- `MPI.PROD`: This multiplies all elements.
- `MPI.LAND`: This performs the AND logical operation 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.

See also

At `http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/`, you can find a good tutorial on this topic and much more.

Optimizing communication

An interesting feature that is provided by MPI regards 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 of *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 an assignment of labels to the different processes: by building a virtual topology, each node will communicate only with its virtual neighbor, improving performance because it reduces execution times.

For example, if the rank was randomly assigned, then 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 clearer and more 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 only deal with Cartesian topologies, through which it is possible to build several structures that are widely used, such as mesh, ring, and toroid.

The `mpi4py` 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 understand how all the processes are disposed of:

1. Import all the relevant libraries:

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

2. Define the following parameter in order to move along the topology:

```
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3
```

3. For each process, the following array defines the neighbor processes:

```
neighbour_processes = [0,0,0,0]
```

4. In the `main` program, the `comm.rank` and `size` parameters are then defined:

```
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    rank = comm.rank
    size = comm.size
```

5. Now, let's build the topology:

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

6. The following conditions ensure that the processes are always within the topology:

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

7. The `rank` equal to `0` process starts the topology construction:

```
    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=(False, False), \
                                  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\n \
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]))
```

How it works...

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

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

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

Then, the Cartesian topology is built using the `comm.Create_cart` function (note also the parameter, `periods = (False,False)`):

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

To know the position of the process, we use the `Get_coords()` method in the following form:

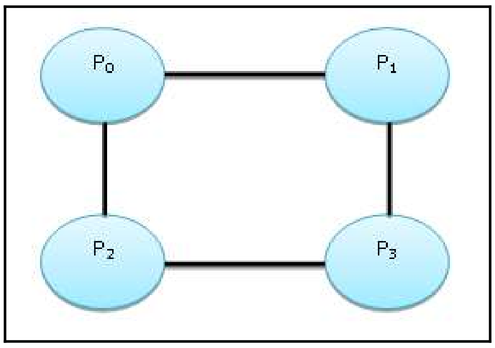
```
my_mpi_row, my_mpi_col =\
cartesian_communicator.Get_coords(cartesian_communicator.rank )
```

For the processes, in addition to getting their coordinates, we must calculate and find out which processes are topologically closer. For this purpose, we use the `comm.Shift (rank_source,rank_dest)` function:

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

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

The topology obtained is as follows:



The virtual mesh 2x2 topology

As the diagram shows, the *P0* process is chained to the **P1** `(RIGHT)` and **P2** `(DOWN)` processes. The **P1** process is chained to the **P3** `(DOWN)` and **P0** `(LEFT)` processes, the **P3** process is chained to the **P1** `(UP)` and **P2** `(LEFT)` processes, and the **P2** process is chained to the **P3** `(RIGHT)` and **P0** `(UP)` processes.

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

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

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

There's more...

To obtain a toroidal topology of the size *M*×*N*, let's use `comm.Create_cart` again, but, this time, let's set the `periods` parameter to `periods=(True,True)`:

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

The following output is obtained:

```
C:\>mpiexec -n 4 python virtualTopology.py
Process = 3 row = 1 column = 1
---->
neighbour_processes[UP] = 1
```

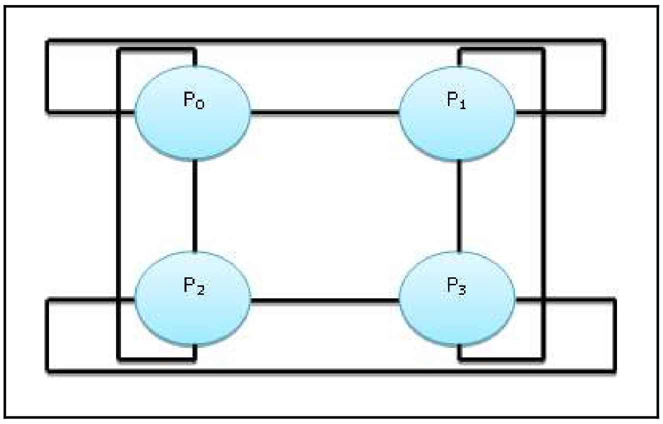
```
neighbour_processes[DOWN] = 1
neighbour_processes[LEFT] =2
neighbour_processes[RIGHT]=2

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

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

The output covers the topology represented here:



The virtual toroidal 2x2 topology

The topology represented in the previous diagram indicates that the **P0** process is chained to the **P1** (`RIGHT` and `LEFT`) and **P2** (`UP` and `DOWN`) processes, the **P1** process is chained to the **P3** (`UP` and `DOWN`) and **P0** (`RIGHT` and `LEFT`) processes, the **P3** process is chained to the **P1** (`UP` and `DOWN`) and **P2** (`RIGHT` and `LEFT`) processes, and the **P2** process is chained to the **P3** (`LEFT` and `RIGHT`) and **P0** (`UP` and `DOWN`) processes.

See also

More information on MPI can be found at `http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-topo.html`.