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).

```
Command Window
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).
fx
```

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.

```
Command Window
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.

```
Command Window
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.

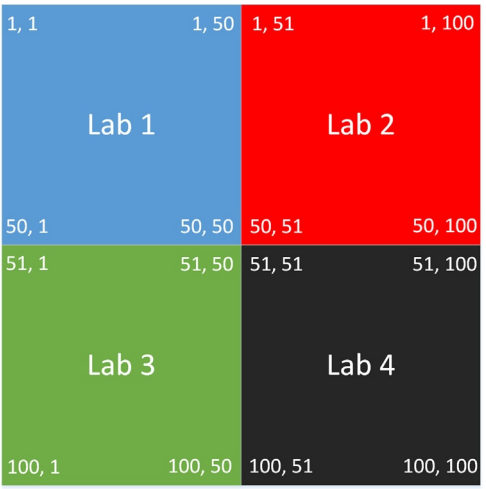

FIG. 3.13

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.