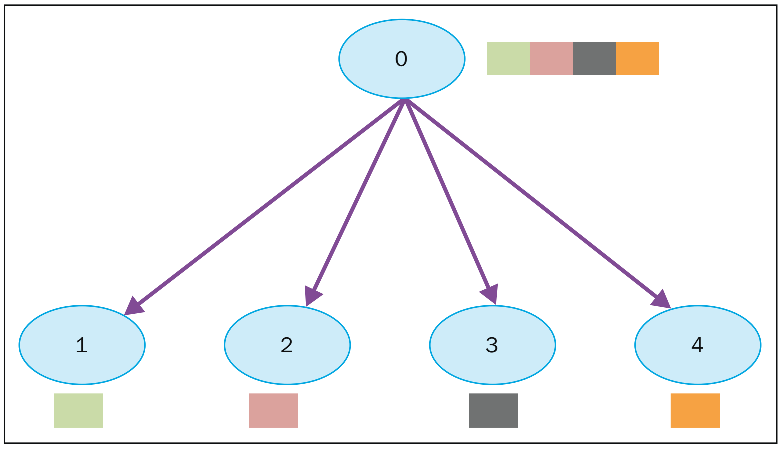


Scattering data from process 0 to processes 1, 2, 3, 4

The `comm.scatter` function takes the elements of the array and distributes them to the processes according to their rank, for which the first element will be sent to the process zero, the second element to the process 1, and so on. The function implemented in `mpi4py` is as follows:

```
recvbuf  = comm.scatter(sendbuf, rank_of_root_process)
```

How to do it...

In the next example, we see how to distribute data to different processes using the `scatter` functionality:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   array_to_share = [1, 2, 3, 4 ,5 ,6 ,7, 8 ,9 ,10]

else:
   array_to_share = None

recvbuf = comm.scatter(array_to_share, root=0)
print("process = %d" %rank + " recvbuf = %d " %array_to_share)
```

The output of the preceding code is, as follows:

```
C:\>mpiexec -n 10 python scatter.py
process = 0 variable shared  = 1
process = 4 variable shared  = 5
process = 6 variable shared  = 7
process = 2 variable shared  = 3
process = 5 variable shared  = 6
process = 3 variable shared  = 4
process = 7 variable shared  = 8
process = 1 variable shared  = 2
process = 8 variable shared  = 9
process = 9 variable shared  = 10
```

How it works...

The process of rank zero distributes the `array_to_share` data structure to other processes:

```
array_to_share = [1, 2, 3, 4 ,5 ,6 ,7, 8 ,9 ,10]
```

The `recvbuf` parameter indicates the value of the *i*th variable that will be sent to the *i*th process through the `comm.scatter` statement:

```
recvbuf = comm.scatter(array_to_share, root=0)
```

We also remark that one of the restrictions to `comm.scatter` is that you can scatter as many elements as the processors you specify in the execution statement. In fact attempting to scatter more elements than the processors specified (three in this example), you will get an error like this:

```
C:\> mpiexec -n 3 python scatter.py
Traceback (most recent call last):
  File "scatter.py", line 13, in <module>
    recvbuf = comm.scatter(array_to_share, root=0)
  File "Comm.pyx", line 874, in mpi4py.MPI.Comm.scatter (c:\users\utente\
appdata
\local\temp\pip-build-h14iaj\mpi4py\src\mpi4py.MPI.c:73400)
  File "pickled.pxi", line 658, in mpi4py.MPI.PyMPI_scatter (c:\users\
utente\app
data\local\temp\pip-build-h14iaj\mpi4py\src\mpi4py.MPI.c:34035)
  File "pickled.pxi", line 129, in mpi4py.MPI._p_Pickle.dumpv (c:\users\
utente\a
ppdata\local\temp\pip-build-h14iaj\mpi4py\src\mpi4py.MPI.c:28325)
ValueError: expecting 3 items, got 10
mpiexec aborting job...

job aborted:
rank: node: exit code[: error message]
0: Utente-PC: 123: mpiexec aborting job
1: Utente-PC: 123
2: Utente-PC: 123
```

There's more...

The `mpi4py` library provides two other functions that are used to scatter data:

- `comm.scatter(sendbuf, recvbuf, root=0)`: This sends data from one process to all other processes in a communicator.
- `comm.scatterv(sendbuf, recvbuf, root=0)`: This scatters data from one process to all other processes in a group that provides different amount of data and displacements at the sending side.

The `sendbuf` and `recvbuf` arguments must be given in terms of a list (as in, the point-to-point function `comm.send`):

```
buf = [data, data_size, data_type]
```

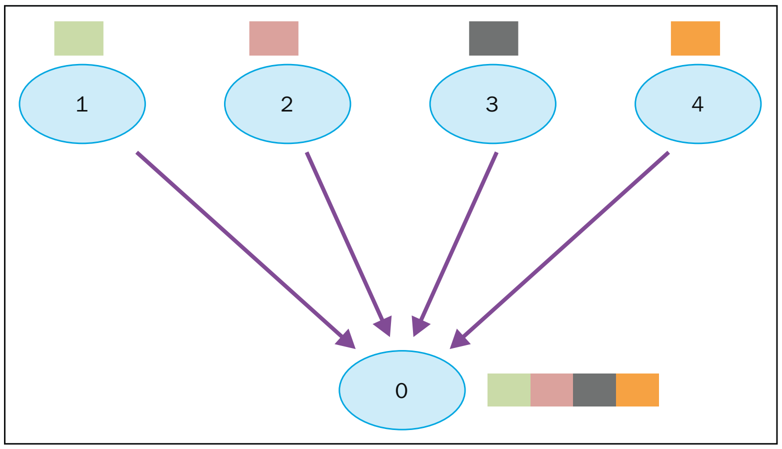
Here, `data` must be a buffer-like object of the size `data_size` and of the type `data_type`.

Collective communication using gather

The `gather` function performs the inverse of the `scatter` functionality. In this case, all processes send data to a root process that collects the data received. The `gather` function implemented in `mpi4py` is, as follows:

```
recvbuf  = comm.gather(sendbuf, rank_of_root_process)
```

Here, `sendbuf` is the data that is sent and `rank_of_root_process` represents the process receiver of all the data:



Gathering data from processes 1, 2, 3, 4

How to do it...

In the following example, we wanted to represent just the condition shown in the preceding figure. Each process builds its own data that is to be sent to the root processes that are identified with the rank zero:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
   print ("rank = %s " %rank +\
          "...receiving data to other process")
   for i in range(1,size):
      data[i] = (i+1)**2
      value = data[i]
      print(" process %s receiving %s from process %s"\
            %(rank , value , i))
```

Finally, we run the code with a group of processes equal to five:

```
C:\>mpiexec -n 5 python gather.py
rank = 0 ...receiving data to other process
 process 0 receiving 4 from process 1
 process 0 receiving 9 from process 2
 process 0 receiving 16 from process 3
 process 0 receiving 25 from process 4
```

The root process zero receives data from the other four processes, as we represented in the previous figure.

How it works...

We have *n* processes sending their data:

```
data = (rank+1)**2
```

If the rank of the process is zero, then the data is collected in an array:

```
if rank == 0:
   for i in range(1,size):
      data[i] = (i+1)**2
      value = data[i]
...
```

The gathering of data is given instead by the following function:

```
data = comm.gather(data, root=0)
```

There's more...

To collect data, mpi4py provides the following functions:

- **gathering to one task**: comm.Gather, comm.Gatherv, and comm.gather
- **gathering to all tasks**: comm.Allgather, comm.Allgatherv, and comm.allgather

Collective communication using Alltoall

The Alltoall collective communication combines the scatter and gather functionality. In mpi4py, there are three types of Alltoall collective communication:

- comm .Alltoall(sendbuf, recvbuf): The all-to-all scatter/gather sends data from all-to-all processes in a group
- comm.Alltoallv(sendbuf, recvbuf): The all-to-all scatter/gather vector sends data from all-to-all processes in a group, providing different amount of data and displacements
- comm.Alltoallw(sendbuf, recvbuf): Generalized all-to-all communication allows different counts, displacements, and datatypes for each partner

How to do it...

In the following example, we'll see a mpi4py implementation of comm.Alltoall. We consider a communicator group of processes, where each process sends and receives an array of numerical data from the other processes defined in the group:

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

a_size = 1
senddata = (rank+1)*numpy.arange(size,dtype=int)
recvdata = numpy.empty(size*a_size,dtype=int)
comm.Alltoall(senddata,recvdata)


print(" process %s sending %s receiving %s"\
      %(rank , senddata , recvdata))
```

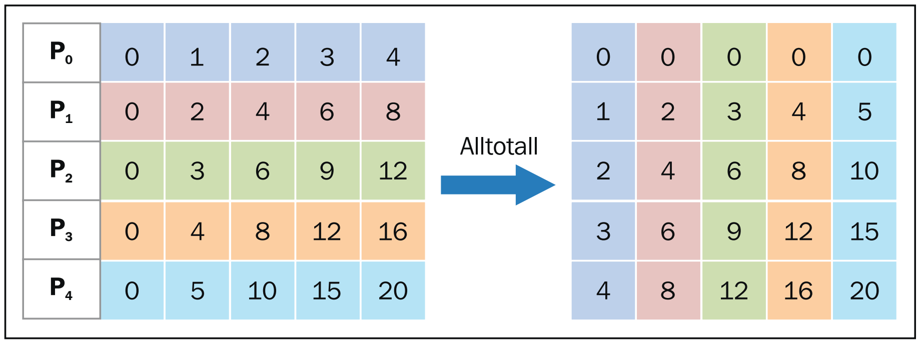
We run the code with a communicator group of five processes and the output we get is as follows:

```
C:\>mpiexec -n 5 python alltoall.py
process 0 sending [0 1 2 3 4] receiving [0 0 0 0 0]
process 1 sending [0 2 4 6 8] receiving [1 2 3 4 5]
process 2 sending [ 0  3  6  9 12] receiving [ 2  4  6  8 10]
process 3 sending [ 0  4  8 12 16] receiving [ 3  6  9 12 15]
process 4 sending [ 0  5 10 15 20] receiving [ 4  8 12 16 20]
```

How it works...

The `comm.alltoall` method takes the *i*th object from `sendbuf` of the task `j` and copies it into the *j*th object of the `recvbuf` argument of the task `i`.

We could also figure out what happened using the following schema:



The Alltoall collective communication

The following are our observations regarding the schema:

- The process **P0** contains the data array [0 1 2 3 4], where it assigns **0** to itself, **1** to the process **P1**, **2** to the process **P2**, **3** to the process **P3**, and **4** to the process **P4**.
- The process **P1** contains the data array [0 2 4 6 8], where it assigns **0** to **P0**, **2** to itself, **4** to the process **P2**, **6** to the process **P3**, and **8** to the process **P4**.
- The process **P2** contains the data array [0 3 6 9 12], where it assigns **0** to **P0**, **3** to the process **P1**, **6** to itself, **9** to the process **P3**, and **12** to the process **P4**.
- The process **P3** contains the data array [0 4 8 12 16], where it assigns **0** to **P0**, **4** to the process **P1**, **8** to the process **P2**, **12** to itself, and **16** to the process **P4**.
- The process **P4** contains the data array [0 5 10 15 20], where it assigns **0** to **P0**, **5** to the process **P1**, **10** to the process **P2**, **15** to the process, and **P3** and **20** to itself.

There's more...

All-to-all personalized communication is also known as total exchange. This operation is used in a variety of parallel algorithms, such as the Fast Fourier transform, matrix transpose, sample sort, and some parallel database join operations.

The reduction operation

Similar to `comm.gather`, `comm.reduce` takes an array of input elements in each process and returns an array of output elements to the root process. The output elements contain the reduced result.

In `mpi4py`, we define the reduction operation through the following statement:

```
comm.Reduce(sendbuf, recvbuf, rank_of_root_process, op = type_of_
reduction_operation)
```

We must note that the difference with the `comm.gather` statement resides in the `op` parameter, which is the operation that you wish to apply to your data, and the `mpi4py` module contains a set of reduction operations that can be used. Some of the reduction operations defined by MPI are:

- `MPI.MAX`: This returns the maximum element
- `MPI.MIN`: This returns the minimum element
- `MPI.SUM`: This sums up the elements
- `MPI.PROD`: This multiplies all elements
- `MPI.LAND`: This performs a logical operation and across the elements
- `MPI.MAXLOC`: This returns the maximum value and the rank of the process that owns it
- `MPI.MINLOC`: This returns the minimum value and the rank of the process that owns it

How to do it...

Now, we'll see how to implement a sum of an array of elements with the reduction operation `MPI.SUM`, using the reduction functionality. Each process will manipulate an array of size three. For array manipulation, we used the functions provided by the `numpy` Python module:

```
import numpy
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
```

```
size = comm.size
rank = comm.rank

array_size = 3
recvdata = numpy.zeros(array_size,dtype=numpy.int)
senddata = (rank+1)*numpy.arange(a_size,dtype=numpy.int)
print(" process %s sending %s " %(rank , senddata))
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
print ('on task',rank,'after Reduce:    data = ',recvdata)
```

It makes sense to run the code with a communicator group of three processes, that is, the size of the manipulated array. Finally, we obtain the result as:

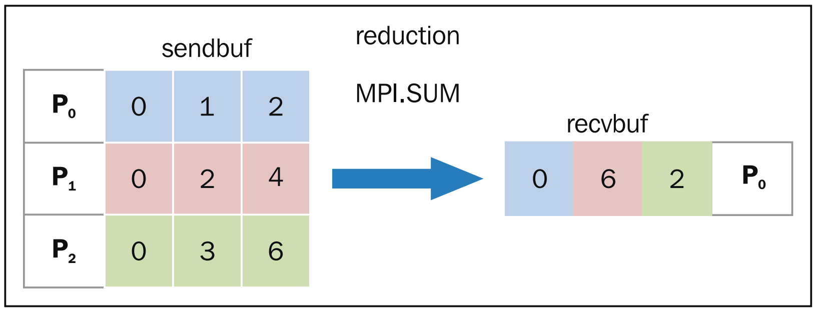
```
C:\>mpiexec -n 3 python reduction2.py
 process 2 sending [0 3 6]
on task 2 after Reduce:    data =  [0 0 0]
 process 1 sending [0 2 4]
on task 1 after Reduce:    data =  [0 0 0]
 process 0 sending [0 1 2]
on task 0 after Reduce:    data =  [ 0  6 12]
```

How it works...

To perform the reduction sum, we use the `comm.Reduce` statement and also identify with rank zero, the root process, which will contain `recvbuf`, that represents the final result of the computation:

```
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
```

Also, we must note that with the `op=MPI.SUM` option, we apply the sum operation to all of the elements of the column array. To better understand how the reduction operates, let's take a look at the following figure:



The reduction collective communication

The sending operation is as follows:

- The process **P0** sends the data array [0 1 2]
- The process **P1** sends the data array [0 2 4]
- The process **P2** sends the data array [0 3 6]

The reduction operation sums the *i*th elements of each task and then puts the result in the *i*th element of the array in the root process **P0**.

For the receiving operation, the process **P0** receives the data array [0 6 12].

How to optimize communication

An interesting feature that is provided by MPI concerns the virtual topologies. As already noted, all the communication functions (point-to-point or collective) refer to a group of processes. We have always used the `MPI_COMM_WORLD` group that includes all processes. It assigns a rank `0` to *n-1* for each process that belongs to a communicator of the size *n*. However, MPI allows us to assign a virtual topology to a communicator. It defines a particular assignment of labels to the different processes. A mechanism of this type permits you to increase the execution performance. In fact, if you build a virtual topology, then every node will communicate only with its virtual neighbor, optimizing the performance.

For example, if the rank was randomly assigned, a message could be forced to pass to many other nodes before it reaches the destination. Beyond the question of performance, a virtual topology makes sure that the code is more clear and readable. MPI provides two building topologies. The first construct creates Cartesian topologies, while the latter creates any kind of topologies. Specifically, in the second case, we must supply the adjacency matrix of the graph that you want to build. We will deal only with Cartesian topologies, through which it is possible to build several structures that are widely used: mesh, ring, toroid, and so on. The function used to create a Cartesian topology is, as follows:

```
comm.Create_cart((number_of_rows,number_of_columns))
```

Here, `number_of_rows` and `number_of_columns` specify the rows and columns of the grid that is to be made.

How to do it…

In the following example, we see how to implement a Cartesian topology of the size *M*×*N*. Also, we define a set of coordinates to better understand how all the processes are disposed:

```
from mpi4py import MPI
import numpy as np
```

```
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3
neighbour_processes = [0,0,0,0]
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    rank = comm.rank
    size = comm.size

    grid_rows = int(np.floor(np.sqrt(comm.size)))
    grid_column = comm.size // grid_rows


    if grid_rows*grid_column > size:
        grid_column -= 1
    if grid_rows*grid_column > size:
        grid_rows -= 1

    if (rank == 0) :
        print("Building a %d x %d grid topology:"\
              % (grid_rows, grid_column) )


    cartesian_communicator = \
                           comm.Create_cart( \
                               (grid_rows, grid_column), \
                               periods=(True, True), reorder=True)
    my_mpi_row, my_mpi_col = \
                cartesian_communicator.Get_coords\
                ( cartesian_communicator.rank )

    neighbour_processes[UP], neighbour_processes[DOWN]\
                             = cartesian_communicator.Shift(0, 1)
    neighbour_processes[LEFT],  \
                               neighbour_processes[RIGHT]  = \
                               cartesian_communicator.Shift(1, 1)
    print ("Process = %s \
row = %s \
column = %s ----> neighbour_processes[UP] = %s \
neighbour_processes[DOWN] = %s \
neighbour_processes[LEFT] =%s neighbour_processes[RIGHT]=%s" \
           %(rank, my_mpi_row, \
             my_mpi_col,neighbour_processes[UP], \
                neighbour_processes[DOWN], \
                neighbour_processes[LEFT] , \
                neighbour_processes[RIGHT]))
```

By running the script, we obtain the following result:

```
C:\>mpiexec -n 4 python virtualTopology.py
Building a 2 x 2 grid topology:
Process = 0 row = 0 column = 0 ---->
neighbour_processes[UP] = -1
neighbour_processes[DOWN] = 2
neighbour_processes[LEFT] =-1
neighbour_processes[RIGHT]=1

Process = 1 row = 0 column = 1 ---->
neighbour_processes[UP] = -1
neighbour_processes[DOWN] = 3
neighbour_processes[LEFT] =0
neighbour_processes[RIGHT]=-1

Process = 2 row = 1 column = 0 ---->
neighbour_processes[UP] = 0
neighbour_processes[DOWN] = -1
neighbour_processes[LEFT] =-1
neighbour_processes[RIGHT]=3

Process = 3 row = 1 column = 1 ---->
neighbour_processes[UP] = 1
neighbour_processes[DOWN] = -1
neighbour_processes[LEFT] =2
neighbour_processes[RIGHT]=-1
```

For each process, the output should read as: if `neighbour_processes = -1`, then it has no topological proximity; otherwise, `neighbour_processes` shows the rank of the process closely.

How it works...

The resulting topology is a mesh of *2×2* (refer to the previous figure for a mesh representation), the size of which is equal to the number of processes in the input, that is, four:

```
grid_rows = int(np.floor(np.sqrt(comm.size)))
grid_column = comm.size // grid_rows
    if grid_rows*grid_column > size:
        grid_column -= 1
    if grid_rows*grid_column > size:
        grid_rows -= 1
```

Then, the Cartesian topology is built:

```
cartesian_communicator = comm.Create_cart( \
    (grid_rows, grid_column), periods=(False, False), reorder=True)
...
```

To find out the position of the *i*th process, we use the `Get_coords()` method in the following form:

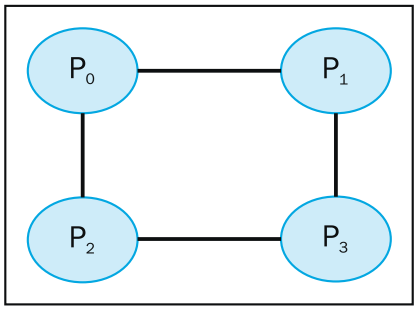
```
my_mpi_row, my_mpi_col = cartesian_communicator.Get_coords( cartesian_
communicator.rank )
For each process, in addition to their coordinates, we calculated
and got to know which processes are topologically closer. For
this purpose, we used the comm.Shift function comm.Shift (rank_
source,rank_dest)
```

In this form we have:

```
neighbour_processes[UP], neighbour_processes[DOWN] = \ cartesian_
communicator.Shift(0, 1)

neighbour_processes[LEFT],  neighbour_processes[RIGHT] = \ cartesian_
communicator.Shift(1, 1)
```

The obtained topology is shown in the following figure:



The virtual mesh 2x2 topology

There's more...

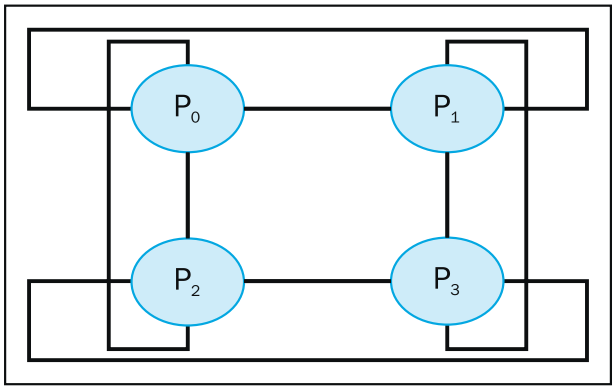
To obtain a toroidal topology of the size *M*×*N*, we need the following lines of code:

```
cartesian_communicator = comm.Create_cart( (grid_rows, grid_column),
periods=(True, True), reorder=True)
```

This corresponds to the following output:

```
C:\>mpiexec -n 4 python VirtualTopology.py
Building a 2 x 2 grid topology:
Process = 0 row = 0 column = 0 ---->
neighbour_processes[UP] = 2
neighbour_processes[DOWN] = 2
neighbour_processes[LEFT] =1
neighbour_processes[RIGHT]=1
Process = 1 row = 0 column = 1 ---->
neighbour_processes[UP] = 3
neighbour_processes[DOWN] = 3
neighbour_processes[LEFT] =0
neighbour_processes[RIGHT]=0
Process = 2 row = 1 column = 0 ---->
neighbour_processes[UP] = 0
neighbour_processes[DOWN] = 0
neighbour_processes[LEFT] =3 neighbour_processes[RIGHT]=3
Process = 3 row = 1 column = 1 ---->
neighbour_processes[UP] = 1
neighbour_processes[DOWN] = 1
neighbour_processes[LEFT] =2
neighbour_processes[RIGHT]=2
```

Also, it covers the topology represented here:



The virtual toroidal 2x2 topology