```
        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.

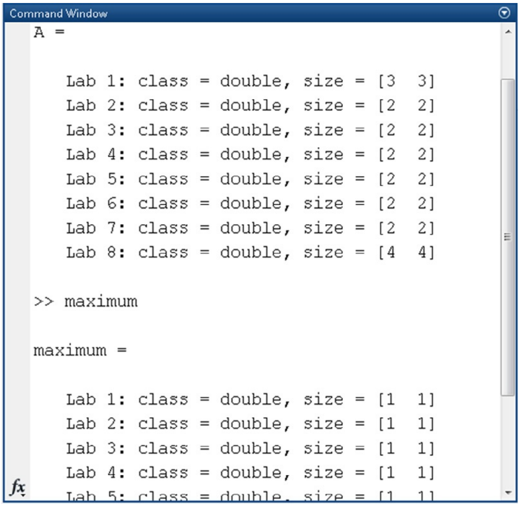
Each MATLAB worker used in an *spmd* statement has a unique value of *labindex* that can be used to run codes on specific MATLAB workers or access unique data. The total number of MATLAB workers used in an *spmd* statement can be obtained using the *numlabs* value. For example, we can create different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

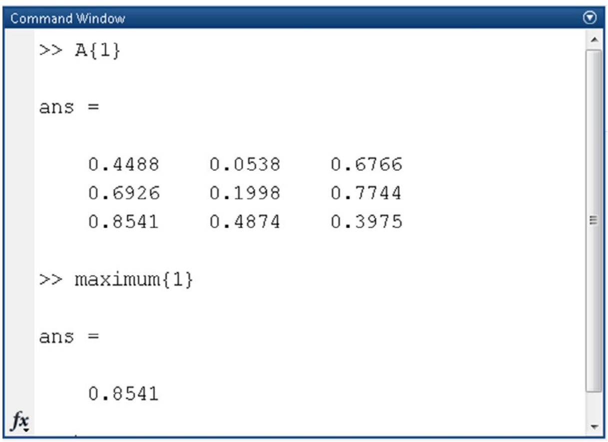
- Define variables on MATLAB workers inside an *spmd* statement. These variables are accessible by the MATLAB client after the *spmd* statement, as shown in the previous example (Figs. 3.8 and 3.9).



```
Command Window
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.



```
Command Window
>> 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.