

# 3

## Process-based Parallelism

In this chapter, we will cover the following recipes:

- ▶ Using the `multiprocessing` Python module
- ▶ How to spawn a process
- ▶ How to name a process
- ▶ How to run a process in the background
- ▶ How to kill a process
- ▶ How to use a process in a subclass
- ▶ How to exchange objects between processes
- ▶ Using a queue to exchange objects
- ▶ Using pipes to exchange objects
- ▶ How to synchronize processes
- ▶ How to manage a state between processes
- ▶ How to use a process pool
- ▶ Using the `mpi4py` Python module
- ▶ Point-to-point communication
- ▶ Avoiding deadlock problems
- ▶ Collective communication using `broadcast`
- ▶ Collective communication using a `scatter` function

- ▶ Collective communication using a gather function
- ▶ Collective communication using `AlltoAll`
- ▶ Reduction operation
- ▶ How to optimize the communication

## Introduction

In the previous chapter, we saw how to use threads to implement concurrent applications. This section will examine the process-based approach. In particular, the focus is on two libraries: the Python `multiprocessing` module and the Python `mpi4py` module.

The Python `multiprocessing` library, which is part of the standard library of the language, implements the shared memory programming paradigm, that is, the programming of a system that consists of one or more processors that have access to a common memory.

The Python library `mpi4py` implements the programming paradigm called message passing. It is expected that there are no shared resources (and this is also called shared nothing) and that all communications take place through the messages that are exchanged between the processes.

For these features, it is in contrast with the techniques of communication that provide memory sharing and the use of lock or similar mechanisms to achieve mutual exclusion. In a message passing code, the processes are connected via the communication primitives of the types `send()` and `receive()`.

In the introduction of the Python `multiprocessing` docs, it is clearly mentioned that all the functionality within this package requires the main module to be importable to the children (<https://docs.python.org/3.3/library/multiprocessing.html>).

The `__main__` module is not importable to the children in IDLE, even if you run the script as a file with IDLE. To get the correct result, we will run all the examples from the Command Prompt:

```
python multiprocessing_example.py
```

Here, `multiprocessing_example.py` is the script's name. For the examples described in this chapter, we will refer to the Python distribution 3.3 (even though Python 2.7 could be used).

## How to spawn a process

The term "spawn" means the creation of a process by a parent process. The parent process can of course continue its execution asynchronously or wait until the child process ends its execution. The multiprocessing library of Python allows the spawning of a process through the following steps:

1. Build the object process.
2. Call its `start()` method. This method starts the process's activity.
3. Call its `join()` method. It waits until the process has completed its work and exited.

### How to do it...

This example shows you how to create a series (five) of processes. Each process is associated with the function `foo(i)`, where `i` is the ID associated with the process that contains it:

```
#Spawn a Process: Chapter 3: Process Based Parallelism
import multiprocessing

def foo(i):
    print ('called function in process: %s' %i)
    return

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=foo, args=(i,))
        Process_jobs.append(p)
        p.start()
        p.join()
```

To run the process and display the results, let's open the Command Prompt, preferably in the folder containing the example file (named `spawn_a_process.py`), and then type the following command:

```
python spawn_a_process.py
```

We obtain the following output using this command:

```
C:\Python CookBook\ Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python spawn_a_process.py
called function in process: 0
called function in process: 1
called function in process: 2
called function in process: 3
called function in process: 4
```

### How it works...

As explained in the introduction section of this recipe, to create the object process, we must first import the multiprocessing module with the following command:

```
import multiprocessing
```

Then, we create the object process in the main program:

```
p = multiprocessing.Process(target=foo, args=(i,))
```

Further, we call the `start()` method:

```
p.start()
```

The object process has for argument the function to which the child process is associated (in our case, the function is called `foo()`). We also pass an argument to the function that takes into account the process in which the associated function is situated. Finally, we call the `join()` method on the process created:

```
p.join()
```

Without `p.join()`, the child process will sit idle and not be terminated, and then, you must manually kill it.

### There's more...

This reminds us once again of the importance of instantiating the `Process` object within the main section:

```
if __name__ == '__main__':
```

This is because the child process created imports the script file where the target function is contained. Then, by instantiating the process object within this block, we prevent an infinite recursive call of such instantiations. A valid workaround is used to define the target function in a different script, and then imports it to the namespace. So for our first example, we could have:

```
import multiprocessing
import target_function

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process \
            (target=target_function.function, args=(i,))
        Process_jobs.append(p)
        p.start()
    p.join()
```

Here, `target_function.py` is as shown:

```
#target_function.py

def function(i):
    print ('called function in process: %s' %i)
    return
```

The output is always similar to that shown in the preceding example.

## How to name a process

In the previous example, we identified the processes and how to pass a variable to the target function. However, it is very useful to associate a name to the processes as debugging an application requires the processes to be well marked and identifiable.

### How to do it...

The procedure to name a process is similar to that described for the threading library (see the recipe *How to determine the current thread* in *Chapter 2, Thread-based Parallelism*, of the present book.)

In the main program, we create a process with a name and a process without a name. Here, the common target is the `foo()` function:

```
#Naming a Process: Chapter 3: Process Based Parallelism
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)

if __name__ == '__main__':
    process_with_name = \
        multiprocessing.Process\
        (name='foo_process',\
         target=foo)
    process_with_name.daemon = True
    process_with_default_name = \
        multiprocessing.Process\
        (target=foo)

    process_with_name.start()
    process_with_default_name.start()
```

To run the process, open the Command Prompt and type the following command:

```
python naming_process.py
```

This is the result that we get after using the preceding command:

```
C:\Python Cookbook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python naming_process.py
Starting foo_process
Starting Process-2
Exiting foo_process
Exiting Process-2
```

## How it works...

The operation is similar to the procedure used for naming a thread. To name a process, we should provide an argument with the object's name:

```
process_with_name = multiprocessing.Process
                    (name='foo_function', target=foo)
```

In this case, we called the `foo_function` process. If the process child wants to know which its parent process is, it must use the following statement:

```
name = multiprocessing.current_process().name
```

This statement will provide the name of the parent process.

## How to run a process in the background

Running a process in background is a typical mode of execution of laborious processes that do not require your presence or intervention, and this course may be concurrent to the execution of other programs. The Python multiprocessing module allows us, through the daemon option, to run background processes.

## How to do it...

To run a background process, simply follow the given code:

```
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)

if __name__ == '__main__':
    background_process = multiprocessing.Process\
        (name='background_process',\
         target=foo)
    background_process.daemon = True
```

```
NO_background_process = multiprocessing.Process\  
                        (name='NO_background_process',\  
                         target=foo)  
  
NO_background_process.daemon = False  
  
background_process.start()  
NO_background_process.start()
```

To run the script from the Command Prompt, type the following command:

```
python background_process.py
```

The final output of this command is as follows:

```
C:\Python CookBook\ Chapter 3 - Process Based Parallelism\Example Codes  
Chapter 3>python background_process.py
```

```
Starting NO_background_process
```

```
Exiting NO_background_process
```

## How it works...

To execute the process in background, we set the `daemon` parameter:

```
background_process.daemon = True
```

The processes in the no-background mode have an output, so the daemon process ends automatically after the main program ends to avoid the persistence of running processes.

## There's more...

Note that a daemon process is not allowed to create child processes. Otherwise, a daemon process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are not Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemon processes have exited.



## How to kill a process

It's possible to kill a process immediately using the `terminate()` method. Also, we use the `is_alive()` method to keep track of whether the process is alive or not.

### How to do it...

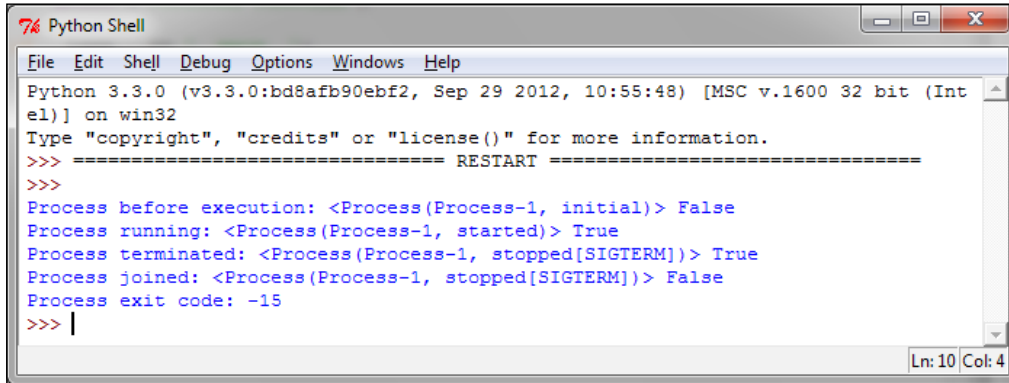
In this example, a process is created with the target function `foo()`. After the start, we kill it with the `terminate()` function:

```
#kill a Process: Chapter 3: Process Based Parallelism
import multiprocessing
import time

def foo():
    print ('Starting function')
    time.sleep(0.1)
    print ('Finished function')

if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print ('Process before execution:', p, p.is_alive())
    p.start()
    print ('Process running:', p, p.is_alive())
    p.terminate()
    print ('Process terminated:', p, p.is_alive())
    p.join()
    print ('Process joined:', p, p.is_alive())
    print ('Process exit code:', p.exitcode)
```

The following is the output we get when we use the preceding command:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Process before execution: <Process(Process-1, initial)> False
Process running: <Process(Process-1, started)> True
Process terminated: <Process(Process-1, stopped[SIGTERM])> True
Process joined: <Process(Process-1, stopped[SIGTERM])> False
Process exit code: -15
>>> |
```

## How it works...

We create the process and then monitor its lifetime by the `is_alive()` method. Then, we finish it with a call to `terminate()`:

`p.terminate()`

Finally, we verify the status code when the process is finished, and read the attribute of the `ExitCode` process. The possible values of `ExitCode` are, as follows:

- ▶ `== 0`: This means that no error was produced
- ▶ `> 0`: This means that the process had an error and exited that code
- ▶ `< 0`: This means that the process was killed with a signal of `-1 * ExitCode`

For our example, the output value of the `ExitCode` code is equal to `-15`. The negative value `-15` indicates that the child was terminated by an interrupt signal identified by the number 15.

## How to use a process in a subclass

To implement a custom subclass and process, we must:

- ▶ Define a new subclass of the `Process` class
- ▶ Override the `__init__(self [,args])` method to add additional arguments
- ▶ Override the `run(self [,args])` method to implement what `Process` should when it is started

Once you have created the new `Process` subclass, you can create an instance of it and then start by invoking the `start()` method, which will in turn call the `run()` method.

## How to do it...

We will rewrite the first example in this manner:

```
#Using a process in a subclass Chapter 3: Process Based #Parallelism

import multiprocessing

class MyProcess(multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' %self.name)
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = MyProcess ()
        jobs.append(p)
        p.start()
    p.join()
```

To run the script from the Command Prompt, type the following command:

```
python subclass_process.py
```

The result of the preceding command is as follows:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python subclass_process.py
```

```
called run method in process: MyProcess-1
called run method in process: MyProcess-2
called run method in process: MyProcess-3
called run method in process: MyProcess-4
called run method in process: MyProcess-5
```

## How it works...

Each Process subclass could be represented by a class that extends the Process class and overrides its run() method. This method is the starting point of Process:

```
class MyProcess (multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' %self.name)
        return
```

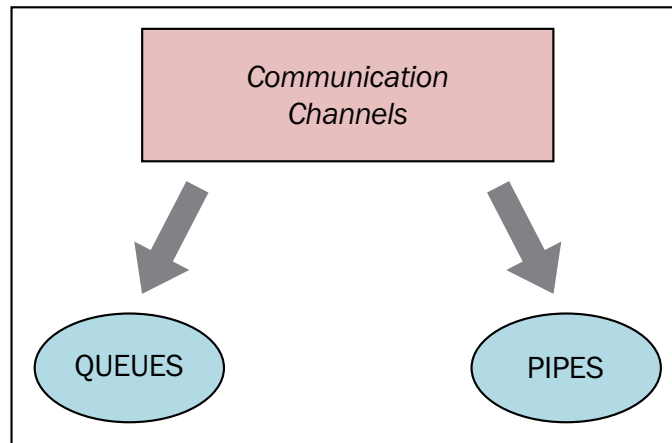
In the main program, we create several objects of the type `MyProcess()`. The execution of the thread begins when the `start()` method is called:

```
p = MyProcess()  
p.start()
```

The `join()` command just handles the termination of processes.

## How to exchange objects between processes

The development of parallel applications has the need for the exchange of data between processes. The multiprocessing library has two communication channels with which it can manage the exchange of objects: queues and pipes.



Communication channels in the multiprocessing module

## Using queue to exchange objects

As explained before, it is possible for us to share data with the queue data structure.

A queue returns a process shared queue, is thread and process safe, and any serializable object (Python serializes an object using the `pickle` module) can be exchanged through it.

## How to do it...

In the following example, we show you how to use a queue for a producer-consumer problem. The `producer` class creates the item and queues and then, the `consumer` class provides the facility to remove the inserted item:

```
import multiprocessing
import random
import time

class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ("Process Producer : item %d appended to queue %s" \
                    % (item, self.name))
            time.sleep(1)
            print ("The size of queue is %s" \
                    % self.queue.qsize())

class consumer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            if (self.queue.empty()):
                print("the queue is empty")
                break
            else :
                time.sleep(2)
                item = self.queue.get()
                print ('Process Consumer : item %d popped from by %s \n' \
                        % (item, self.name))
                time.sleep(1)
```

```
if __name__ == '__main__':
    queue = multiprocessing.Queue()
    process_producer = producer(queue)
    process_consumer = consumer(queue)
    process_producer.start()
    process_consumer.start()
    process_producer.join()
    process_consumer.join()
```

This is the output that we get after the execution:

C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes  
Chapter 3>python using\_queue.py

```
Process Producer : item 69 appended to queue producer-1
The size of queue is 1
Process Producer : item 168 appended to queue producer-1
The size of queue is 2
Process Consumer : item 69 popped from by consumer-2
Process Producer : item 235 appended to queue producer-1
The size of queue is 2
Process Producer : item 152 appended to queue producer-1
The size of queue is 3
Process Producer : item 213 appended to queue producer-1
Process Consumer : item 168 popped from by consumer-2
The size of queue is 3
Process Producer : item 35 appended to queue producer-1
The size of queue is 4
Process Producer : item 218 appended to queue producer-1
The size of queue is 5
Process Producer : item 175 appended to queue producer-1
Process Consumer : item 235 popped from by consumer-2
The size of queue is 5
Process Producer : item 140 appended to queue producer-1
The size of queue is 6
Process Producer : item 241 appended to queue producer-1
The size of queue is 7
Process Consumer : item 152 popped from by consumer-2
Process Consumer : item 213 popped from by consumer-2
```

```

Process Consumer : item 35 popped from by consumer-2
Process Consumer : item 218 popped from by consumer-2
Process Consumer : item 175 popped from by consumer-2
Process Consumer : item 140 popped from by consumer-2
Process Consumer : item 241 popped from by consumer-2
the queue is empty

```

## How it works...

The multiprocessing class has its `Queue` object instantiated in the main program:

```

if __name__ == '__main__':
    queue = multiprocessing.Queue()

```

Then, we create the two processes, `producer` and `consumer`, with the `Queue` object as an attribute:

```

process_producer = producer(queue)
process_consumer = consumer(queue)

```

The process `producer` is responsible for entering 10 items in the queue using its `put()` method:

```

for i in range(10):
    item = random.randint(0, 256)
    self.queue.put(item)

```

The process `consumer` has the task of removing the items from the queue (using the `get` method) and verifying that the queue is not empty. If this happens, the flow inside the `while` loop ends with a `break` statement:

```

def run(self):
    while True:
        if (self.queue.empty()):
            print("the queue is empty")
            break
        else :
            time.sleep(2)
            item = self.queue.get()
            print ('Process Consumer : item %d popped from by %s'
                  % (item, self.name))
            time.sleep(1)

```

## There's more...

A queue has the `JoinableQueue` subclass. It has the following two additional methods:

- ▶ `task_done()`: This indicates that a task is complete, for example, after the `get()` method is used to fetch items from the queue. So, it must be used only by queue consumers.
- ▶ `join()`: This blocks the processes until all the items in the queue have been achieved and processed.

## Using pipes to exchange objects

The second communication channel is the pipe data structure.

A pipe does the following:

- ▶ Returns a pair of connection objects connected by a pipe
- ▶ In this, every object has send/receive methods to communicate between processes

## How to do it...

Here is a simple example with pipes. We have one process pipe the gives out numbers from 0 to 9 and another process that takes the numbers and squares them:

```
import multiprocessing

def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()

def multiply_items(pipe_1, pipe_2):
    close, input_pipe = pipe_1
    close.close()
    output_pipe, _ = pipe_2
    try:
        while True:
            item = input_pipe.recv()
            output_pipe.send(item * item)
    except EOFError:
        output_pipe.close()
```



```

if __name__ == '__main__':

    #First process pipe with numbers from 0 to 9
    pipe_1 = multiprocessing.Pipe(True)
    process_pipe_1 = \
        multiprocessing.Process\
            (target=create_items, args=(pipe_1,))
    process_pipe_1.start()

    #second pipe,
    pipe_2 = multiprocessing.Pipe(True)
    process_pipe_2 = \
        multiprocessing.Process\
            (target=multiply_items, args=(pipe_1, pipe_2,))
    process_pipe_2.start()

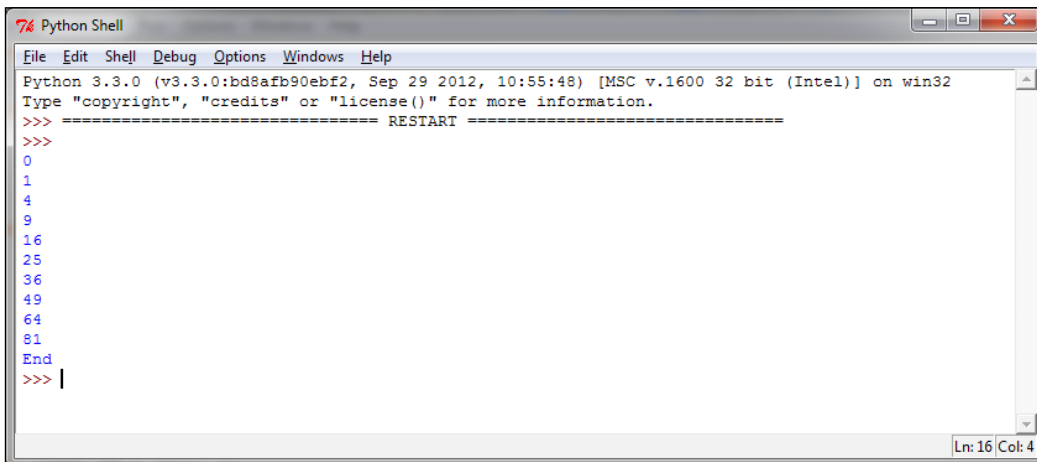
    pipe_1[0].close()
    pipe_2[0].close()

    try:
        while True:

            print (pipe_2[1].recv())
    except EOFError:
        print("End")

```

The output obtained is as follows:



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8a9b90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
0
1
4
9
16
25
36
49
64
81
End
>>> |
Ln: 16 Col: 4

```

## How it works...

Let's remember that the `pipe()` function returns a pair of connection objects connected by a two way pipe. In the example, `out_pipe` contains the numbers from 0 to 9, generated by the target function `create_items()`:

```
def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()
```

In the second process, we have two pipes: the input pipe and final output pipe that contains the results:

```
process_pipe_2 = multiprocessing.Process(target=multiply_items,
                                         args=(pipe_1, pipe_2,))
```

These are finally printed as:

```
try:
    while True:
        print (pipe_2[1].recv())
except EOFError:
    print ("End")
```

## How to synchronize processes

Multiple processes can work together to perform a given task. Usually, they share data. It is important that the access to shared data by various processes does not produce inconsistent data. Processes that cooperate by sharing data must therefore act in an orderly manner in order to access that data. Synchronization primitives are quite similar to those encountered for the library and threading.

They are as follows:

- ▶ **Lock:** This object can be in one of the states: locked and unlocked. A lock object has two methods, `acquire()` and `release()`, to manage the access to a shared resource.
- ▶ **Event:** This realizes simple communication between processes, one process signals an event and the other processes wait for it. An `Event` object has two methods, `set()` and `clear()`, to manage its own internal flag.
- ▶ **Condition:** This object is used to synchronize parts of a workflow, in sequential or parallel processes. It has two basic methods, `wait()` is used to wait for a condition and `notify_all()` is used to communicate the condition that was applied.

- ▶ **Semaphore:** This is used to share a common resource, for example, to support a fixed number of simultaneous connections.
- ▶ **RLock:** This defines the recursive lock object. The methods and functionality for RLock are the same as the Threading module.
- ▶ **Barrier:** This divides a program into phases as it requires all of the processes to reach it before any of them proceeds. Code that is executed after a barrier cannot be concurrent with the code executed before the barrier.

## How to do it...

The example here shows the use of `barrier()` to synchronize two processes. We have four processes, wherein `process1` and `process2` are managed by a barrier statement, while `process3` and `process4` have no synchronizations directives:

```
import multiprocessing
from multiprocessing import Barrier, Lock, Process
from time import time
from datetime import datetime

def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
    now = time()
    with serializer:
        print("process %s ----> %s" \
              %(name, datetime.fromtimestamp(now)))

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print("process %s ----> %s" \
          %(name, datetime.fromtimestamp(now)))

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p3 - test_without_barrier'\
```

```
,target=test_without_barrier).start()
Process(name='p4 - test_without_barrier'\
, target=test_without_barrier).start()
```

By running the script, we can see that process1 and process2 print out the same timestamps:

C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes  
Chapter 3>python process\_barrier.py

```
process p1 - test_with_barrier ----> 2015-05-09 11:11:33.291229
process p2 - test_with_barrier ----> 2015-05-09 11:11:33.291229
process p3 - test_without_barrier ----> 2015-05-09 11:11:33.310230
process p4 - test_without_barrier ----> 2015-05-09 11:11:33.333231
```

### How it works...

In the main program, we created four processes; however, we also need a barrier and lock primitive. The parameter 2 in the barrier statement stands for the total number of process that are to be managed:

```
if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier'\
, target=test_with_barrier,\
    args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier'\
, target=test_with_barrier,\
    args=(synchronizer,serializer)).start()
```

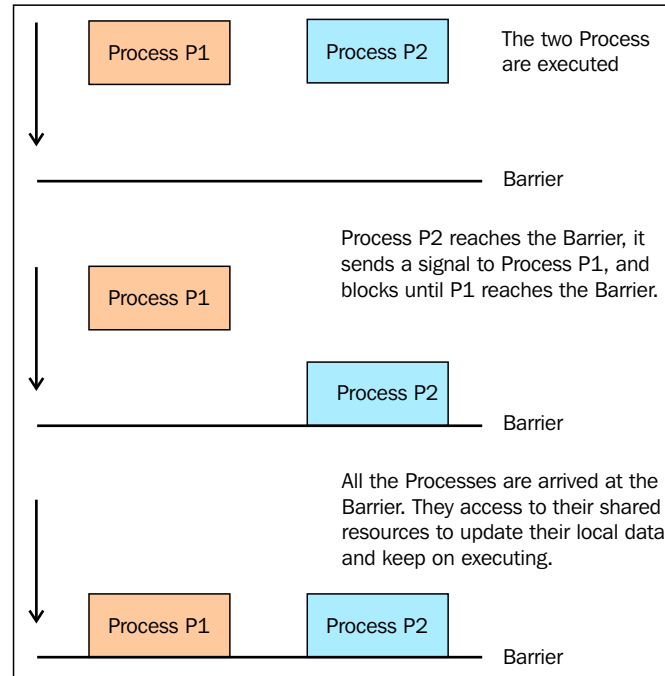
The test\_with\_barrier\_function executes the barrier's wait() method:

```
def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
```

When the two processes have called the wait() method, they are released simultaneously:

```
now = time()
with serializer:
    print("process %s ----> %s" %(name \
,datetime.fromtimestamp(now)))
```

The following figure shows you how a barrier works with the two processes:



Process management with a barrier

## How to manage a state between processes

Python multiprocessing provides a manager to coordinate shared information between all its users. A manager object controls a server process that holds Python objects and allows other processes to manipulate them.

A manager has the following properties:

- ▶ It controls the server process that manages a shared object
- ▶ It makes sure the shared object gets updated in all processes when anyone modifies it

## How to do it...

Let's see an example of how to share a state between processes:

1. First, the program creates a manager list, shares it between  $n$  number of taskWorkers, and every worker updates an index.
2. After all workers finish, the new list is printed to `stdout`:

```
import multiprocessing

def worker(dictionary, key, item):
    dictionary[key] = item

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    dictionary = mgr.dict()
    jobs = [ multiprocessing.Process\
               (target=worker, args=(dictionary, i, i*2))
             for i in range(10)
           ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print ('Results:', dictionary)
```

The output is as follows:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python manager.py
```

```
key = 0 value = 0
key = 2 value = 4
key = 6 value = 12
key = 4 value = 8
key = 8 value = 16
key = 7 value = 14
key = 3 value = 6
key = 1 value = 2
key = 5 value = 10
key = 9 value = 18
```

```
Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9:
18}
```

## How it works...

We declare the manager with the following statement:

```
mgr = multiprocessing.Manager()
```

In the next statement, a data structure of the type dictionary is created:

```
dictionary = mgr.dict()
```

Then, the multiprocessing is launched:

```
jobs = [multiprocessing.Process \
        (target=taskWorker, args=(dictionary, i, i*2))
        for i in range(10)]

for j in jobs:
    j.start()
```

Here, the target function `taskWorker` adds an item to the data structure dictionary:

```
def taskWorker(dictionary, key, item):
    dictionary[key] = value
```

Finally, we get the output and all the dictionaries are printed out:

```
for j in jobs:
    j.join()
    print ('Results:', d)
```

## How to use a process pool

The multiprocessing library provides the `Pool` class for simple parallel processing tasks. The `Pool` class has the following methods:

- ▶ `apply()`: It blocks until the result is ready.
- ▶ `apply_async()`: This is a variant of the `apply()` method, which returns a result object. It is an asynchronous operation that will not lock the main thread until all the child classes are executed.
- ▶ `map()`: This is the parallel equivalent of the `map()` built-in function. It blocks until the result is ready, this method chops the iterable data in a number of chunks that submits to the process pool as separate tasks.

- ▶ `map_async()`: This is a variant of the `map()` method, which returns a result object. If a callback is specified, then it should be callable, which accepts a single argument. When the result becomes ready, a callback is applied to it (unless the call failed). A callback should be completed immediately; otherwise, the thread that handles the results will get blocked.

## How to do it...

This example shows you how to implement a process pool to perform a parallel application. We create a pool of four processes and then we use the pool's `map` method to perform a simple calculation:

```
def function_square(data):
    result = data*data
    return result

if __name__ == '__main__':
    inputs = list(range(100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)
    pool.close()
    pool.join()
    print ('Pool      :', pool_outputs)
```

This is the result that we get after completing the calculation:

C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes  
Chapter 3>\python process\_pool.py

```
Pool      : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784,
841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704,
2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096,
4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776,
5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744,
7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
```



### How it works...

The `multiprocessing.Pool` method applies `function_square` to the input element to perform a simple calculation. The total number of parallel processes is four:

```
pool = multiprocessing.Pool(processes=4)
```

The `pool.map` method submits to the process pool as separate tasks

```
pool_outputs = pool.map(function_square, inputs)
```

The parameter `inputs` is a list of integer from 0 to 100:

```
inputs = list(range(100))
```

The result of the calculation is stored in `pool_outputs`. Then, the final result is printed:

```
print ('Pool      :', pool_outputs)
```

It is important to note that the result of the `pool.map()` method is equivalent to Python's built-in function `map()`, except that the processes run parallelly.

## Using the mpi4py Python module

The Python programming language provides a number of MPI modules to write parallel programs. The most interesting of these is the `mpi4py` library. It is constructed on top of the MPI-1/2 specifications and provides an object-oriented interface, which closely follows MPI-2 C++ bindings. A C MPI user could use this module without learning a new interface. Therefore, it is widely used as an almost full package of an MPI library in Python.

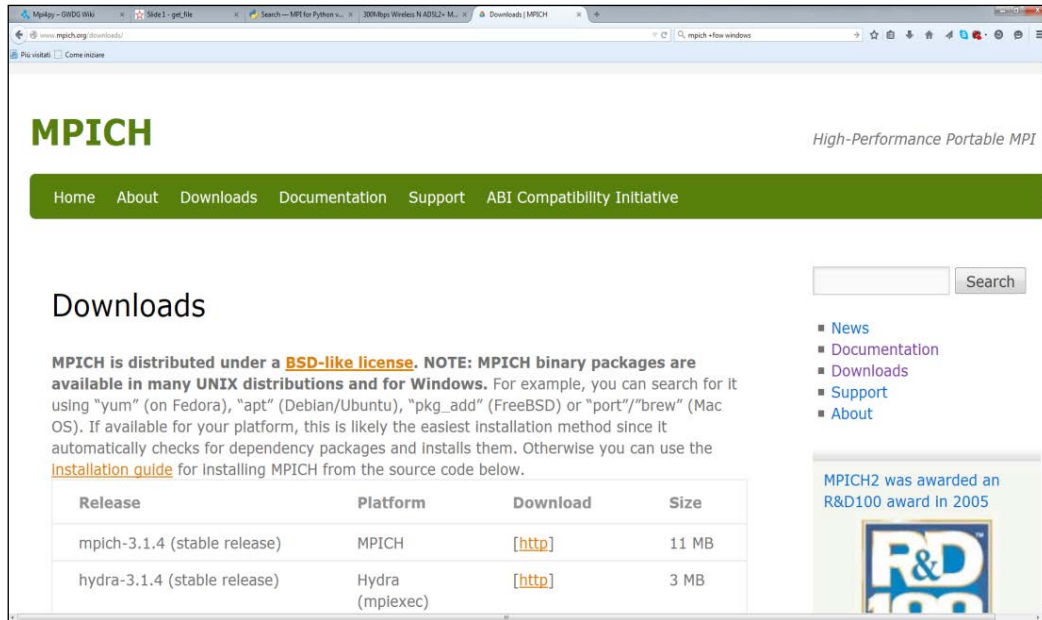
The main applications of the module, which will be described in this chapter, are:

- ▶ Point-to-point communication
- ▶ Collective communication
- ▶ Topologies

## Getting ready

The installation procedure of `mpi4py` using a Windows machine is, as follows (for other OS, refer to <http://mpi4py.scipy.org/docs/usrman/install.html#>):

1. Download the MPI software library `mpich` from <http://www.mpich.org/downloads/>.

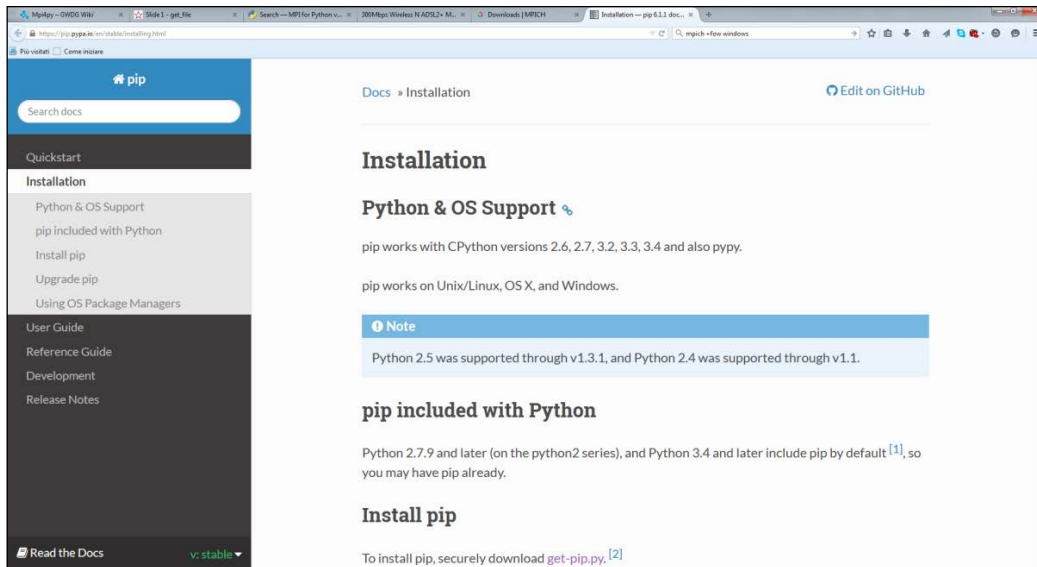


The MPICH download page

2. Open an admin Command Prompt by right-clicking on the command prompt icon and select **Run as administrator**.
3. Run `msiexec /i mpich_installation_file.msi` from the admin Command Prompt to install MPICH2.
4. During the installation, select the option that installs MPICH2 for all users.
5. Run `wmpiconfig` and store the username/password. Use your *real* Windows login name and password.
6. Add `C:\Program Files\MPICH2\bin` to the system path—no need to reboot the machine.
7. Check `smpd` using `smpd -status`. It should return `smpd running on $hostname$`.
8. To test the execution environment, go to the `$MPICHROOT\examples` directory and run `cpi.exe` using `mpiexec -n 4 cpi`.

- Download the Python installer `pip` from <https://pip.pypa.io/en/stable/installing.html>.

It will create a `pip.exe` file in the `Scripts` directory of your Python distribution.



The PIP download page

- Then, from the Command Prompt, type the following to install `mpi4py`:

```
C:> pip install mpi4py
```

## How to do it...

Let's start our journey to the MPI library by examining the classic code or a program that prints the phrase "Hello, world!" on each process that is instantiated:

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print ("hello world from process ", rank)
```

To execute the code, type the following command line:

```
C:> mpiexec -n 5 python helloWorld_MPI.py
```

This is the result that we would get after we execute this code:

```
('hello world from process ', 1)
('hello world from process ', 0)
('hello world from process ', 2)
('hello world from process ', 3)
('hello world from process ', 4)
```

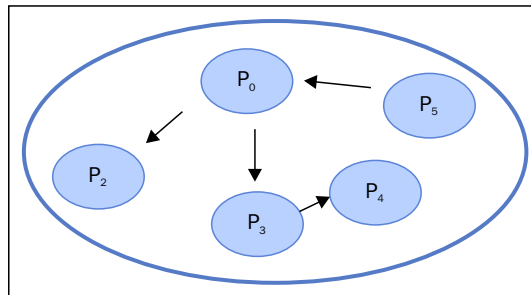
### How it works...

In MPI, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers called ranks. If we have a number  $p$  of processes that runs a program, the processes will then have a rank that goes from 0 to  $p-1$ . The function MPI that comes to us to solve this problem has the following function calls:

```
rank = comm.Get_rank()
```

This function returns the rank of the process that called it. The `comm` argument is called a communicator, as it defines its own set of all processes that can communicate together, namely:

```
comm = MPI.COMM_WORLD
```



An example of communication between processes in MPI.COMM\_WORLD

### There's more...

It should be noted that, for illustration purposes only, the `stdout` output will not always be ordered, as multiple processes can apply at the same time by writing on the screen and the operating system arbitrarily chooses the order. So, we are ready for a fundamental observation: every process involved in the execution of MPI runs the same compiled binary, so each process receives the same instructions to be executed.

## Point-to-point communication

One of the most important features among those provided by MPI is the point-to-point communication, which is a mechanism that enables data transmission between two processes: a process receiver, and process sender.

The Python module `mpi4py` enables point-to-point communication via two functions:

- ▶ `Comm.Send(data, process_destination)`: This sends data to the destination process identified by its rank in the communicator group
- ▶ `Comm.Recv(process_source)`: This receives data from the source process, which is also identified by its rank in the communicator group

The `Comm` parameter, which stands for communicator, defines the group of processes, that may communicate through message passing:

```
comm = MPI.COMM_WORLD
```

### How to do it...

In the following example, we show you how to utilize the `comm.send` and `comm.recv` directives to exchange messages between different processes:

```
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
    print ("sending data %s " %data + \
          "to process %d" %destination_process)

if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
    print ("sending data %s :" %data + \
          "to process %d" %destination_process)
```

```
if rank==4:
    data=comm.recv(source=0)
    print ("data received is = %s" %data)

if rank==8:
    data1=comm.recv(source=1)
    print ("data1 received is = %s" %data1)
```

To run the script, type the following:

```
C:\>mpiexec -n 9 python pointToPointCommunication.py
```

This is the output that you'll get after you run the script:

```
('my rank is : ', 5)
('my rank is : ', 1)
sending data hello :to process 8
('my rank is : ', 3)
('my rank is : ', 0)
sending data 10000000 to process 4
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 4)
data received is = 10000000
('my rank is : ', 8)
data1 received is = hello
('my rank is : ', 6)
```

## How it works...

We ran the example with a total number of processes equal to nine. So in the communicator group, `comm`, we have nine tasks that can communicate with each other:

```
comm=MPI.COMM_WORLD
```

Also, to identify a task or processes inside the group, we use their `rank` value:

```
rank = comm.rank
```

We have two sender processes and two receiver processes.

The process of a rank equal to zero sends numerical data to the receiver process of a rank equal to four:

```
if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
```

Similarly, we must specify the receiver process of rank equal to four. Also, we note that the `comm.recv` statement must contain as an argument, the rank of the sender process:

```
...
if rank==4:
    data=comm.recv(source=0)
```

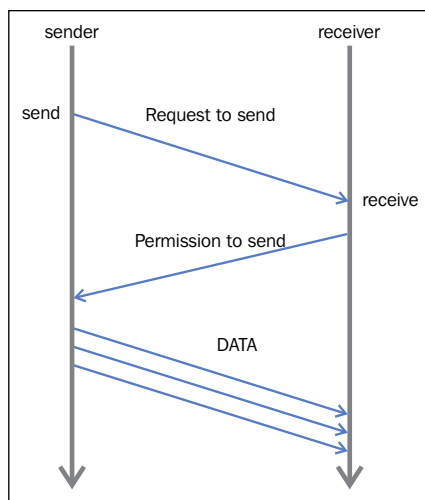
For the other sender and receiver processes, the process of a rank equal to one and the process of a rank equal to eight, respectively, the situation is the same but the only difference is the type of data. In this case, for the sender process, we have a string that is to be sent:

```
if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
```

For the receiver process of a rank equal to eight, the rank of the sender process is pointed out:

```
if rank==8:
    data1=comm.recv(source=1)
```

The following figure summarizes the point-to-point communication protocol in `mpi4py`:



The send/receive transmission protocol

It is a two-step process, consisting of sending some data from one task (**sender**) and of receiving these data by another task (**receiver**). The sending task must specify the data to be sent and their destination (the receiver process), while the receiving task has to specify the source of the message to be received.

### There's more...

The `comm.send()` and `comm.recv()` functions are *blocking* functions; they block the caller until the buffered data involved can safely be used. Also in MPI, there are two management methods of sending and receiving messages:

- ▶ The buffered mode
- ▶ The synchronous mode

In the buffered mode, the flow control returns to the program as soon as the data to be sent has been copied to a buffer. This does not mean that the message is sent or received. In the synchronous mode, however, the function only gets terminated when the corresponding receive function begins receiving the message.

## Avoiding deadlock problems

A common problem we face is that of the deadlock. This is a situation where two (or more) processes block each other and wait for the other to perform a certain action that serves to another, and vice versa. The `mpi4py` module doesn't provide any specific functionality to resolve this but only some measures, which the developer must follow to avoid problems of deadlock.

### How to do it...

Let's first analyze the following Python code, which will introduce a typical deadlock problem; we have two processes, `rank` equal to one and `rank` equal to five, that communicate with each other and both have the data sender and data receiver functionality:

```
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)
```



---

```

if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5

    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)

    print ("sending data %s " %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1

    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

    print ("sending data %s :" %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)

```

### How it works...

If we try to run this program (it makes sense to execute it with only two processes), we note that none of the two processes are able to proceed:

```
C:\>mpirun -n 9 python deadLockProblems.py
```

```

('my rank is : ', 8)
('my rank is : ', 3)
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 4)
('my rank is : ', 6)

```

Both prepare to receive a message from the other and get stuck there. This happens because the function `MPI comm.recv()` as well as the `comm.send()` MPI blocks them. It means that the calling process waits for their completion. As for the `comm.send()` MPI, the completion occurs when the data has been sent and may be overwritten without modifying the message. The completion of the `comm.recv()` MPI, instead, is when the data has been received and can be used. To solve the problem, the first idea that occurs is to invert the `comm.recv()` MPI with the `comm.send()` MPI in this way:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)
```

This solution, however, even if correct from the logical point of view, not always ensures the avoidance of a deadlock. Since the communication is carried out through a buffer, where the `comm.send()` MPI copies the data to be sent, the program runs smoothly only if this buffer is able to hold them all. Otherwise, there is a deadlock: the sender cannot finish sending data because the buffer is committed and the receiver cannot receive data as it is blocked by a `comm.send()` MPI, which is not yet complete. At this point, the solution that allows us to avoid deadlocks is used to swap the sending and receiving functions so as to make them asymmetrical:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)
```

Finally, we get the correct output:

```
C:\>mpiexec -n 9 python deadLockProblems.py
```

```
('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 8)
('my rank is : ', 1)
sending data a to process 5
data received is = b
('my rank is : ', 5)
sending data b :to process 1
data received is = a
('my rank is : ', 2)
('my rank is : ', 3)
('my rank is : ', 4)
('my rank is : ', 6)
```

### There's more...

The solution to the deadlock is not the only solution. There is, for example, a particular function that unifies the single call that sends a message to a given process and receives another message that comes from another process. This function is called `Sendrecv`:

```
Sendrecv(self, sendbuf, int dest=0, int sendtag=0, recvbuf=None, int
source=0, int recvtag=0, Status status=None)
```

As you can see, the required parameters are the same as the `comm.send()` MPI and the `comm.recv()` MPI. Also, in this case, the function blocks, but compared to the two already seen previously it offers the advantage of leaving the communication subsystem responsible for checking the dependencies between sending and receiving, thus avoiding the deadlock. In this way the code of the previous example becomes as shown:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.sendrecv(data_send,dest=destination_process,
                                source =source_process)

if rank==5:
    data_send= "b"
```

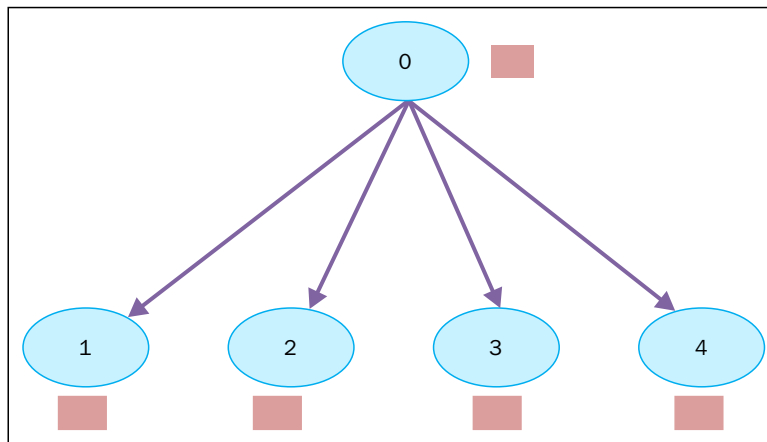
```
destination_process = 1
source_process = 1
data_received=comm.sendrecv(data_send,dest=destination_process,
                             source=source_process)
```

## Collective communication using broadcast

During the development of a parallel code, we often find ourselves in the situation where we have to share between multiple processes the value of a certain variable at runtime or certain operations on variables that each process provides (presumably with different values).

To resolve this type of situations, the communication trees are used (for example the process 0 sends data to the processes 1 and 2, which respectively will take care of sending them to the processes 3, 4, 5, and 6, and so on).

Instead, MPI libraries provide functions ideal for the exchange of information or the use of multiple processes that are clearly optimized for the machine in which they are performed.



Broadcasting data from process 0 to processes 1, 2, 3, and 4

A communication method that involves all the processes belonging to a communicator is called a collective communication. Consequently, a collective communication generally involves more than two processes. However, instead of this, we will call the collective communication broadcast, wherein a single process sends the same data to any other process. The `mpi4py` functionalities in the broadcast are offered by the following method:

```
buf = comm.bcast(data_to_share, rank_of_root_process)
```

This function simply sends the information contained in the message process root to every other process that belongs to the `comm` communicator; each process must, however, call it by the same values of `root` and `comm`.

## How to do it...

Let's now see an example wherein we've used the broadcast function. We have a root process of rank equal to zero that shares its own data, `variable_to_share`, with the other processes defined in the communicator group:

```
from mpi4py import MPI

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

if rank == 0:
    variable_to_share = 100

else:
    variable_to_share = None

variable_to_share = comm.bcast(variable_to_share, root=0)
print("process = %d" %rank + " variable shared = %d " \
      %variable_to_share)
```

The output obtained with a communicator group of ten processes is:

```
C:\>mpiexec -n 10 python broadcast.py
process = 0 variable shared = 100
process = 8 variable shared = 100
process = 2 variable shared = 100
process = 3 variable shared = 100
process = 4 variable shared = 100
process = 5 variable shared = 100
process = 9 variable shared = 100
process = 6 variable shared = 100
process = 1 variable shared = 100
process = 7 variable shared = 100
```

## How it works...

The process root of rank zero instantiates a variable, `variable_to_share`, equal to 100. This variable will be shared with the other processes of the communication group:

```
if rank == 0:
    variable_to_share = 100
```

To perform this, we also introduce the broadcasting communication statement:

```
variable_to_share = comm.bcast(variable_to_share, root=0)
```

Here, the parameters in the function are the data to be shared and the root process or main sender process, as denoted in the previous figure. When we run the code, in our case, we have a communication group of ten processes, `variable_to_share` is shared between the others processes in the group. Finally, the `print` statement visualizes the rank of the running process and the value of its variable:

```
print("process = %d" %rank + " variable shared = %d " \
      %variable_to_share)
```

### There's more...

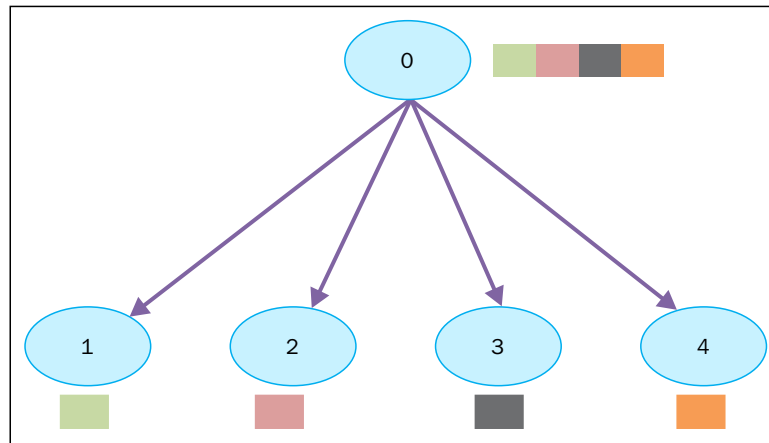
Collective communication allows simultaneous data transmission between multiple processes in a group. In `mpi4py` the collective communication are provided only in their blocking version (they block the caller method until the buffered data involved can safely be used.)

The most commonly collective operations are:

- ▶ Barrier synchronization across the group's processes
- ▶ Communication functions:
  - Broadcasting data from one process to all process in the group
  - Gathering data from all process to one process
  - Scattering data from one process to all process
- ▶ Reduction operation

## Collective communication using scatter

The scatter functionality is very similar to a scatter broadcast but has one major difference, while `comm.bcast` sends the same data to all listening processes, `comm.scatter` can send the chunks of data in an array to different processes. The following figure illustrates the functionality of scatter:



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\appdata\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]
```

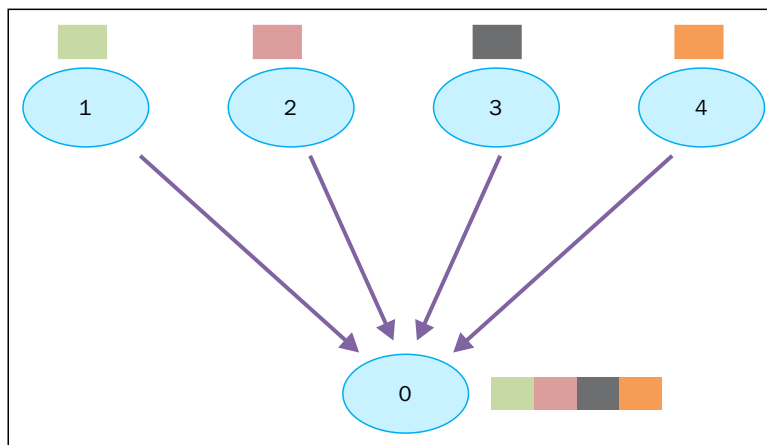
Here, `data` must be a buffer-like object of the size `data_size` and of the type `data_type`.

## 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:\>mpirun -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))
```

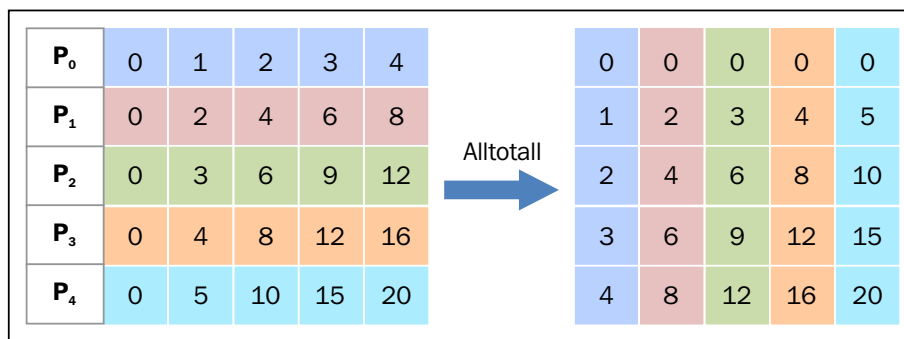
We run the code with a communicator group of five processes and the output we get is as follows:

```
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 **P<sub>0</sub>** contains the data array [0 1 2 3 4], where it assigns **0** to itself, **1** to the process **P<sub>1</sub>**, **2** to the process **P<sub>2</sub>**, **3** to the process **P<sub>3</sub>**, and **4** to the process **P<sub>4</sub>**.
- ▶ The process **P<sub>1</sub>** contains the data array [0 2 4 6 8], where it assigns **0** to **P<sub>0</sub>**, **2** to itself, **4** to the process **P<sub>2</sub>**, **6** to the process **P<sub>3</sub>**, and **8** to the process **P<sub>4</sub>**.
- ▶ The process **P<sub>2</sub>** contains the data array [0 3 6 9 12], where it assigns **0** to **P<sub>0</sub>**, **3** to the process **P<sub>1</sub>**, **6** to itself, **9** to the process **P<sub>3</sub>**, and **12** to the process **P<sub>4</sub>**.
- ▶ The process **P<sub>3</sub>** contains the data array [0 4 8 12 16], where it assigns **0** to **P<sub>0</sub>**, **4** to the process **P<sub>1</sub>**, **8** to the process **P<sub>2</sub>**, **12** to itself, and **16** to the process **P<sub>4</sub>**.
- ▶ The process **P<sub>4</sub>** contains the data array [0 5 10 15 20], where it assigns **0** to **P<sub>0</sub>**, **5** to the process **P<sub>1</sub>**, **10** to the process **P<sub>2</sub>**, **15** to the process **P<sub>3</sub>**, 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]

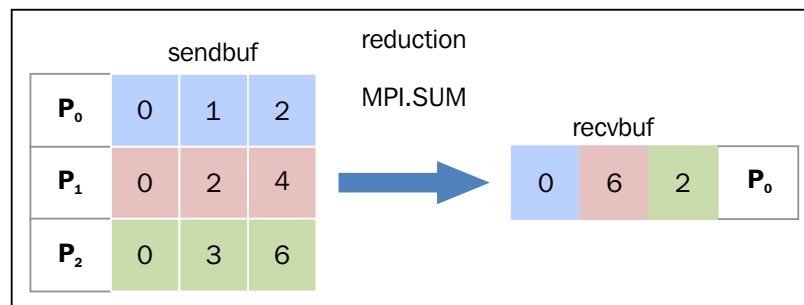
```

### 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  $M \times N$ . 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  $2 \times 2$  (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:

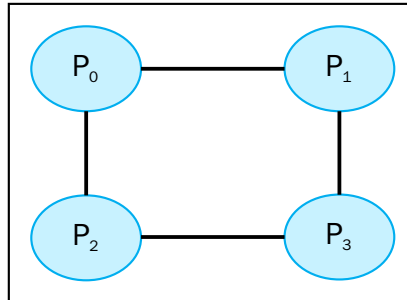
```
my_mpi_row, my_mpi_col = cartesian_communicator.Get_coords( cartesian_
communicator.rank )
For each process, in addition to their coordinates, we calculated
and got to know which processes are topologically closer. For
this purpose, we used the comm.Shift function comm.Shift (rank_
source, rank_dest)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

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...

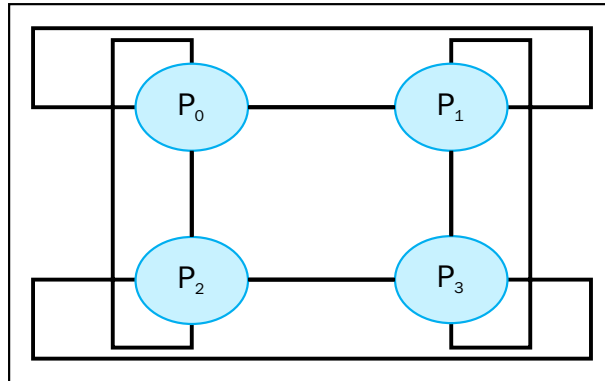
To obtain a toroidal topology of the size  $M \times N$ , we need the following lines of code:

```
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

