timeout` seconds. `OK` is false if `timeout` is exceeded before `state` is reached.

**See Also**

`fetchNext` | `fetchOutputs` | `isequal` | `parfeval` | `parfevalOnAll`

**Introduced in R2013b**
wait (GPUDevice)

Wait for GPU calculation to complete

Syntax

wait(gpudev)

Description

wait(gpudev) blocks execution in MATLAB until the GPU device identified by the GPUDevice object gpudev completes its calculations. This can be used before calls to toc when timing GPU code that does not gather results back to the workspace. When gathering results from a GPU, MATLAB automatically waits until all GPU calculations are complete, so you do not need to explicitly call wait in that situation.

See Also

gather | gpuArray | gpuDevice | gputimeit

Topics

“Measure Performance on the GPU” on page 9-44

Introduced in R2014b
write

Write distributed data to an output location

Syntax

write(location, D)
write(filepattern,D)
write(___ ,Name,Value)

Description

write(location, D) writes the values in the distributed array D to files in the folder location. The data is stored in an efficient binary format suitable for reading back using datastore(location). If not distributed along the first dimension, MATLAB redistributes the data before writing, so that the resulting files can be reread using datastore.

write(filepattern,D) uses the file extension from filepattern to determine the output format. filepattern must include a folder to write the files into followed by a file name that includes a wildcard *. The wildcard represents incremental numbers for generating unique file names, for example write('folder/myfile_*.csv',D).

write(___ ,Name,Value) specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify the file type with 'FileType' and a valid file type ('mat', 'seq', 'text', or 'spreadsheet'), or you can specify a custom write function to process the data with 'WriteFcn' and a function handle.

Examples

Write Distributed Arrays

This example shows how to write a distributed array to a file system, then read it back using a datastore.
Create a distributed array and write it to an output folder.

d = distributed.rand(5000,1);
location = 'hdfs://myHadoopCluster/some/output/folder';
write(location, d);

Recreate the distributed array from the written files.

ds = datastore(location);
d1 = distributed(ds);

**Write Distributed Arrays Using File Patterns**

This example shows how to write distributed arrays to different formats using a file pattern.

Create a distributed table and write it to a simple text-based format that many applications can read.

dt = distributed(array2table(rand(5000,3)));
location = '/tmp/CSVData/dt_*.csv';
write(location, dt);

Recreate the distributed table from the written files.

ds = datastore(location);
dt1 = distributed(ds);

**Write and Read Back Tall and Distributed Data**

You can write distributed data and read it back as tall data and vice versa.

Create a distributed timetable and write it to disk.

dt = distributed(array2table(rand(5000,3)));
location = '/tmp/CSVData/dt_*.csv';
write(location, dt);

Build a tall table from the written files.

ds = datastore(location);
tt = tall(ds);
Alternatively, you can read data written from tall data into distributed data. Create a tall timetable and write it to disk.

```matlab
tt = tall(array2table(rand(5000,3)));
location = "~/tmp/CSVData/dt_*.csv";
write(location, tt);
```

Read back into a distributed timetable.

```matlab
ds = datastore(location);
dt = distributed(ds);
```

### Write Distributed Arrays Using a Write Function

This example shows how to write distributed arrays to a file system using a custom write function.

Create a simple write function that writes out spreadsheet files.

```matlab
function dataWriter(info, data)
    filename = info.SuggestedFilename;
    writetable(data, filename, "FileType", "spreadsheet");
end
```

Create a distributed table and write it to disk using the custom write function.

```matlab
dt = distributed(array2table(rand(5000,3)));
location = "~/tmp/MyData/tt_*.xlsx";
write(location, dt, "WriteFcn", @dataWriter);
```

### Input Arguments

- **location** — Folder location to write data
  
  character vector | string

  Folder location to write data, specified as a character vector or string. `location` can specify a full or relative path. The specified folder can be either of these options:

  - Existing empty folder that contains no other files
  - New folder that `write` creates
write supports remote locations, such as Amazon S3™, Windows Azure® Storage Blob, or a Hadoop Distributed File System (HDFS). For more information, see “Work with Remote Data” (MATLAB).

Example: `location = 'c:/..//dir/data'`
Example: `location = 'C:\Users\MyName\Desktop'`
Example: `location = 'hdfs://myHadoopCluster/some/output/folder'`
Example: `location = 's3://bucketname/some/output/folder'`

Data Types: `char` | `string`

**D — Input array**
distributed array

Input array, specified as a distributed array.

**filepattern — File naming pattern**
string | character vector

File naming pattern, specified as a string or a character vector. The file naming pattern must contain a folder to write the files into followed by a file name that includes a wildcard *. `write` replaces the wildcard with sequential numbers to ensure unique file names.

Example: `write('folder/data_*.txt', D)` writes the distributed array D as a series of `.txt` files in folder with the file names `data_1.txt`, `data_2.txt`, and so on.

Data Types: `char` | `string`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `write('C:\myData', D, 'FileType', 'text', 'WriteVariableNames', false)` writes the distributed array D to `C:\myData` as a collection of text files that do not use variable names as column headings.
General Options

**FileType — Type of file**

'auto' (default) | 'mat' | 'seq' | 'text' | 'spreadsheet'

Type of file, specified as the comma-separated pair consisting of 'FileType' and one of the allowed file types: 'auto', 'mat', 'seq', 'text', or 'spreadsheet'.

Use the 'FileType' name-value pair with the location argument to specify what type of files to write. By default, write attempts to automatically detect the proper file type. You do not need to specify the 'FileType' name-value pair argument if write can determine the file type from an extension in the location or filepattern arguments. write can determine the file type from these extensions:

- .mat for MATLAB data files
- .seq for sequence files
- .txt, .dat, or .csv for delimited text files
- .xls, .xlsx, .xlsb, .xlsm, .xltx, or .xltm for spreadsheet files

Example: `write('C:\myData', D, 'FileType', 'text')`

**WriteFcn — Custom writing function**

function handle

Custom writing function, specified as the comma-separated pair consisting of 'WriteFcn' and a function handle. The specified function receives blocks of data from D and is responsible for creating the output files. You can use the 'WriteFcn' name-value pair argument to write data in a variety of formats, even if the output format is not directly supported by write.

**Functional Signature**

The custom writing function must accept two input arguments, info and data:

```matlab
function myWriter(info, data)
```

- data contains a block of data from D.
- info is a structure with fields that contain information about the block of data. You can use the fields to build a new file name that is globally unique within the final location. The structure fields are:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequiredLocation</td>
<td>Fully qualified path to a temporary output folder. All output files must be written to this folder.</td>
</tr>
<tr>
<td>RequiredFilePattern</td>
<td>The file pattern required for output file names. This field is empty if only a folder name is specified.</td>
</tr>
<tr>
<td>SuggestedFilename</td>
<td>A fully qualified, globally unique file name that meets the location and naming requirements.</td>
</tr>
<tr>
<td>PartitionIndex</td>
<td>Index of the distributed array partition being written.</td>
</tr>
<tr>
<td>NumPartitions</td>
<td>Total number of partitions in the distributed array.</td>
</tr>
<tr>
<td>BlockIndexInPartition</td>
<td>Position of current data block within the partition.</td>
</tr>
<tr>
<td>IsFinalBlock</td>
<td>true if current block is the final block of the partition.</td>
</tr>
</tbody>
</table>

**File Naming**

The file name used for the output files determines the order that the files are read back in later by datastore. If the order of the files matters, then the best practice is to use the SuggestedFilename field to name the files since the suggested name guarantees the file order. If you do not use the suggested file name, the custom writing function must create globally unique, correctly ordered file names. The file names should follow the naming pattern outlined in RequiredFilePattern. The file names must be unique and correctly ordered between workers, even though each worker writes to its own local folder.

**Arrays with Multiple Partitions**

A distributed array is divided into partitions to facilitate running calculations on the array in parallel with Parallel Computing Toolbox. When writing a distributed array, each of the partitions is divided in smaller blocks.

info contains several fields related to partitions: PartitionIndex, NumPartitions, BlockIndexInPartition, and IsFinalBlock. These fields are useful when you are writing out a single file and appending to it, which is a common task for arrays with large
partitions that have been split into many blocks. The custom writing function is called once per block, and the blocks in one partition are always written in order on one worker. However, different partitions can be written by different workers.

**Example Function**

A simple writing function that writes out spreadsheet files is:

```matlab
function dataWriter(info, data)
    filename = info.SuggestedFilename;
    writetable(data, filename, 'FileType', 'spreadsheet')
end
```

To invoke `dataWriter` as the writing function for some data `D`, use the commands:

```matlab
D = distributed(array2table(rand(5000,3)));
location = '/tmp/MyData/D_*.xlsx';
write(location, D, 'WriteFcn', @dataWriter);
```

For each block, the `dataWriter` function uses the suggested file name in the `info` structure and calls `writetable` to write out a spreadsheet file. The suggested file name takes into account the file naming pattern that is specified in the `location` argument.

**Data Types:** `function_handle`

**Text or Spreadsheet Files**

**WriteVariableNames** — Indicator for writing variable names as column headings
type: `true` or `1` (default) | `false` or `0`

Indicator for writing variable names as column headings, specified as the comma-separated pair consisting of `'WriteVariableNames'` and a numeric or logical `1` (true) or `0` (false).

<table>
<thead>
<tr>
<th>Indicator</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>Variable names are included as the column headings of the output. This is the default behavior.</td>
</tr>
<tr>
<td>false</td>
<td>Variable names are not included in the output.</td>
</tr>
</tbody>
</table>

**DateLocale** — Locale for writing dates
type: `character vector` | `string scalar`

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values

---

**11-414**
to the file, use `DateLocale` to specify the locale in which `write` should write month and
day-of-week names and abbreviations. The character vector or string takes the form
`xx_YY`, where `xx` is a lowercase ISO 639-1 two-letter code indicating a language, and `YY`
is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values
for the locale, see the `Locale` name-value pair argument for the `datetime` function.

For Excel® files, `write` writes variables containing `datetime` arrays as Excel dates and
ignores the `'DateLocale'` parameter value. If the `datetime` variables contain years
prior to either 1900 or 1904, then `write` writes the variables as text. For more
information on Excel dates, see Differences between the 1900 and the 1904 date system
in Excel.

Example: `'DateLocale','ja_JP'` or `'DateLocale','ja_JP`

Data Types: `char` | `string`

**Text Files Only**

**Delimiter — Field delimiter character**

',', or 'comma' | ' ' or 'space' | ...

Field delimiter character, specified as the comma-separated pair consisting of
'Delimiter' and one of these specifiers:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Field Delimiter</th>
</tr>
</thead>
<tbody>
<tr>
<td>','</td>
<td>Comma. This is the default behavior.</td>
</tr>
<tr>
<td>'comma'</td>
<td></td>
</tr>
<tr>
<td>' '</td>
<td>Space</td>
</tr>
<tr>
<td>'space'</td>
<td></td>
</tr>
<tr>
<td>'\t'</td>
<td>Tab</td>
</tr>
<tr>
<td>'tab'</td>
<td></td>
</tr>
<tr>
<td>';'</td>
<td>Semicolon</td>
</tr>
<tr>
<td>'semi'</td>
<td></td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>'bar'</td>
<td></td>
</tr>
</tbody>
</table>
You can use the 'Delimiter' name-value pair argument only for delimited text files.
Example: 'Delimiter','space' or 'Delimiter','"space"

**QuoteStrings — Indicator for writing quoted text**

false (default) | true

Indicator for writing quoted text, specified as the comma-separated pair consisting of 'QuoteStrings' and either false or true. If 'QuoteStrings' is true, then write encloses the text in double quotation marks, and replaces any double-quote characters that appear as part of that text with two double-quote characters. For an example, see “Write Quoted Text to CSV File” (MATLAB).

You can use the 'QuoteStrings' name-value pair argument only with delimited text files.

**Encoding — Character encoding scheme**

'UTF-8' | 'ISO-8859-1' | 'windows-1251' | 'windows-1252' | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of 'Encoding' and 'system' or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as 'system', the write function uses your system default encoding to write the file.

<table>
<thead>
<tr>
<th>'Big5'</th>
<th>'ISO-8859-1'</th>
<th>'windows-847'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Big5-HKSCS'</td>
<td>'ISO-8859-2'</td>
<td>'windows-949'</td>
</tr>
<tr>
<td>'CP949'</td>
<td>'ISO-8859-3'</td>
<td>'windows-1250'</td>
</tr>
<tr>
<td>'EUC-KR'</td>
<td>'ISO-8859-4'</td>
<td>'windows-1251'</td>
</tr>
<tr>
<td>'EUC-JP'</td>
<td>'ISO-8859-5'</td>
<td>'windows-1252'</td>
</tr>
<tr>
<td>'EUC-TW'</td>
<td>'ISO-8859-6'</td>
<td>'windows-1253'</td>
</tr>
<tr>
<td>'GB18030'</td>
<td>'ISO-8859-7'</td>
<td>'windows-1254'</td>
</tr>
<tr>
<td>'GB2312'</td>
<td>'ISO-8859-8'</td>
<td>'windows-1255'</td>
</tr>
<tr>
<td>'GBK'</td>
<td>'ISO-8859-9'</td>
<td>'windows-1256'</td>
</tr>
<tr>
<td>'IBM866'</td>
<td>'ISO-8859-11'</td>
<td>'windows-1257'</td>
</tr>
<tr>
<td>'K0I8-R'</td>
<td>'ISO-8859-13'</td>
<td>'windows-1258'</td>
</tr>
<tr>
<td>'K0I8-U'</td>
<td>'ISO-8859-15'</td>
<td>'US-ASCII'</td>
</tr>
<tr>
<td>Encoding</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------</td>
<td></td>
</tr>
<tr>
<td>'Macintosh'</td>
<td>'UTF-8'</td>
<td></td>
</tr>
<tr>
<td>'Shift_JIS'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: 'Encoding','system' or 'Encoding','system" uses the system default encoding.

**Spreadsheet Files Only**

**Sheet — Target worksheet**

Character vector | string scalar | positive integer

Target worksheet, specified as the comma-separated pair consisting of 'Sheet' and a character vector or a string scalar containing the worksheet name or a positive integer indicating the worksheet index. The worksheet name cannot contain a colon (:). To determine the names of sheets in a spreadsheet file, use `[status, sheets] = xlsfinfo(filename)`.

If the sheet does not exist, then `write` adds a new sheet at the end of the worksheet collection. If the sheet is an index larger than the number of worksheets, then `write` appends empty sheets until the number of worksheets in the workbook equals the sheet index. In either case, `write` generates a warning indicating that it has added a new worksheet.

You can use the 'Sheet' name-value pair argument only with spreadsheet files.

Example: 'Sheet',2
Example: 'Sheet', 'MySheetName'

Data Types: char | string | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Limitations**

In some cases, `write(location, D, 'FileType', type)` creates files that do not represent the original array `D` exactly. If you use `datastore(location)` to read the checkpoint files, then the result might not have the same format or contents as the original distributed table.

For the 'text' and 'spreadsheet' file types, `write` uses these rules:
- write outputs numeric variables using `longG` format, and categorical, character, or string variables as unquoted text.
- For non-text variables that have more than one column, write outputs multiple delimiter-separated fields on each line, and constructs suitable column headings for the first line of the file.
- write outputs variables with more than two dimensions as two-dimensional variables, with trailing dimensions collapsed.
- For cell-valued variables, write outputs the contents of each cell as a single row, in multiple delimiter-separated fields, when the contents are numeric, logical, character, or categorical, and outputs a single empty field otherwise.

Do not use the 'text' or 'spreadsheet' file types if you need to write an exact checkpoint of the distributed array.

**Tips**

- Use the write function to create checkpoints or snapshots of your data as you work. This practice allows you to reconstruct distributed arrays directly from files on disk rather than re-executing all of the commands that produced the distributed array.

**See Also**

datastore | distributed | tall

**Topics**

“Distributed Arrays”

**Introduced in R2017a**
zeros

Array of zeros

Syntax

\[ Z = \text{zeros}(sz, \text{arraytype}) \]
\[ Z = \text{zeros}(sz, \text{datatype}, \text{arraytype}) \]
\[ Z = \text{zeros}(sz, 'like', P) \]
\[ Z = \text{zeros}(sz, \text{datatype}, 'like', P) \]

\[ C = \text{zeros}(sz, \text{codist}) \]
\[ C = \text{zeros}(sz, \text{datatype}, \text{codist}) \]
\[ C = \text{zeros}(sz, \text{codist}, 'noCommunication') \]
\[ C = \text{zeros}(sz, \text{codist}, 'like', P) \]

Description

\[ Z = \text{zeros}(sz, \text{arraytype}) \] creates a matrix with underlying class of double, with zeros in all elements.

\[ Z = \text{zeros}(sz, \text{datatype}, \text{arraytype}) \] creates a matrix with underlying class of \text{datatype}, with zeros in all elements.

The size and type of array are specified by the argument options according to the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>( n )</td>
<td>Specifies size as an ( n )-by-( n ) matrix.</td>
</tr>
<tr>
<td></td>
<td>( m,n ) or ([m \ n])</td>
<td>Specifies size as an ( m )-by-( n ) matrix.</td>
</tr>
<tr>
<td></td>
<td>( m,n,\ldots,k ) or ([m \ n \ \ldots \ k])</td>
<td>Specifies size as an ( m )-by-( n )-by-( k ) array.</td>
</tr>
<tr>
<td>arraytype</td>
<td>'distributed'</td>
<td>Specifies distributed array.</td>
</tr>
<tr>
<td>Argument</td>
<td>Values</td>
<td>Descriptions</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>'codistributed'</td>
<td>Specifies codistributed array, using the default distribution scheme.</td>
</tr>
<tr>
<td></td>
<td>'gpuArray'</td>
<td>Specifies gpuArray.</td>
</tr>
<tr>
<td>datatype</td>
<td>'double' (default), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'</td>
<td>Specifies underlying class of the array, i.e., the data type of its elements.</td>
</tr>
</tbody>
</table>

\[
Z = \text{zeros}(sz,'\text{like}',P)\]
creates an array of zeros with the same type and underlying class (data type) as array \( P \).

\[
Z = \text{zeros}(sz,\text{datatype},'\text{like}',P)\]
creates an array of zeros with the specified underlying class (\text{datatype}), and the same type as array \( P \).

\[
C = \text{zeros}(sz,\text{codist}) \text{ or } C = \text{zeros}(sz,\text{datatype},\text{codist})\]
creates a codistributed array of zeros with the specified size and underlying class (the default \text{datatype} is 'double'). The codistributor object \text{codist} specifies the distribution scheme for creating the codistributed array. For information on constructing codistributor objects, see the reference pages for \text{codistributor1d} and \text{codistributor2dbc}. To use the default distribution scheme, you can specify a codistributor constructor without arguments. For example:

```matlab
spmd
    C = zeros(8,\text{codistributor1d}());
end
```

\[
C = \text{zeros}(sz,___,\text{codist},'\text{noCommunication}')\]
specifies that no interworker communication is to be performed when constructing a codistributed array, skipping some error checking steps.

\[
C = \text{zeros}(sz,___,\text{codist},'\text{like}',P)\]
creates a codistributed array of zeros with the specified size, underlying class, and distribution scheme. If either the class or codistributor argument is omitted, the characteristic is acquired from the codistributed array \( P \).
Examples

Create Distributed Zeros Matrix

Create a 1000-by-1000 distributed array of zeros with underlying class double:

\[ D = \text{zeros}(1000, 'distributed'); \]

Create Codistributed Zeros Matrix

Create a 1000-by-1000 codistributed double matrix of zeros, distributed by its second dimension (columns).

\[
\text{spmd}(4) \\
C = \text{zeros}(1000, 'codistributed'); \\
\text{end}
\]

With four workers, each worker contains a 1000-by-250 local piece of \( C \).

Create a 1000-by-1000 codistributed uint16 matrix of zeros, distributed by its columns.

\[
\text{spmd}(4) \\
\text{codist} = \text{codistributor}('1d', 2, 100*[1:numlabs]); \\
C = \text{zeros}(1000, 1000, 'uint16', \text{codist}); \\
\text{end}
\]

Each worker contains a 100-by-\text{labindex} local piece of \( C \).

Create gpuArray Zeros Matrix

Create a 1000-by-1000 gpuArray of zeros with underlying class uint32:

\[ G = \text{zeros}(1000, 'uint32', 'gpuArray'); \]

See Also

Inf | NaN | eye | false | ones | true | zeros

Introduced in R2006b
<table>
<thead>
<tr>
<th><strong>asynchronous processing</strong></th>
<th>execute a task without waiting for it to complete, in a non-blocking way, so that you can carry on with other tasks, using <code>parfeval</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CHECKPOINTBASE</strong></td>
<td>The name of the parameter in the <code>mdce_def</code> file that defines the location of the checkpoint directories for the MATLAB job scheduler and workers.</td>
</tr>
<tr>
<td><strong>checkpoint directory</strong></td>
<td>See CHECKPOINTBASE.</td>
</tr>
<tr>
<td><strong>client</strong></td>
<td>The MATLAB session that defines and submits the job. This is the MATLAB session in which the programmer usually develops and prototypes applications. Also known as the MATLAB client.</td>
</tr>
<tr>
<td><strong>client computer</strong></td>
<td>The computer running the MATLAB client; often your desktop.</td>
</tr>
<tr>
<td><strong>cluster</strong></td>
<td>A collection of computers that are connected via a network and intended for a common purpose.</td>
</tr>
<tr>
<td><strong>coarse-grained application</strong></td>
<td>An application for which run time is significantly greater than the communication time needed to start and stop the program. Coarse-grained distributed applications are also called embarrassingly parallel applications.</td>
</tr>
<tr>
<td><strong>codistributed array</strong></td>
<td>An array partitioned into segments, with each segment residing in the workspace of a different worker. When created, viewed, accessed, or manipulated from one of the worker sessions that contains part of the array, it is referred to as a codistributed array. Compare to distributed array.</td>
</tr>
<tr>
<td><strong>communicating job</strong></td>
<td>Job composed of tasks that communicate with each other during evaluation. All tasks must run simultaneously. A special case of communicating job is a parallel pool, used for executing <code>parfor</code>-loops and <code>spmd</code> blocks.</td>
</tr>
<tr>
<td><strong>Composite</strong></td>
<td>An object in a MATLAB client session that provides access to data values stored on the workers in a parallel pool,</td>
</tr>
</tbody>
</table>
such as the values of variables that are assigned inside an `spmd` statement.

<table>
<thead>
<tr>
<th><strong>computer</strong></th>
<th>A system with one or more processors.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Central Processing Unit, comprising multiple cores / processors</td>
</tr>
<tr>
<td><strong>distributed application</strong></td>
<td>The same application that runs independently on several nodes, possibly with different input parameters. There is no communication, shared data, or synchronization points between the nodes, so they are generally considered to be coarse-grained.</td>
</tr>
<tr>
<td><strong>distributed array</strong></td>
<td>An array partitioned into segments, with each segment residing in the workspace of a different worker. When created, viewed, accessed, or manipulated from the client session, it is referred to as a distributed array. Compare to codistributed array.</td>
</tr>
<tr>
<td><strong>DNS</strong></td>
<td>Domain Name System. A system that translates Internet domain names into IP addresses.</td>
</tr>
<tr>
<td><strong>dynamic licensing</strong></td>
<td>The ability of a MATLAB worker to employ all the functionality you are licensed for in the MATLAB client, while checking out only an engine license. When a job is created in the MATLAB client with Parallel Computing Toolbox software, the products for which the client is licensed will be available for all workers that evaluate tasks for that job. This allows you to run any code on the cluster that you are licensed for on your MATLAB client, without requiring extra licenses for the worker beyond MATLAB Distributed Computing Server software. For a list of products that are not eligible for use with Parallel Computing Toolbox software, see <a href="https://www.mathworks.com/products/ineligible_programs/">https://www.mathworks.com/products/ineligible_programs/</a>.</td>
</tr>
<tr>
<td><strong>fine-grained application</strong></td>
<td>An application for which run time is significantly less than the communication time needed to start and stop the program. Compare to coarse-grained applications.</td>
</tr>
<tr>
<td><strong>Term</strong></td>
<td><strong>Definition</strong></td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit, now widely used for general purpose (GP) GPU computing.</td>
</tr>
<tr>
<td>head node</td>
<td>Usually, the node of the cluster designated for running the job scheduler and license manager. It is often useful to run all the nonworker related processes on a single machine.</td>
</tr>
<tr>
<td>heterogeneous cluster</td>
<td>A cluster that is not homogeneous.</td>
</tr>
<tr>
<td>homogeneous cluster</td>
<td>A cluster of identical machines, in terms of both hardware and software.</td>
</tr>
<tr>
<td>independent job</td>
<td>A job composed of independent tasks, which do not communicate with each other during evaluation. Tasks do not need to run simultaneously.</td>
</tr>
<tr>
<td>job</td>
<td>The complete large-scale operation to perform in MATLAB, composed of a set of tasks.</td>
</tr>
<tr>
<td>job scheduler</td>
<td>The MathWorks process that queues jobs and assigns tasks to workers. Formerly known as a job manager.</td>
</tr>
<tr>
<td>checkpoint information</td>
<td>Snapshot of information necessary for the MATLAB job scheduler to recover from a system crash or reboot.</td>
</tr>
<tr>
<td>job scheduler database</td>
<td>The database that the MATLAB job scheduler uses to store the information about its jobs and tasks.</td>
</tr>
<tr>
<td>LOGDIR</td>
<td>The name of the parameter in the mdce_def file that defines the directory where logs are stored.</td>
</tr>
<tr>
<td>MATLAB client</td>
<td>See client.</td>
</tr>
<tr>
<td>MATLAB job scheduler</td>
<td>The MATLAB process that queues jobs and assigns tasks to workers. Formerly known as a job manager.</td>
</tr>
<tr>
<td>(MJS)</td>
<td></td>
</tr>
<tr>
<td>MATLAB worker</td>
<td>See worker.</td>
</tr>
<tr>
<td>mdce</td>
<td>The service that has to run on all machines before they can run a MATLAB job scheduler or worker. This is the engine foundation process, making sure that the job scheduler and worker processes that it controls are always running.</td>
</tr>
</tbody>
</table>

Note that the program and service name is all lowercase letters.
<p>| <strong>mdce_def file</strong> | The file that defines all the defaults for the mdce processes by allowing you to set preferences or definitions in the form of parameter values. |
| <strong>MPI</strong> | Message Passing Interface, the means by which workers communicate with each other while running tasks in the same job. |
| <strong>node</strong> | A computer that is part of a cluster. |
| <strong>parallel application</strong> | The same application that runs on several workers simultaneously, with communication, shared data, or synchronization points between the workers. |
| <strong>parallel pool</strong> | A collection of workers that is reserved by the client, created using <code>parpool</code> and runs a special communicating job for execution of <code>parfor</code>-loops, <code>spmd</code> statements, and distributed arrays. |
| <strong>private array</strong> | An array which resides in the workspaces of one or more, but perhaps not all workers. There might or might not be a relationship between the values of these arrays among the workers. |
| <strong>random port</strong> | A random unprivileged TCP port, i.e., a random TCP port above 1024. |
| <strong>register a worker</strong> | The action that happens when both worker and MATLAB job scheduler are started and the worker contacts the job scheduler. |
| <strong>replicated array</strong> | An array which resides in the workspaces of all workers, and whose size and content are identical on all workers. |
| <strong>scheduler</strong> | The process, either local, third-party, or the MATLAB job scheduler, that queues jobs and assigns tasks to workers. |
| <strong>spmd (single program multiple data)</strong> | A block of code that executes simultaneously on multiple workers in a parallel pool. Each worker can operate on a different data set or different portion of distributed data, and can communicate with other participating workers while performing the parallel computations. |</p>
<table>
<thead>
<tr>
<th><strong>task</strong></th>
<th>One segment of a job to be evaluated by a worker.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>thread</strong></td>
<td>smallest set of instructions that can be managed independently by a scheduler. On a GPU, multiprocessor or multicore system, multiple threads can be executed simultaneously (multi-threading).</td>
</tr>
<tr>
<td><strong>variant array</strong></td>
<td>An array which resides in the workspaces of all workers, but whose content differs on these workers.</td>
</tr>
<tr>
<td><strong>worker</strong></td>
<td>The MATLAB session that performs the task computations. Also known as the MATLAB worker or worker process.</td>
</tr>
<tr>
<td><strong>worker checkpoint information</strong></td>
<td>Files required by the worker during the execution of tasks.</td>
</tr>
</tbody>
</table>