CHAPTER

Parallel Computing Toolbox 3

CHAPTER OBJECTIVES

This Chapter introduces the key features of MATLAB's Parallel Computing Toolbox. Task- and data-parallel applications can be parallelized using the features provided by Parallel Computing Toolbox. After reading this Chapter, you should be able to:

- understand the key features of Parallel Computing Toolbox.
- execute loop iterations in parallel using a *parfor-loop*.
- execute code in parallel on MATLAB workers using an *spmd* statement.
- use *distributed* and *codistributed* arrays across many MATLAB workers.
- work interactively with a communicating job using the *pmode* functionality.
- create and submit jobs on a computer cluster.

3.1 PRODUCT DESCRIPTION AND OBJECTIVES

MATLAB's Parallel Computing Toolbox allows users to solve computationally and data-intensive problems using multicore and multiprocessor computers, computer clusters, and GPUs. Users can use high-level MATLAB functions to parallelize applications without OpenMP, MPI, and CUDA programming. One of the important features of MATLAB's Parallel Computing Toolbox is that the same application can be executed on a simple core, a multicore processor, or a computer cluster without changing the code. Combining Parallel Computing Toolbox with Distributed Computing Server, users can run their MATLAB programs on computer clusters, grids, and clouds.

Other MATLAB products can take advantage of parallel computing resources and speedup users' codes. As of MATLAB R2015b, the toolboxes that have parallel capabilities are presented in Table 3.1 [12].

The toolboxes that provide GPU support will be presented in Chapter 5. Readers who are interested in utilizing Parallel Computing Toolbox in other toolboxes presented in Table 3.1 are directed to toolboxes user's guides provided by MathWorks. The rest of this chapter will present the most important programming features of the Parallel Computing Toolbox.

Table 3.1 Parallel computing support in MATLAB and Simulink products

Product name	**Parallel capabilities**
Simulink	Run multiple Simulink simulations using simcommand with parfor. Run multiple simulations in rapid accelerator mode using parfor with prebuilt Simulink models.
Embedded Coder	Generate and build code in parallel using model blocks.
Simulink Coder	Generate and build code in parallel using model blocks.
Bioinformatics Toolbox	Distribute pairwise alignments to a computer cluster using functions for progressive alignment of multiple sequences and pairwise distance between sequences.
Robust Control Toolbox	Tune fixed-structure control systems with the looptune, systune, and hinfstruct commands.
Simulink Control Design	Perform frequency response estimation of Simulink models.
Simulink Design Optimization	Estimate model parameters and optimize system response.
Image Processing Toolbox	Improve performance in Batch Image Processor. Improve performance of block processing tasks in blockproc function. **Improve performance of many functions using a GPU.**
Computer Vision System Toolbox	Improve performance of functions in bag-of-words workflow.
Global Optimization Toolbox	Explore simultaneously local solution space in genetic algorithm, particle swarm, and pattern search solvers.
Model-Based Calibration Toolbox	Fit multiple models to experimental data. Run of multiple optimizations in parallel.
Neural Network Toolbox	Improve performance of training and simulation. **Improve performance of training and simulation using a GPU.**
Optimization Toolbox	Accelerate gradient estimation in selected constrained nonlinear solvers. Launch parallel computations from optimtool GUI.
Statistics and Machine Learning Toolbox	Improve performance of many resampling functions. Fit multiclass support vector machines and other classifiers. Generate reproducible random streams in parallel. Improve performance of many D-optimal design generation functions. Improve performance of many functions with multiple starting points. **Improve performance of many functions using a GPU.**
Communications System Toolbox	**Improve performance of many System objects using a GPU.** **MATLAB Compiler support for GPU System objects.**
Phased Array System Toolbox	Accelerate clutter model simulation using parfor. **Accelerate clutter model simulation using a GPU.**
Signal Processing Toolbox	**Improve performance of many functions using a GPU.**
SystemTest	Run test iterations on multiple processors or machines.

The Parallel Computing Toolbox can be used to improve the performance of various types of applications:

- Applications that involve repetitive segments of code. Different iterations can be evaluated in parallel using parallel *for-loops* (Section 3.2). Applications can include many iterations that might not take long to execute or few iterations that take long to execute. The only restriction on parallel *for-loops* is that each iteration is not allowed to depend on any other iteration.
- Applications that run a series of independent tasks. Different tasks can be evaluated in parallel using parallel *for-loops* (Section 3.2).
- Applications evaluating the same code on multiple data sets. Multiple workers can be used to run the same code on multiple data sets (Section 3.3).
- Applications involving data that is too large for your computer's memory. Arrays can be distributed among multiple MATLAB workers, and each MATLAB worker can operate only on its part of the array (Sections 3.3 and 3.4).
- Applications that their execution in parallel can improve their performance (Sections 3.2 and 3.3).
- Applications that their execution on GPUs can improve their performance (Section 3.6).

3.2 PARALLEL FOR-LOOPS (*parfor*)

It is well-known that a *for-loops* is a series of statements that are executed over a range of values. Similarly, a *parfor-loop* in MATLAB is the same as the standard *for-loops* with the difference that the loop iterations can be executed in parallel. To understand better the concept of the *parfor-loop*, let's introduce some terminology. The MATLAB client is the MATLAB instance that initiated the execution of the *parfor-loop*, while the MATLAB worker is a MATLAB process that runs in parallel without a graphical user interface. Several MATLAB workers can be used to evaluate a *parfor-loop*. As of MATLAB R2015b, a maximum number of 512 MATLAB workers can be used in parallel. To set the number of the MATLAB workers on your machine, select the Parallel Preferences option from MATLAB's workspace (Fig. 3.1). Then, select the Cluster Profile Manager option (Fig. 3.2). Finally, select a cluster profile (the default is named 'local'), click on Edit, and enter a value for the number of workers that you want to use (Fig. 3.3). The default option is the number of cores. If multithreading is enabled, you can set the number of workers equal to the number of threads.

Before running a code that includes a *parfor-loop* or an *spmd* statement, you should open a parallel pool. To open a parallel pool using a specific cluster, e.g., 'local', and a specific size, e.g., 4, type the following command:

```
>> parpool('local', 4)
```

After running your code, you can shut down the parallel pool by typing the following commands (*gcp* returns the current parallel pool):

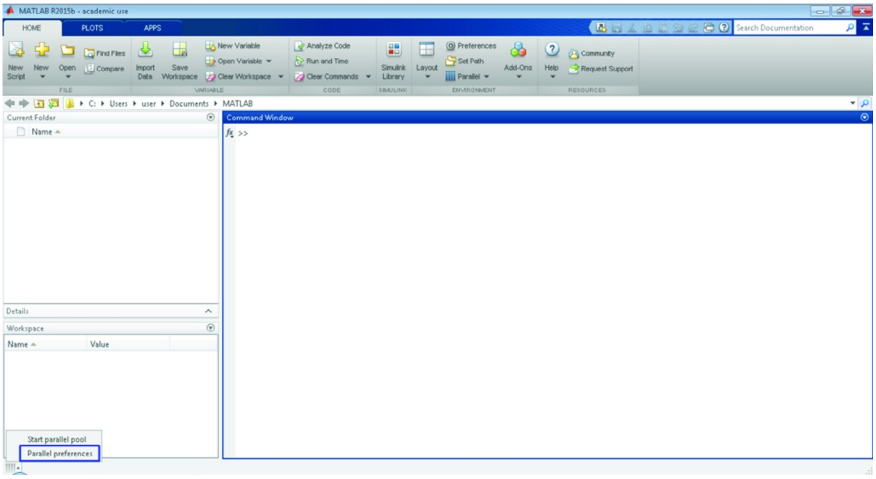
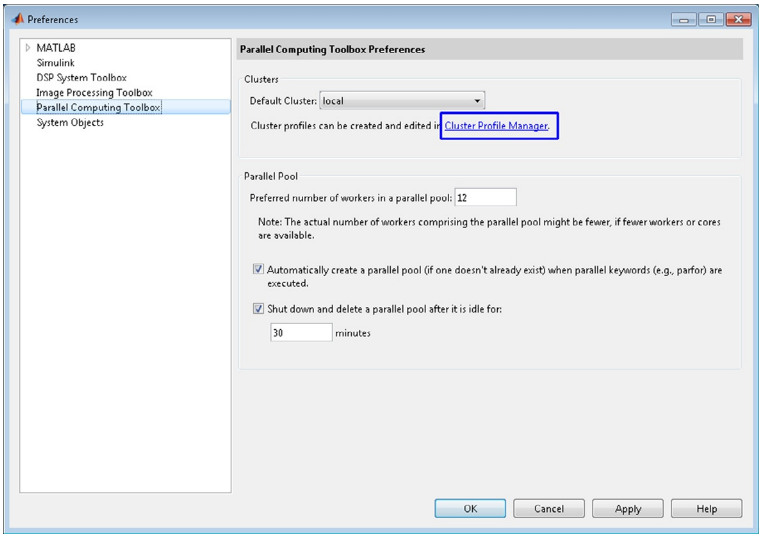


FIG. 3.1

Select the parallel preferences option.



Preferences

- MATLAB
- Simulink
- DSP System Toolbox
- Image Processing Toolbox
- Parallel Computing Toolbox
- System Objects

Parallel Computing Toolbox Preferences

Clusters

Default Cluster: local

Cluster profiles can be created and edited in Cluster Profile Manager.

Parallel Pool

Preferred number of workers in a parallel pool: 12

Note: The actual number of workers comprising the parallel pool might be fewer, if fewer workers or cores are available.

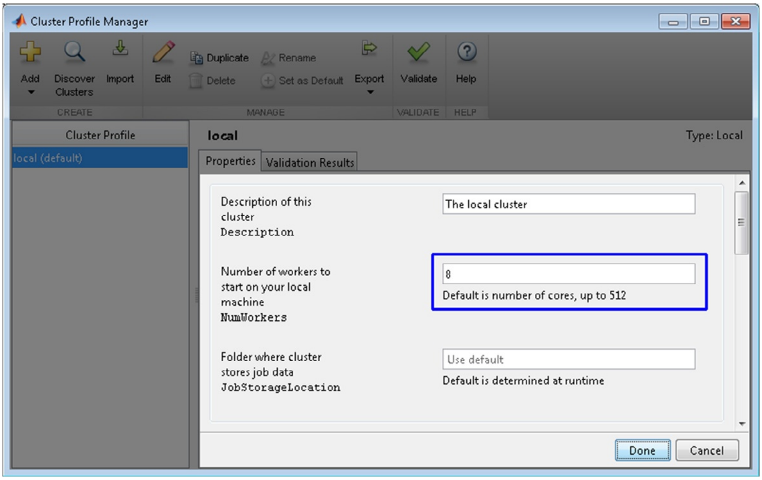
[✓] Automatically create a parallel pool (if one doesn't already exist) when parallel keywords (e.g., parfor) are executed.

[✓] Shut down and delete a parallel pool after it is idle for:

30 minutes

FIG. 3.2

Select the cluster profile manager option.



Description of this cluster
Description
The local cluster

Number of workers to start on your local machine
NumWorkers
8
Default is number of cores, up to 512

Folder where cluster stores job data
JobStorageLocation
Use default
Default is determined at runtime

FIG. 3.3

Set the number of workers.

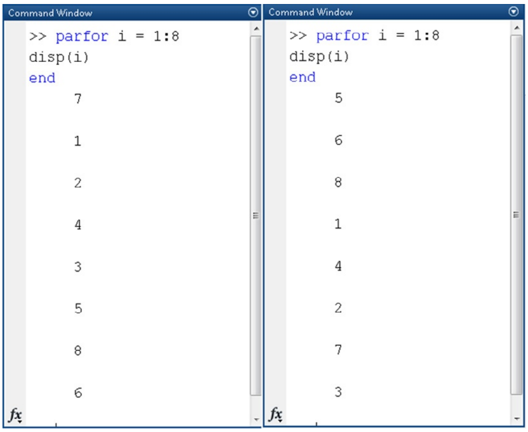
```
>> p = gcp;
>> delete(p)
```

A *parfor-loop* is initiated on the MATLAB client, and the MATLAB client sends the necessary data to the MATLAB workers. Each MATLAB worker performs the computations that were assigned to it and sends the results back to the MATLAB client. Finally, the MATLAB client pieces the results together. There is no guarantee that all MATLAB workers will be assigned the same number of iterations. If the number of MATLAB workers is equal to the number of the iterations, then each MATLAB worker performs one iteration. If there are more iterations, then some MATLAB workers may perform more iterations than others. Moreover, the iterations are evaluated in no particular order and independent of each other. For example, let's consider the following example:

```
parfor i = 1:8
    disp(i)
end
```

Using a parallel pool with 4 MATLAB workers, let's execute the preceding code twice (Fig. 3.4). The results show that the iterations are evaluated in no particular order.

Moreover, each iteration must be independent of all other iterations. For instance, the MATLAB worker evaluating the third iteration might not be the same MATLAB worker evaluating the fifth iteration. Thus, each iteration cannot use data used in other iterations. Let's take, for example, the following code in which the first *n* Fibonacci numbers are calculated.


```
>> parfor i = 1:8
disp(i)
end
     7

     1

     2

     4

     3

     5

     8

     6
```

```
>> parfor i = 1:8
disp(i)
end
     5

     6

     8

     1

     4

     2

     7

     3
```

FIG. 3.4

Iterations evaluation in no particular order.

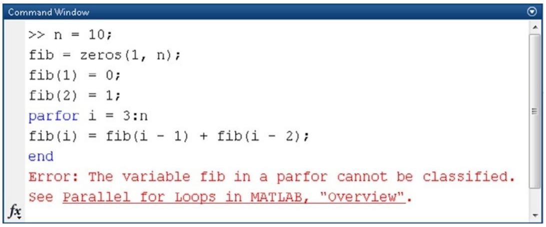
```
n = 10;
fib = zeros(1, n);
fib(1) = 0;
fib(2) = 1;
for i = 3:n
   fib(i) = fib(i - 1) + fib(i - 2);
end
```

As you may already know, the sequence F_n of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$. The statement in each iteration depends on results that are calculated in previous iterations. For instance, to compute the fifth Fibonacci number ($i = 5$), the fourth Fibonacci number ($i = 4$) and the third Fibonacci number ($i = 3$) are needed. However, as mentioned earlier, each iteration must be independent of all other iterations, and the iterations are evaluated in no particular order. Thus, we cannot use *parfor* in this example. Even if we used the *parfor* to parallelize the *for-loop*, MATLAB outputs an error that cannot classify the variable *fib* in the *parfor-loop* (Fig. 3.5). Note that the calculation of the first n Fibonacci numbers can be calculated in parallel if we use Binet's formula, $f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$, as already mentioned in Chapter 1.

MATLAB classifies the variables found in a *parfor-loop* in one of the following categories:

(1) **Loop Variables**: Loop variables define the loop index value for each iteration. A loop variable is set at the beginning line of a *parfor-loop*:

```
parfor i = 1:100
      ...
end
```



```
>> n = 10;
fib = zeros(1, n);
fib(1) = 0;
fib(2) = 1;
parfor i = 3:n
fib(i) = fib(i - 1) + fib(i - 2);
end
Error: The variable fib in a parfor cannot be classified.
See Parallel for Loops in MATLAB, "Overview".
```

FIG. 3.5

Wrong use of parfor in non-independent iterations.

There are some restrictions while using a loop variable:

- A loop variable must evaluate to ascending consecutive integers. Hence, the *parfor* statements *parfor i = 1:100* and *parfor i = -10:10* are valid, while the *parfor* statements *parfor i = -100:-1:1*, *parfor i = 1:2:100*, and *parfor i = 1:0.5:10* are not valid.
- Assignments to the loop variable are not allowed. Hence, the following *parfor-loop* is not valid:

```
parfor i = 1:100
    i = i + 2; % not valid
    ...
end
```

- A loop variable cannot be indexed or subscripted. Hence, the following *parfor-loop* is not valid:

```
parfor i = 1:100
    x = i(1); % not valid
    y = i.a; % not valid
end
```

(2) **Sliced Variables**: Sliced variables are arrays whose segments are manipulated by different workers on different loop iterations. Sliced variables are used to reduce the communication time between the MATLAB client and MATLAB workers because only those slices needed by a worker are sent to it when it starts working on a particular range of indices. Let *i* be a loop variable. A variable is sliced if it has all the following characteristics [5]:

- The first level of indexing is enclosed in either parentheses ‘()’ or braces ‘{}’. After the first level, any valid MATLAB indexing can be used. Hence, the variable *A.p{i}* is not sliced, while the variables *A{i}*, *A(i)*, *A(i, 5)*, and *A(i).p* are sliced.
- Within the first-level parenthesis or braces, the list of indices is the same for all occurrences of a given variable. In the following code, variables *A* and *B* are not sliced because *A* is indexed by *i* and $i+1$ in different places and *B* is indexed by $(i, 1)$ and $(i, 2)$ in different places.

```
>> A = rand(1, 101);
>> B = rand(100, 2);
>> parfor i = 1:100
       x = max(A(i), A(i + 1)); % not sliced
       y = B(i, 1) + B(i, 2); % not sliced
   end
```

- Within the list of indices for the variable, exactly one index involves the loop variable, and every other index is a scalar constant, a broadcast variable, a nested *for-loop* index, *colon*, or *end*. Hence, *A(i, i + 1)* and *B(i, 20:30, end)* are not sliced, while *A(i, :, end)* and *B(i + 2, j, :, 3)* are sliced.
- The sliced variable maintains a constant shape (no deletion or insertion operators are allowed). In the following code, variables *A* and *B* are not sliced because they change their shape by adding or deleting elements.

```
>> A = rand(100, 2);
>> B = rand(100);
>> parfor i = 1:100
       A(i, :) = []; % not sliced
       B(end + 1) = i; % not sliced
   end
```

All sliced variables have the characteristics of being input or output. Some sliced variables have both characteristics. Sliced input variables are transmitted from the MATLAB client to the MATLAB workers, sliced output variables are transmitted from the MATLAB workers to the MATLAB client, and sliced input and output variables are transmitted in both directions. In the following code, variable *A* is a sliced input variable, variable *B* is a sliced output variable, and variable *C* is a sliced input and output variable.

```
>> A = rand(100);
>> C = rand(100);
>> parfor i = 1:100
       B(i) = A(i);
       if rand < 0.5
          C(i) = 0;
       end
   end
```

(3) **Broadcast Variables**: Broadcast variables are variables other than loop and sliced variables that are defined before a *parfor-loop* and are used inside the loop but never modified. The MATLAB client sends the broadcast variables to all MATLAB workers at the start of a *parfor-loop*. In the following code, variables a and *B* are broadcast variables.

```
>> a = 0.4;
>> B = rand(101, 1);
>> C = zeros(100, 1);
>> parfor i = 1:100
       if B(i) + B(i + 1) < a
          C(i) = 0;
```

```
      else
         C(i) = 1;
      end
   end
```

Note that variable *B* is not a sliced variable because it is indexed by *i* and $i + 1$ in different places. Large broadcast variables can cause a lot of communication between the MATLAB client and the MATLAB workers, and degrade performance. In some cases it is more efficient to use temporary instead of broadcast variables, if possible.

(4) **Reduction Variables**: Reduction variables accumulate a value that depends on all the iterations together. This is the only exception to the rule that loop iterations must be independent. However, the value of the reduction variable is independent of the iteration order. Reductions variables appear on both sides of an assignment statement. Table 3.2 presents all the possible assignments in which a reduction variable can appear (*expr* is a MATLAB expression).
In the following code, the usage of a reduction variable *X* is illustrated:

```
>> A = rand(100, 1);
>> X = 0;
>> parfor i = 1:100
      X = X + A(i);
   end
```

The value of variable *X* is not transmitted from the MATLAB client to MATLAB workers or from a MATLAB worker to another. Rather, each MATLAB worker performs a number of additions of *A(i)* and transmits back to the MATLAB client the result. The MATLAB client receives the results from each MATLAB worker and adds these into *X*. Some other rules for the reduction variables are the following:

Table 3.2 Possible assignments involving a reduction variable *X*

Assignment	Assignment
X= X + expr	X = expr + X
X = X - expr	
X = X .* expr	X = expr .* X
X = X * expr	X = expr * X
X = X & expr	X = expr & X
X = X | expr	X = expr | X
X = [X, expr]	X = [expr, X]
X = [X; expr]	X = [expr; X]
X = min(X, expr)	X = min(expr, X)
X= max(X, expr)	X = max(expr, X)
X = union(X, expr)	X = union(expr, X)
X = intersect(X, expr)	X = intersect(expr, X)

- For any reduction variable, the same reduction function or operation must be used in all reduction assignments for that variable. For example, the following *parfor-loop* is not valid because the reduction assignment uses '+' in one instance and '*' in another.

  ```
  >> X = 0;
  >> parfor i = 1:100
         if rand < 0.5
            X = X + 5;
         else
            X = X * 2;
         end
     end
  ```

  On the other hand, the following *parfor-loop* is valid.

  ```
  >> X = 0;
  >> parfor i = 1:100
         if rand < 0.5
            X = X + 5;
         else
            X = X + i + 2;
         end
     end
  ```

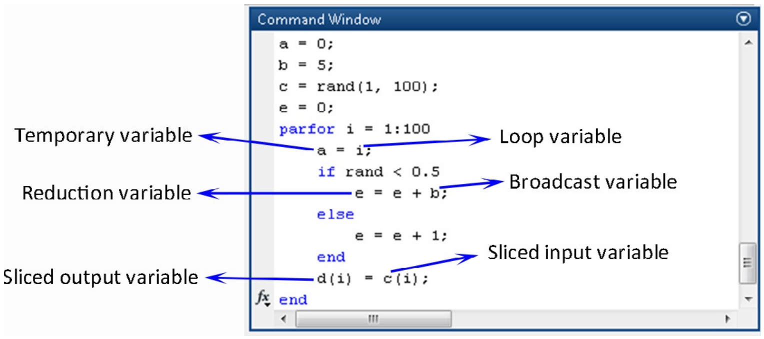
- If the reduction assignment uses '*' or '[,]' or '[;]', then in every reduction assignment for *X*, *X* must be consistently specified as the first argument or consistently specified as the second. For example, the following code is not valid because variable *X* does not occur in the same argument position in all reduction assignments.

  ```
  >> X = 0;
  >> parfor i = 1:100
         if rand < 0.5
            X = X * 5;
         else
            X = 2 * X;
         end
     end
  ```

  On the other hand, the following *parfor-loop* is valid.

  ```
  >> X = 0;
  >> parfor i = 1:100
         if rand < 0.5
            X = [X, 5];
         else
            X = [X, i + 2];
         end
     end
  ```

- The logical operators && and || are not allowed in reduction assignments.



```
a = 0;
b = 5;
c = rand(1, 100);
e = 0;
parfor i = 1:100
    a = i;
    if rand < 0.5
        e = e + b;
    else
        e = e + 1;
    end
    d(i) = c(i);
end
```

FIG. 3.6

Types of *parfor* variables.

(5) **Temporary Variables**: Temporary variables are variables that are the target of direct non-indexed assignments, but they are not reduction variables. Temporary variables exist only in a MATLAB worker's workspace and are not transferred back to the MATLAB client. In the following code, variables a and *b* are temporary variables.

```
>> a = 0;
>> parfor i = 1:100
      a = i;
      if rand < 0.5
         b = 0;
      else
         b = 1;
      end
   end
```

To sum up, Fig. 3.6 presents an example where all different types of *parfor* variables are used.

If a *parfor-loop* contains a variable that cannot be uniquely classified into one of the preceding categories or a variable violates its category restrictions, MATLAB generates an error, as shown in Fig. 3.5. Understanding the different types of *parfor* variables will help you find performance degrades.

You should also consider the following limitations/concerns before parallelizing your application using a *parfor-loop*:

- A code including a *parfor-loop* may result in better execution times in a multicore machine but not in a computer cluster due to the communication cost.
- You cannot use a *parfor-loop* if an iteration depends on the results of other iterations.
- There might be no benefit to using a *parfor-loop* if there is a small number of simple calculations due to the communication cost.

- When running a *parfor-loop* in a computer cluster, the communication cost will be greater than running the same code on local MATLAB workers.
- Be careful when using *cd*, *addpath*, and *rmpath* on a *parfor-loop* as the MATLAB search path might not be the same on all MATLAB workers.
- If an error occurs in one iteration, then all MATLAB workers terminate.
- Be careful when using commands like *input* and *keyboard*, which are not strictly computational in nature, as they might not have a visible effect or might hang the corresponding MATLAB worker.
- Any graphical output, for example, a figure, will not be displayed at all if it is used inside a *parfor-loop*.
- If you want to use objects inside a loop, then explicitly assign them to a variable, as they are not automatically propagated to the client.
- If you want to call a function handle with the loop variable as an argument, use *feval*.
- The body of a *parfor-loop* cannot make reference to a nested function, but it can call a nested function by means of a function handle.
- The body of a *parfor-loop* cannot contain another *parfor-loop*. Hence, if you have a code with two nested *for-loops*, you can parallelize either of the nested loops. Usually, the best strategy is to convert the outer *for-loop* to a *parfor-loop*.
- The range of a *for-loop* nested in a *parfor-loop* must be defined by constant numbers or variables and not by using function calls, like *length(A)*.
- When using a nested *for-loop* variable for indexing a sliced variable, you must use the variable in plain form and not as part of an expression (e.g., you can use *A(j)* but not *A(j + 1)*, where *j* is the nested *for-loop* variable and *A* a sliced variable).
- If you index into a sliced variable inside a nested *for-loop*, you cannot use that variable elsewhere in the *parfor-loop*.
- If you index into a sliced variable inside multiple *for-loops* (not nested) in a *parfor-loop*, they must loop over the same range of values.
- A sliced output variable can be used in only one nested *for-loop*.
- The body of a *parfor-loop* cannot contain *break* or *return* statements.
- You cannot clear variables from a MATLAB worker's workspace (e.g., clear x;), but instead you can set its value to empty (e.g., x = [];).
- You cannot use functions *eval*, *evalc*, *evalin* if the variables involved in these functions are not visible in the *parfor-loop* (e.g., they are declared prior to the *parfor-loop*).
- You cannot use *save* and *load* inside a *parfor-loop* unless the output of *load* is assigned to a variable.

There are mainly two types of applications that the use of a *parfor-loop* will improve their computation time:

- Applications that involve repetitive segments of code and include many iterations that do not take long to execute
- Applications that involve repetitive segments of code and include few iterations that take long to execute

Example: Monte Carlo simulation to approximate the area of a figure

A Monte Carlo simulation is a useful method to approximate the area of a figure. In some cases, it is difficult to find the exact area of a figure; so, this approximation method is very useful. A Monte Carlo method performs the following steps to find the area of a figure:

- Generates uniformly n random points on the square that encloses the figure and counts the number of points (*counter*) that lie inside the figure.
- Calculates the ratio $counter/n$, which approximates the ratio between the area of the figure and the square that encloses the figure.
- Computes the area of the given figure by multiplying the calculated ratio with the known area of the square that encloses the figure.

For instance, the previously mentioned Monte Carlo simulation will be applied to compute the area of the following definite integral:

$$\int_0^2 x(x-2)^6 dx$$

The area of this definite integral is the area under the curve $y = x(x-2)^6$ from $x = 0$ to $x = 2$ (Fig. 3.7).

A single-threaded MATLAB code to compute the area of the above definite integral is the following:

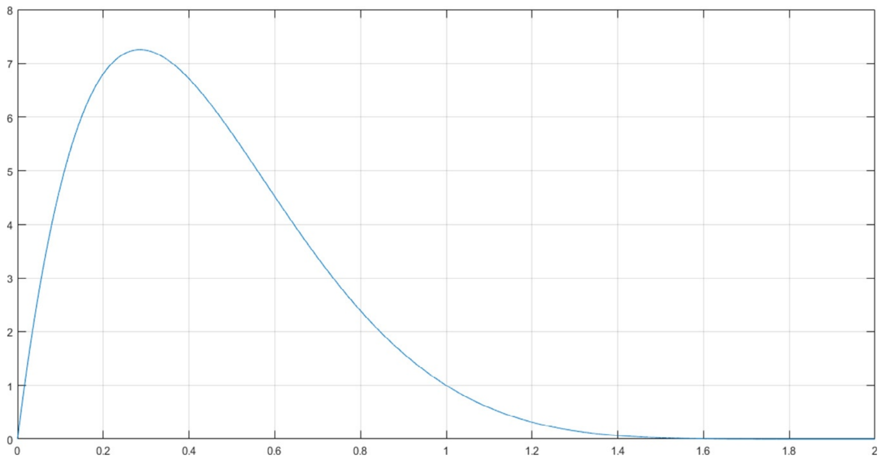


FIG. 3.7

Plot of the curve $y = x(x-2)^6$ from $x = 0$ to $x = 2$.

```
function area = test(n)
% Filename: test.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (single-threaded version)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = test(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

counter = 0;
for i = 1:n
   x = 2 * rand;
   y = 8 * rand;
   if(y <= x * (x - 2) ^ 6)
         counter = counter + 1;
   end
end
area = counter / n * 2 * 8;
end
```

The number of the uniformly *n* random points that will be generated is an important part of each Monte Carlo method. The higher the number, the more accurate will be the calculation of the area. So, let's use 1,000,000,000 random points to calculate the area. The *test* function runs in 188.89 seconds.

Someone could argue that the use of vectorized code will improve the execution time of the function *test*. The equivalent vectorized function is the following:

```
function area = testVectorized(n)
% Filename: testVectorized.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (vectorized version)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = testVectorized(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

x = 2 * rand(n, 1);
y = 8 * rand(n, 1);
counter = sum(y < (x .* (x - 2) .^ 6));
area = counter / n * 2 * 8;
end
```

The *testVectorized* function runs in 56.45 seconds, a ~3.35× speedup compared to the *test* function. This is a very good speedup and it was expected because the vectorized function is executed in multiple threads. However, the *testVectorized* function requires 16 Gb of RAM to create and store variables *x* and *y*. This is a huge amount of memory that is not available in many computers.

Assuming that our computer has a smaller amount of memory, we can modify the *test* function to take advantage of the vectorized code. Instead of generating vectors *x* and *y* with 1,000,000,000 values, we will generate them with 10,000,000 values and perform the *for-loop* 100 times. So, the following code only uses 160 MB for variables *x* and *y*.

```
function area = test2(n)
% Filename: test2.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (single-threaded semi-vectorized version)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = test2(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

n = n / 100;
counter = 0;
for i = 1:100
    x = 2 * rand(n, 1);
    y = 8 * rand(n, 1);
    counter = counter + sum(y < (x .* (x - 2) .^ 6));
end
area = counter / (n * 100) * 2 * 8;
end
```

The *test2* function runs in 56.83 seconds, a ∼3.32× speedup compared to the *test* function.

Initially, let's use a *parfor-loop* to parallelize the *test* function.

```
function area = testparfor(n)
% Filename: testparfor.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (multi-threaded version using parfor)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = testparfor(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

counter = 0;
parfor i = 1:n
   x = 2 * rand;
   y = 8 * rand;
   if(y <= x * (x - 2) ^ 6)
```

```
        counter = counter + 1;
    end
end
area = counter / n * 2 * 8;
end
```

The *testparfor* function runs in 36.12 seconds using 8 MATLAB workers, a ~5.23× speedup compared to the *test* function. We cannot achieve a linear speedup of 8× due to the communication cost.

Next, let's use a *parfor-loop* to parallelize the *test2* function.

```
function area = testparfor2(n)
% Filename: testparfor2.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (multi-threaded semi-vectorized version using parfor)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = testparfor2(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

n = n / 100;
counter = 0;
parfor i = 1:100
    x = 2 * rand(n, 1);
    y = 8 * rand(n, 1);
    counter = counter + sum(y < (x .* (x - 2) .^ 6));
end
area = counter / (n * 100) * 2 * 8;
end
```

The *testparfor2* function runs in 25.95 seconds using 8 MATLAB workers, a ~7.28× speedup compared to the *test* function and a ~2.19× speedup compared to the *test2* function.

3.3 SINGLE PROGRAM MULTIPLE DATA (*spmd*)

The single programming multiple data (*spmd*) statement allows seamless interleaving of serial and parallel programming. A block of code to run simultaneously on multiple MATLAB workers can be defined in an *spmd* statement. An *spmd* statement is used when we want to execute an identical code on multiple data. Each MATLAB worker will execute the same code on different data. Before and after an *spmd* statement, the code is executed on the MATLAB client, and the code inside an *spmd* statement is executed on the MATLAB workers. Hence, the general form of an *spmd* statement is the following:

```
... % statements executed on the MATLAB client
spmd
    ... % statements executed on multiple MATLAB workers
end
... % statements executed on the MATLAB client
```

You can also define the number of MATLAB workers to be used in an *spmd* statement:

```
spmd(n)
    ...
end
```

or

```
spmd(m, n)
    ...
end
```

In the first case (*spmd(n)*), the *spmd* statement requires that *n* MATLAB workers will run the *spmd* block of code. If the pool is large enough, but *n* MATLAB workers are not available, the statement waits until enough MATLAB workers are available. In the second case (*spmd(m, n)*), the *spmd* statement requires a minimum of *m* MATLAB workers, and it uses a maximum of *n* MATLAB workers, if available in the pool.

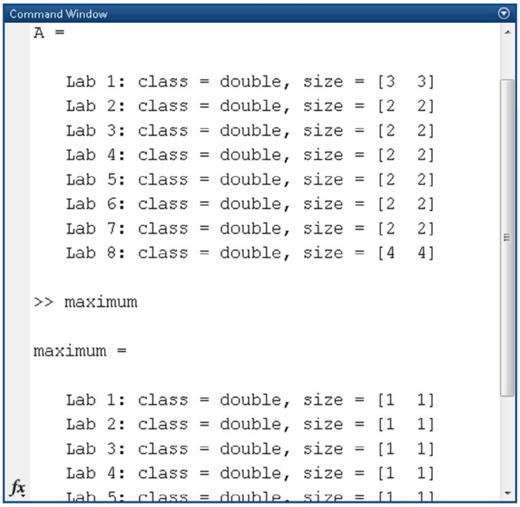
Each MATLAB worker used in an *spmd* statement has a unique value of *labindex* that can be used to run codes on specific MATLAB workers or access unique data. The total number of MATLAB workers used in an *spmd* statement can be obtained using the *numlabs* value. For example, we can create different sized arrays depending on *labindex*, as shown in the following code:

```
spmd
    if labindex == 1
        A = rand(3, 3);
    elseif labindex == numlabs
        A = rand(4, 4);
    else
        A = rand(2, 2);
    end
    maximum = max(max(A));
end
```

In the preceding example *A* and *maximum* are *composite* objects (Fig. 3.8). *Composite* objects contain references to the values stored on the MATLAB workers and can be retrieved on the MATLAB client using cell-array indexing (Fig. 3.9).

There are two ways to create *composite* objects:

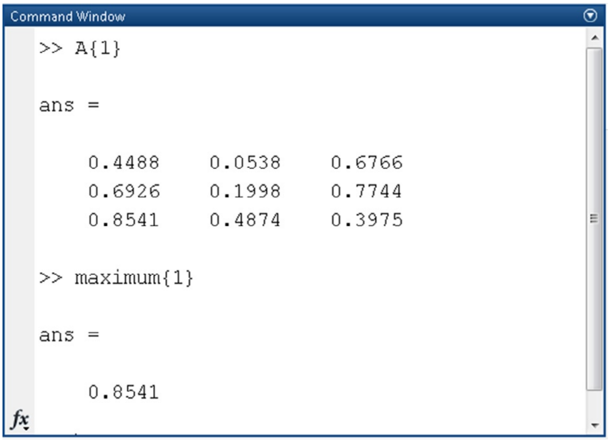
- Define variables on MATLAB workers inside an *spmd* statement. These variables are accessible by the MATLAB client after the *spmd* statement, as shown in the previous example (Figs. 3.8 and 3.9).



```
A =

   Lab 1: class = double, size = [3  3]
   Lab 2: class = double, size = [2  2]
   Lab 3: class = double, size = [2  2]
   Lab 4: class = double, size = [2  2]
   Lab 5: class = double, size = [2  2]
   Lab 6: class = double, size = [2  2]
   Lab 7: class = double, size = [2  2]
   Lab 8: class = double, size = [4  4]

>> maximum

maximum =

   Lab 1: class = double, size = [1  1]
   Lab 2: class = double, size = [1  1]
   Lab 3: class = double, size = [1  1]
   Lab 4: class = double, size = [1  1]
   Lab 5: class = double, size = [1  1]
```

FIG. 3.8

Composite objects.



```
>> A{1}

ans =

    0.4488    0.0538    0.6766
    0.6926    0.1998    0.7744
    0.8541    0.4874    0.3975

>> maximum{1}

ans =

    0.8541
```

FIG. 3.9

Retrieving *composite* objects on the MATLAB client.

- Use the *Composite* function on the MATLAB client. The MATLAB client may create a *composite* object prior to an *spmd* statement, and the MATLAB workers can access the *composite* object inside an *spmd* statement.

```
>> A = Composite();
>> for i = 1:numel(A)
       A{i} = rand(3, 3);
   end
```

Table 3.3 Functions used by MATLAB workers to communicate with each other

Function	Description
gcat	Global concatenation of an array performed across all MATLAB workers.
gop	Global reduction using binary associative operation performed across all MATLAB workers.
gplus	Global addition performed across all MATLAB workers.
labBarrier	Block execution until all MATLAB workers reach this call.
labBroadcast	Send data to all MATLAB workers or receive data sent to all MATLAB workers.
labProbe	Test to see if messages are ready to be received from other MATLAB worker.
labReceive	Receive data from another MATLAB worker.
labSend	Send data to another MATLAB worker.
labSendReceive	Simultaneously send data to and receive data from another MATLAB worker.

```
>> spmd
       maximum = max(max(A));
   end
```

The *composite* objects are retained on the MATLAB workers until they are cleared on the MATLAB client or until the pool is closed. Moreover, multiple *spmd* statements can use *composite* objects defined in previous *spmd* statements.

The MATLAB workers can communicate with each other using a number of functions. First of all, *distributed* and *codistributed* arrays can be used to partition large data sets. *Distributed* and *codistributed* arrays will be thoroughly presented in Section 3.4. Moreover, a number of functions can be used from the MATLAB workers to communicate with each other (Table 3.3).

You should also consider the following limitations/concerns before parallelizing your application using an *spmd* statement:

- When running an *spmd* statement in a computer cluster, the communication cost will be greater than running the same code on local MATLAB workers.
- Any graphical output, for example, *plot*, will not be displayed at all because the MATLAB workers are sessions without graphical output.
- Be careful when using *cd*, *addpath*, and *rmpath* on an *spmd* statement as the MATLAB search path might not be the same on all MATLAB workers.
- If an error occurs on a worker, then all MATLAB workers terminate.
- The body of an *spmd* statement cannot make any direct reference to a nested function, but it can call a nested function by means of a variable defined as a function handle to the nested function.

- The body of an *spmd* statement cannot define an anonymous function, but it can reference an anonymous function by means of a function handle.
- The body of an *spmd* statement cannot directly contain another *spmd* statement, but it can call a function that contains another *spmd* statement. However, the inner *spmd* statement runs serially in a single thread on the worker running its containing function.
- The body of a *parfor-loop* cannot contain an *spmd* statement and an *spmd* statement cannot contain a *parfor-loop*.
- The body of an *spmd* statement cannot contain *break* and *return* statements.
- The body of an *spmd* statement cannot contain global or persistent variable declarations.

There are mainly two types of applications in which the use of an *spmd* statement will improve their computation time:

- Applications that take a long time to execute: several MATLAB workers compute solutions simultaneously.
- Applications that use large data sets: data is distributed to multiple MATLAB workers.

Example: Monte Carlo simulation to approximate the area of a figure

Let's consider again the example presented in Section 3.2. Initially, let's use an *spmd* statement to parallelize the *test* function.

```
function area = testspmd(n)
% Filename: testspmd.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (multi-threaded version using spmd)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = testspmd(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

spmd(8)
   iterations = floor(n / 8);
   if labindex <= mod(n, 8)
       iterations = iterations + 1;
   end
   counter = 0;
   for i = 1:iterations
        x = 2 * rand;
        y = 8 * rand;
        counter = counter + sum(y < (x * (x - 2) ^ 6));
   end
end
counterStruct = gplus(counter);
```

```
counter = 0;
for i = 1:8
   counter = counter + counterStruct{i};
end
area = counter / n * 2 * 8;
end
```

The *testspmd* function runs in 35.20 seconds using 8 MATLAB workers, a ~5.37× speedup compared to the *test* function and almost the same execution time with the *testparfor* function.

Next, let's use an *spmd* statement to parallelize the *test2* function.

```
function area = testspmd2(n)
% Filename: testspmd2.m
% Description: This function computes the area of the
% definite integral int(x * (x - 2) ^ 6, x = 0..2)
% (multi-threaded semi-vectorized version using spmd)
% Authors: Ploskas, N., & Samaras, N.
% Syntax: area = testspmd2(n)
% Input:
%   -- n: the number of random points to generate
% Output:
%   -- area: the area of the integral

n = n / 100;
spmd(8)
   iterations = floor(100 / 8);
   if labindex <= mod(100, 8)
       iterations = iterations + 1;
   end
   counter = 0;
   for i = 1:iterations
        x = 2 * rand(n, 1);
        y = 8 * rand(n, 1);
        counter = counter + sum(y < (x .* (x - 2) .^6 ));
   end
end
counterStruct = gplus(counter);
counter = 0;
for i = 1:8
   counter = counter + counterStruct{i};
end
area = counter / (n * 100) * 2 * 8;
end
```

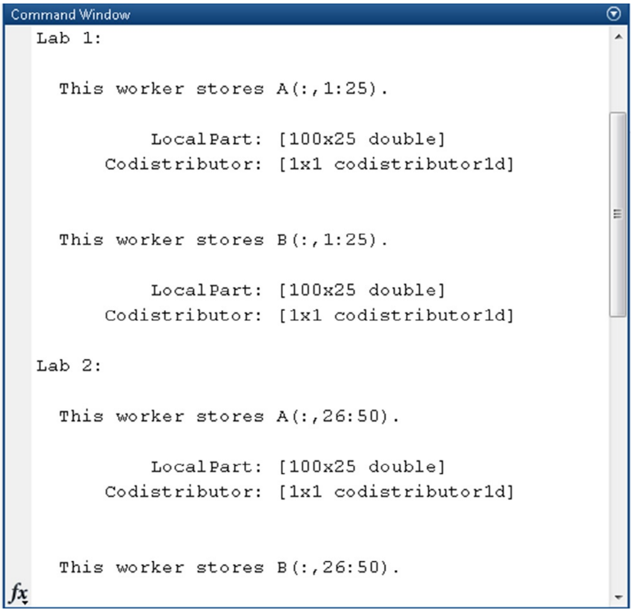
The *testparfor2* function runs in 25.79 seconds using 8 MATLAB workers, a ~7.32× speedup compared to the *test* function, a ~2.20× speedup compared to the *test2* function and almost the same execution time with the *testparfor2* function.

3.4 DISTRIBUTED AND CODISTRIBUTED ARRAYS

The MATLAB client can create a *distributed* array and distribute it across all MATLAB workers. A *codistributed* array can be created inside an *spmd* statement and partitioned among the MATLAB workers. The difference between *distributed* and *codistributed* arrays is one of perspective. *Codistributed* arrays are partitioned among the MATLAB workers from which you execute code to create or manipulate them. *Distributed* arrays are partitioned among workers in the parallel pool. When a *codistributed* array is created in an *spmd* statement, it can be accessed as a *distributed* array on the MATLAB client. When a *distributed* array is created on the MATLAB client, it can be accessed as a *codistributed* array inside an *spmd* statement. The details of distribution cannot be controlled when creating a *distributed* array; a *distributed* array is distributed in one dimension, along the last non-singleton dimension, and as evenly as possible along that dimension among the MATLAB workers. On the other hand, the details of distribution can be controlled when creating *a codistributed* array.

A *distributed* array can be created using one of the following ways:

- Use the *distributed* function to distribute an existing array from the MATLAB client's workspace to the MATLAB workers of a parallel pool. In the following example, *A* and *B* are *distributed* arrays. Each MATLAB worker stores only a part of *A* and *B* (Fig. 3.10).



```
Lab 1:

  This worker stores A(:,1:25).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]


  This worker stores B(:,1:25).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]

Lab 2:

  This worker stores A(:,26:50).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]


  This worker stores B(:,26:50).
```

FIG. 3.10

Part of *distributed* arrays stored on each MATLAB worker.

```
>> parpool('local', 4);
>> A = rand(100, 100);
>> A = distributed(A);
>> spmd
      B = A * 2;
   end
>> delete(gcp);
```

- Use any of the overloaded *distributed* object methods to directly construct a *distributed* array on the MATLAB workers (Table 3.4). In the following example, *A* and *B* are *distributed* arrays. Each MATLAB worker stores only a part of *A* and *B* (Fig. 3.10).

```
>> parpool('local', 4);
>> A = rand(100, 100, 'distributed');
>> spmd
      B = A * 2;
>> end
>> delete(gcp);
```

- Create a *codistributed* array inside an *spmd* statement and access it as a *distributed* array outside the *spmd* statement (see examples for *codistributed* arrays later in this section).

A *codistributed* array can be created using one of the following ways:

- Use the *codistributed* function inside an *spmd* statement, a communicating job or *pmode* to codistribute data already existing on the MATLAB workers running that job. In the following example, *A* and *B* are *codistributed* arrays. Each MATLAB worker stores only a part of *A* and *B* (Fig. 3.10).

```
>> parpool('local', 4);
>> spmd
      A = rand(100, 100);
      A = codistributed(A);
      B = A * 2;
   end
>> delete(gcp);
```

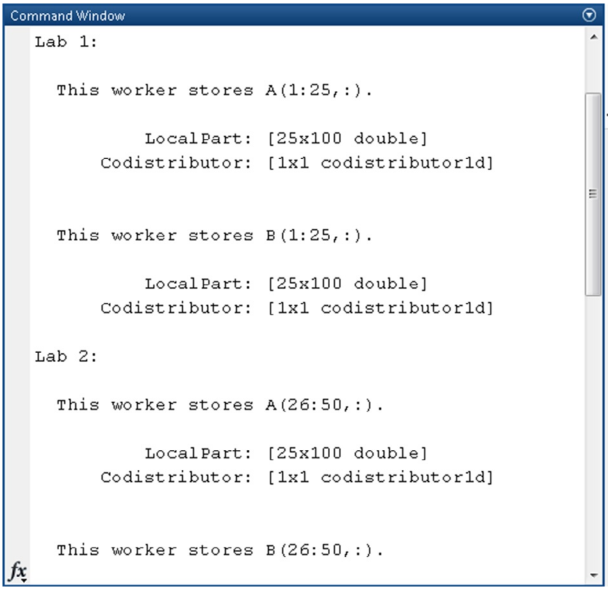
In the preceding code, *codistributed(A)*, which is the same as *codistributed(A, codistributor('1d', 2))*, tells MATLAB to distribute array *A* along its second dimension, that is, columns. If you want to distribute array *A* along its first dimension, that is, rows, use *codistributed(A, codistributor('1d', 1))* (Fig. 3.11).

As an alternative to distributing by a single dimension of rows or columns, you can distribute an array by blocks using '2dbc' or two-dimensional block-cyclic distribution. In the following example, *A* and *B* are distributed with a 50×50 block in 2-by-2 arrangement (Figs. 3.12 and 3.13).

```
>> parpool('local', 4);
>> spmd
      A = rand(100, 100);
```

Table 3.4 Overloaded MATLAB functions for *distributed* arrays

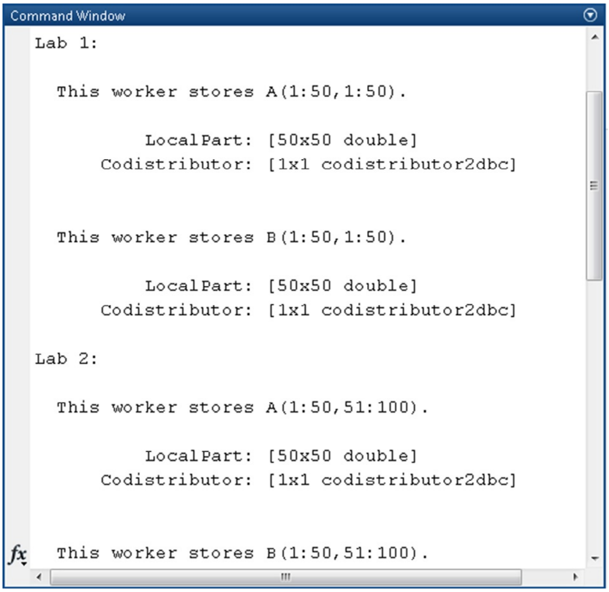
Function	Description
distributed.cell	Create a distributed cell array.
distributed.colon(a, d, b)	Create a distributed array from the vector a:d:b.
distributed.eye	Create a distributed identity matrix.
distributed.false	Create a distributed array of logical zeros.
distributed.Inf	Create a distributed array with Inf values in all elements.
distributed.linspace	Create a distributed linearly spaced vector.
distributed.logspace	Create a distributed logarithmically spaced vector.
distributed.NaN	Create a distributed array with Nan values in all elements.
distributed.ones	Create a distributed array of ones.
distributed.rand	Create a distributed array of uniformly distributed pseudo-random numbers.
distributed.randi	Create a distributed array of uniformly distributed pseudo-random integer numbers.
distributed.randn	Create a distributed array of normally distributed pseudo-random numbers.
distributed.spalloc	Allocate space for a sparse distributed array.
distributed.speye	Create a distributed sparse identity array.
distributed.sprand	Create a distributed sparse array of uniformly distributed pseudo-random values.
distributed.sprandn	Create a distributed sparse array of normally distributed pseudo-random values.
distributed.true	Create a distributed array of logical ones.
distributed.zeros	Create a distributed array of zeros.
eye(..., 'distributed')	Create a distributed identity matrix.
false(..., 'distributed')	Create a distributed array of logical zeros.
Inf(..., 'distributed')	Create a distributed array with Inf values in all elements.
NaN(..., 'distributed')	Create a distributed array with Nan values in all elements.
ones(..., 'distributed')	Create a distributed array of ones.
rand(..., 'distributed')	Create a distributed array of uniformly distributed pseudo-random numbers.
randi(..., 'distributed')	Create a distributed array of uniformly distributed pseudo-random integer numbers.
randn(..., 'distributed')	Create a distributed array of normally distributed pseudo-random numbers.
true(..., 'distributed')	Create a distributed array of logical ones.
zeros(..., 'distributed')	Create a distributed array of zeros.



```
Lab 1:

  This worker stores A(1:25,:).

          LocalPart: [25x100 double]
      Codistributor: [1x1 codistributor1d]


  This worker stores B(1:25,:).

          LocalPart: [25x100 double]
      Codistributor: [1x1 codistributor1d]

Lab 2:

  This worker stores A(26:50,:).

          LocalPart: [25x100 double]
      Codistributor: [1x1 codistributor1d]


  This worker stores B(26:50,:).
```

FIG. 3.11

Codistributed arrays along their first dimension.



```
Lab 1:

  This worker stores A(1:50,1:50).

          LocalPart: [50x50 double]
      Codistributor: [1x1 codistributor2dbc]


  This worker stores B(1:50,1:50).

          LocalPart: [50x50 double]
      Codistributor: [1x1 codistributor2dbc]

Lab 2:

  This worker stores A(1:50,51:100).

          LocalPart: [50x50 double]
      Codistributor: [1x1 codistributor2dbc]


  This worker stores B(1:50,51:100).
```

FIG. 3.12

Codistributed arrays using ‘2dbc’ distribution.

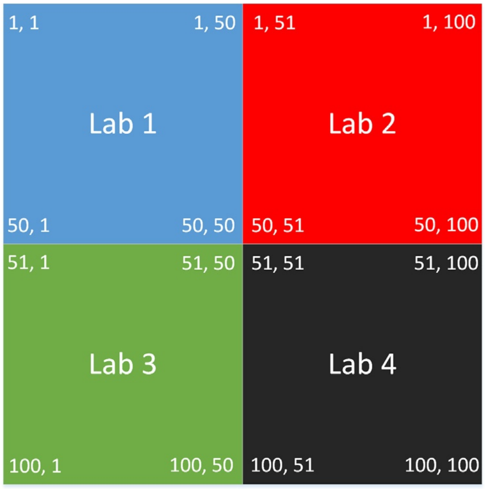


FIG. 3.13

Codistributed arrays using '2dbc' distribution.

```
         A = codistributed(A, codistributor('2dbc', [2 2], 50));
         B = A * 2;
>> end
>> delete(gcp);
```

- Use any of the overloaded *codistributed* object methods to directly construct a *codistributed* array on the MATLAB workers (Table 3.5).

```
>> parpool('local', 4);
>> spmd
         A = rand(100, 100, codistributor1d());
         B = A * 2;
>> end
>> delete(gcp);
```

- Create a *distributed* array outside an *spmd* statement and access it as a *codistributed* array inside the *spmd* statement running on the same pool (see examples for *distributed* arrays earlier in this section).

Indexing into a non-distributed array is straightforward; each dimension is indexed within the range of 1 to the final subscript given by the *end* keyword. The length of any dimension can be determined using either *size* or *length* function. On the other hand, these values are not so easily obtained for *codistributed* arrays because the index range depends on the distribution scheme that was used to distribute the *codistributed* array. MATLAB provides the *globalIndices* function, which provides a correlation between the local and global indexing of the *codistributed* array.

Table 3.5 Overloaded MATLAB functions for *codistributed* arrays

Function	Description
codistributed.build	Build a codistributed array from local parts.
codistributed.cell	Create a codistributed cell array.
codistributed.colon(a, d, b)	Create a codistributed array from the vector a:d:b.
codistributed.eye	Create a codistributed identity matrix.
codistributed.false	Create a codistributed array of logical zeros.
codistributed.linspace	Create a codistributed linearly spaced vector.
codistributed.logspace	Create a codistributed logarithmically spaced vector.
codistributed.Inf	Create a codistributed array with Inf values in all elements.
codistributed.NaN	Create a codistributed array with Nan values in all elements.
codistributed.ones	Create a codistributed array of ones.
codistributed.rand	Create a codistributed array of uniformly distributed pseudo-random numbers.
codistributed.randi	Create a codistributed array of distributed pseudo-random integer numbers.
codistributed.randn	Create a codistributed array of normally distributed pseudo-random numbers.
codistributed.spalloc	Allocate space for a sparse codistributed array.
codistributed.speye	Create a codistributed sparse identity array.
codistributed.sprand	Create a codistributed sparse array of uniformly distributed pseudo-random values.
codistributed.sprandn	Create a codistributed sparse array of normally distributed pseudo-random values.
codistributed.true	Create a codistributed array of logical ones.
codistributed.zeros	Create a codistributed array of zeros.
eye(..., 'codistributed')	Create a codistributed identity matrix.
false(..., 'codistributed')	Create a codistributed array of logical zeros.
Inf(..., 'codistributed')	Create a codistributed array with Inf values in all elements.
NaN(..., 'codistributed')	Create a codistributed array with Nan values in all elements.
ones(..., 'codistributed')	Create a codistributed array of ones.
rand(..., 'codistributed')	Create a codistributed array of uniformly distributed pseudo-random numbers.
randi(..., 'codistributed')	Create a codistributed array of distributed pseudo-random integer numbers.
randn(..., 'codistributed')	Create a codistributed array of normally distributed pseudo-random numbers.
sparse(..., codist)	Create a sparse codistributed matrix.
true(..., 'codistributed')	Create a codistributed array of logical ones.
zeros(..., 'codistributed')	Create a codistributed array of zeros.

Moreover, MATLAB offers *for drange* loop to iterate through a *codistributed* array, as shown below:

```
>> n = 100;
>> A = 1:n;
>> spmd
      B = zeros(1, n, codistributor());
      for i = drange(1:n)
         B(i) = A(i) * 2;
      end
   end
```

Finally, many functions in MATLAB are overloaded so that they operate on *codistributed* arrays in the same way they operate on non-distributed arrays. For a full list, see [5].

3.5 INTERACTIVE PARALLEL DEVELOPMENT (*pmode*)

MATLAB provides the *pmode* functionality where you can work interactively with a communicating job running simultaneously on multiple MATLAB workers. *pmode* and *spmd* are very similar; the main difference to *spmd* is that *pmode* does not allow you to freely interleave serial and parallel work as *spmd* does. When a *pmode* session is terminated, its job is destroyed and all data stored on the MATLAB workers lost. Commands that are typed at the *pmode* prompt are executed on all MATLAB workers at the same time. Each worker has its own workspace. All communication functions supported in *spmd* statements can also be used in *pmode* and the variables can be transferred between the MATLAB client and the MATLAB workers. As with *spmd*, MATLAB workers are sessions without display. Moreover, the MATLAB workers running *pmode* can be on a computer cluster.

We can start *pmode* using the local profile with 4 local workers (Fig. 3.14):

```
>> pmode start local 4
```

Codistributed arrays can be distributed among the MATLAB workers (Fig. 3.15):

```
>> A = rand(100, 100, codistributor());
```

The *gather* function can be used to gather an entire array into the workspace of all MATLAB workers. Moreover, the *client2lab* function can be used to copy a variable from the MATLAB client to all/specified MATLAB workers, while *lab2client* function can be used to copy a variable from a MATLAB worker to the MATLAB client. Note that a *codistributed* array cannot be transferred from a MATLAB worker to the MATLAB client because the MATLAB worker stores only a local portion of the *codistributed* array. To overcome that limitation, the *gather* function must first be used to assemble the entire array into the workspace of all MATLAB workers and then use the *lab2client* function to transfer the *codistributed* array to the MATLAB client.

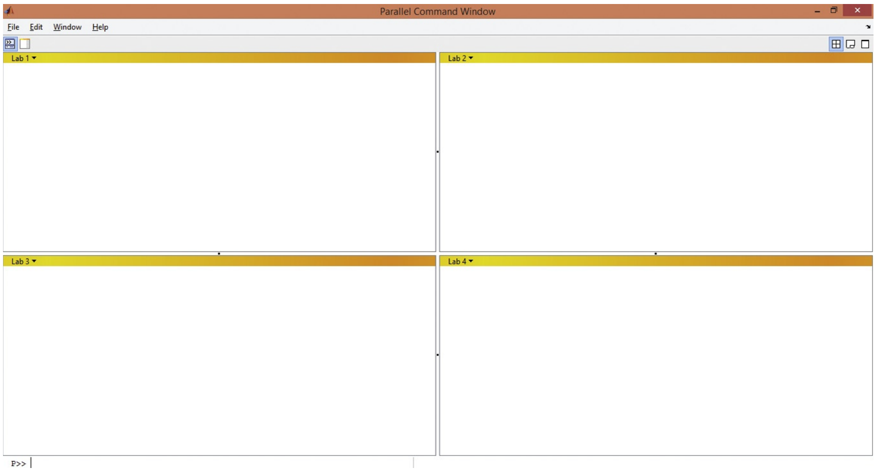
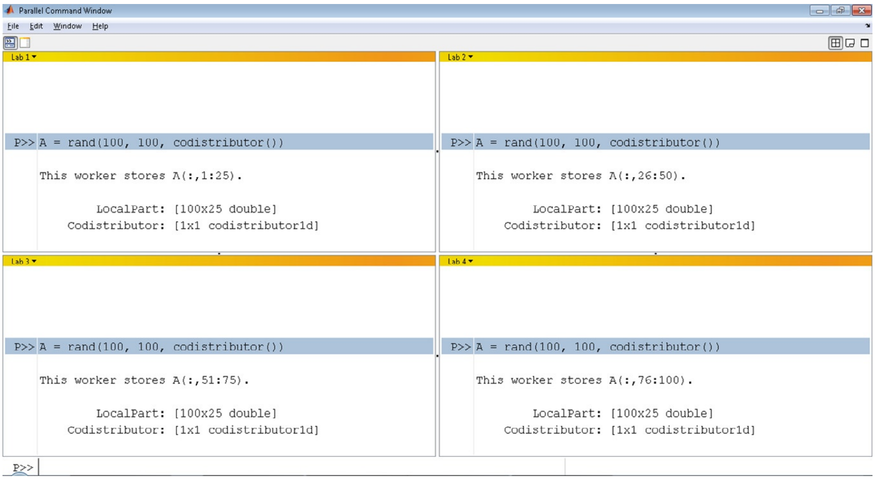


FIG. 3.14

pmode's parallel command window.



```
P>> A = rand(100, 100, codistributor())

  This worker stores A(:,1:25).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]
```

```
P>> A = rand(100, 100, codistributor())

  This worker stores A(:,26:50).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]
```

```
P>> A = rand(100, 100, codistributor())

  This worker stores A(:,51:75).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]
```

```
P>> A = rand(100, 100, codistributor())

  This worker stores A(:,76:100).

          LocalPart: [100x25 double]
      Codistributor: [1x1 codistributor1d]
```

```
P>>
```

FIG. 3.15

Codistributed array in *pmode*'s parallel command window.

3.6 GPU COMPUTING

For the sake of completeness, we have also included this section about GPU computing because the Parallel Computing Toolbox enables you to program MATLAB to use GPUs. However, we will not present MATLAB's GPU capabilities in this section because GPU computing will be extensively discussed in the next chapters of this book.

3.7 CLUSTERS AND JOB SCHEDULING

The Parallel Computing Toolbox and MATLAB Distributed Computing Server let you solve task- and data-parallel algorithms on many multicore and multiprocessor computers. A job is a large operation that you need to perform in MATLAB. A job can be divided into segments called tasks. These tasks can be identical or not. The MATLAB session in which the job is defined is called the MATLAB client. The Parallel Computing Toolbox should be installed on the MATLAB client. If the job is executed on a local cluster, no other MATLAB toolbox is needed. On the other hand, if the job is executed on a computer cluster, the MATLAB Distributed Computing Server should be installed on the machines of the cluster (Fig. 3.16). A job scheduler is needed to coordinate the execution of jobs and the evaluation of their tasks. MATLAB provides the MATLAB Job Scheduler (MJS) for this reason. However, the use of the MJS is optional; third-party schedulers, such as Microsoft Windows HPC Server and IBM Platform LSF, can also be used [13].

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 job in its queue is promoted, demoted, canceled or deleted. The MJS distributes the tasks to MATLAB workers, and the MATLAB workers execute the tasks and return the results to the MJS. The MJS starts running a new job only when all tasks of a running job have been assigned to

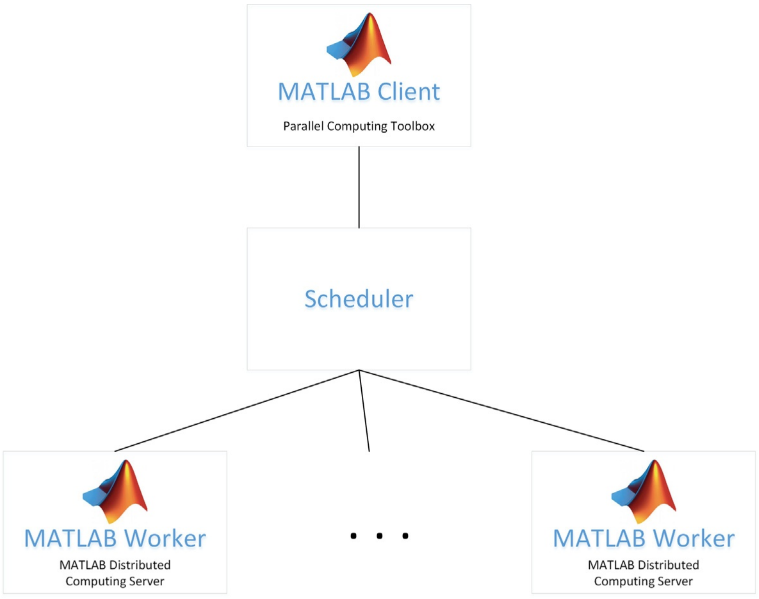

MATLAB Client
Parallel Computing Toolbox
Scheduler
MATLAB Worker
MATLAB Distributed Computing Server
. . .
MATLAB Worker
MATLAB Distributed Computing Server

FIG. 3.16

Running a job on a computer cluster.

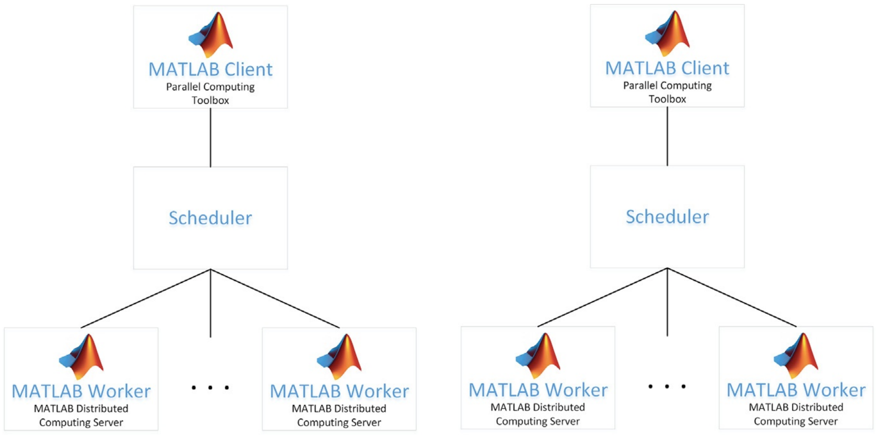

MATLAB Client
Parallel Computing Toolbox
Scheduler
MATLAB Worker
MATLAB Distributed Computing Server
MATLAB Worker
MATLAB Distributed Computing Server
MATLAB Client
Parallel Computing Toolbox
Scheduler
MATLAB Worker
MATLAB Distributed Computing Server
MATLAB Worker
MATLAB Distributed Computing Server

FIG. 3.17

Multiple MATLAB clients and MJSs.

MATLAB workers. In an independent job, the MATLAB workers evaluate tasks one at a time as available, perhaps simultaneously, perhaps not. In a communicating job, the MATLAB workers evaluate tasks simultaneously. Finally, the MJS returns all the results of all the tasks of a job to the MATLAB client.

In a large network, several MATLAB clients and MJSs may be used. Each client can create, run, and access jobs on any MJS, but a MATLAB worker is registered with and dedicated to only one MJS at a time (Fig. 3.17).

A cluster profile can be created either automatically (by discovering a MATLAB Distributed Computing Server cluster on the network) or manually. To create a cluster profile manually, on the *Home* tab in the *Environment* section, click *Parallel* → *Manage Cluster Profiles*. The following information should be given to create a cluster profile:

- Description: the description of the cluster profile.
- Host: the hostname of the machine where the MJS is running (required).
- MJSName: the name of the MJS (required if multiple MJSs exist on the host machine).
- Username: the username for MJS access (required if the MJS uses security level 1–3).
- LicenseNumber: the license number.
- AutoAttachFiles: if code files will automatically be sent to the cluster.
- AttachedFiles: the files and folders to copy from the MATLAB client to cluster nodes.
- AdditionalPaths: the folders to add to the MATLAB workers’ search path.
- NumWorkersRange: the range of number of MATLAB workers to run a job.

- CaptureDiary: if the Command Window output will be returned.
- RestartWorker: if the MATLAB workers will be restarted before a job will be executed.
- MaximumRetries: the maximum number of times that a job will try to run again if it fails.
- DefaultTimeout: the master timeout in seconds when a job or a task timeout is not defined.
- JobTimeout: the job timeout in seconds.
- TaskTimeout: the task timeout in seconds.
- JobFinishedFcn: the function that runs on the client when a job reaches the finished state.
- JobRunningFcn: the function that runs on the client when a job reaches the running state.
- JobQueuedFcn: the function that runs on the client when a job reaches the queued state.
- TaskFinishedFcn: the function that runs on the client when a task reaches the finished state.
- TaskRunningFcn: the function that runs on the client when a task reaches the running state.

When a job is created and executed, it progresses through the following stages:

- Pending: the job's first state is pending. This is when a job is defined by adding tasks to it.
- Queued: when a job is submitted, the MJS places the job in a queue.
- Running: when a job reaches the top of the queue, the MJS distributes the job's tasks to MATLAB workers for evaluation.
- Finished: when all of a job's tasks have been evaluated, the job is moved to the finished state.
- Failed: if a third-party scheduler is used instead of the MJS, a job might fail if the scheduler encounters an error while attempting to execute its commands or access necessary files.
- Deleted: when a job's data has been removed from its data location or from the MJS, the state of the job on the MATLAB client is deleted.

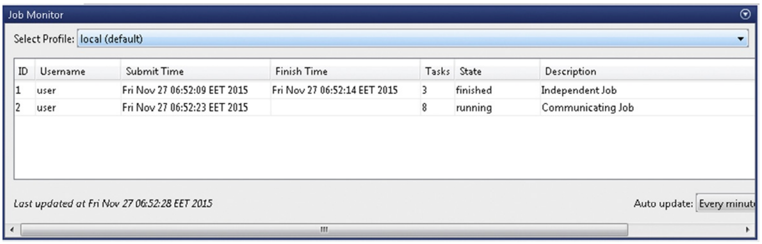
As already mentioned, two type of jobs can be created:

- **Independent job**: an independent job is one whose tasks do not directly communicate with each other, that is, its tasks are independent of each other. The tasks do not need to run simultaneously and a MATLAB worker might run several tasks of the same job in succession. The following example creates (*createJob* function) and executes (*submit* function) an independent job with three simple tasks (the calculation of a vector's maximum and minimum element and the calculation of the sum of a vector's elements). Finally, the results from all tasks of the job are collected (*fetchOutputs* function) after the job is finished (*wait* function).

```
>> parallel.defaultClusterProfile('local');
>> c = parcluster();
>> j = createJob(c);
>> createTask(j, @max, 1, {rand(1, 10)});
>> createTask(j, @min, 1, {rand(1, 10)});
>> createTask(j, @sum, 1, {rand(1, 10)});
>> submit(j);
>> wait(j)
>> results = fetchOutputs(j)
```

- **Communicating job**: a communicating job is one in which the MATLAB 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 MATLAB workers, usually with different data. The task is duplicated on each MATLAB worker, so each MATLAB worker can perform the task on a different set of data or on a particular segment of a large data set. The following example creates (*createCommunicatingJob* function) and executes (*submit* function) a communicating job with a simple task (the calculation of the sum of a magic square). The task is defined in *totalSum* function which is attached to all MATLAB workers. The first MATLAB worker creates and sends to all other MATLAB workers a magic square (*labBroadcast* function), each of which calculates the sum of one column of the matrix. All of these column sums are combined to calculate the total sum of the elements of the original magic square (*gplus* function). Finally, the results from all tasks of the job are collected (*fetchOutputs* function) after the job is finished (*wait* function).

```
function total_sum = totalSum
   if labindex == 1
        A = labBroadcast(1, magic(numlabs));
   else
        A = labBroadcast(1);
   end
   colSum = sum(A(:, labindex));
   total_sum = gplus(colSum);
end

>> parallel.defaultClusterProfile('local');
>> c = parcluster();
>> j = createCommunicatingJob(c, 'Type', 'SPMD');
>> j.AttachedFiles = {'totalSum.m'};
>> j.NumWorkersRange = 8;
>> createTask(j, @totalSum, 1, {})
>> submit(j);
>> wait(j)
>> results = fetchOutputs(j)
```



ID	Username	Submit Time	Finish Time	Tasks	State	Description
1	user	Fri Nov 27 06:52:09 EET 2015	Fri Nov 27 06:52:14 EET 2015	3	finished	Independent Job
2	user	Fri Nov 27 06:52:23 EET 2015		8	running	Communicating Job

Last updated at Fri Nov 27 06:52:28 EET 2015

Auto update: Every minut

FIG. 3.18

Job monitor.

The jobs can be monitored using the Job Monitor by clicking *Parallel* → *Monitor Jobs* on the *Home* tab in the *Environment* section (Fig. 3.18).

3.8 CHAPTER REVIEW

This chapter presented the key features of MATLAB's Parallel Computing Toolbox. The Parallel Computing Toolbox allows users to solve computationally- and data-intensive problems using multicore and multiprocessor computers, computer clusters, and GPUs. The *parfor-loop* and the *spmd* statement, which enable users to parallelize their code, have been presented in detail. Moreover, the *pmode* functionality, where you can work interactively with a communicating job running simultaneously on multiple MATLAB workers, was also introduced. Finally, the combination of the Parallel Computing Toolbox with the Distributed Computing Server has been described to show how codes can be executed on computer clusters, grids, and clouds.

To sum up, the Parallel Computing Toolbox offers many features to execute task- and data-parallel applications on a single core, a multicore processor, or a computer cluster. In general, Parallel Computing Toolbox can be used to improve the performance of the following types of applications: (i) applications that involve repetitive segments of code, (ii) applications that run a series of independent tasks, (iii) applications evaluating the same code on multiple data sets, and (iv) applications involving data that is too large for your computer's memory.