

Primena super-računara u astronomiji



PYTHON



MPI paralelization in Python

- 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()
```

- 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
```

Point-to-point communication

- The point-to-point communication 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)`
 - `Comm.Recv(process_source)`

Point-to-point communication

```
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)
```

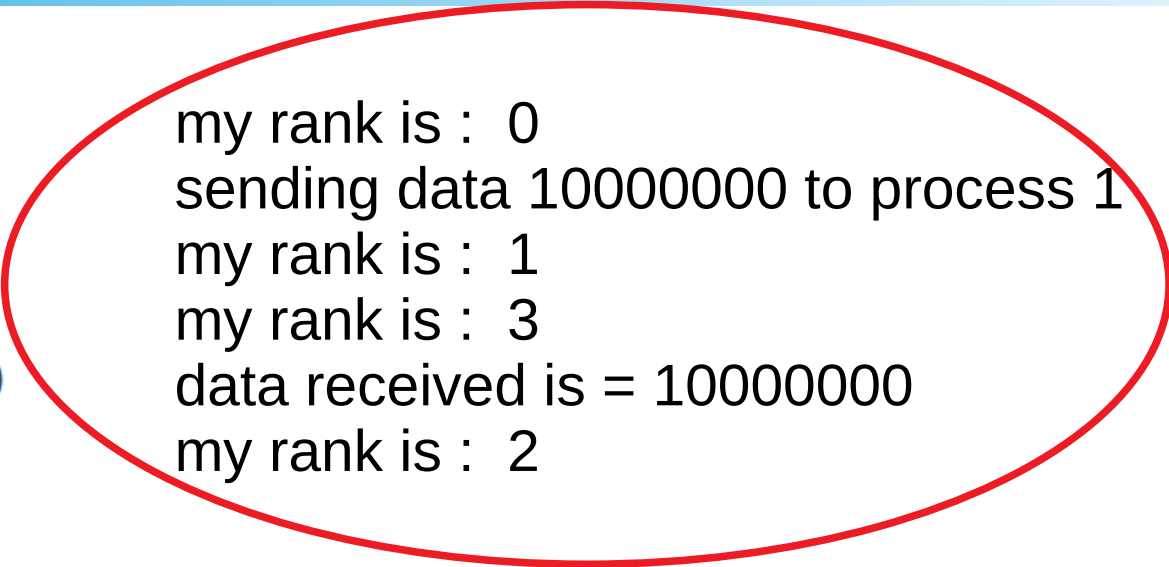
Point-to-point communication

```
from mpi4py import MPI

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

```
if rank==0:
    data= 10000000
    destination_process = 1
    comm.send(data,dest=destination_process)
    print ("sending data %s " %data + \
           "to process %d" %destination_process)
```

```
if rank==1:
    data=comm.recv(source=0)
    print ("data received is = %s" %data)
```



my rank is : 0
sending data 10000000 to process 1
my rank is : 1
my rank is : 3
data received is = 10000000
my rank is : 2

Avoiding deadlock problems

- The deadlock 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

Avoiding deadlock problems

```
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
```

```
    comm.send(data_send,dest=desti
```

```
    print ("sending data %s " %data
```

```
            "to process %d" %desti
```

```
    print ("data received is = %s"
```

```
if rank==5:
```

```
    data_send= "b"
```

```
    destination_process = 1
```

```
    source_process = 1
```

```
    comm.send(data_send,dest=desti
```

```
    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)
```

```
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)
```

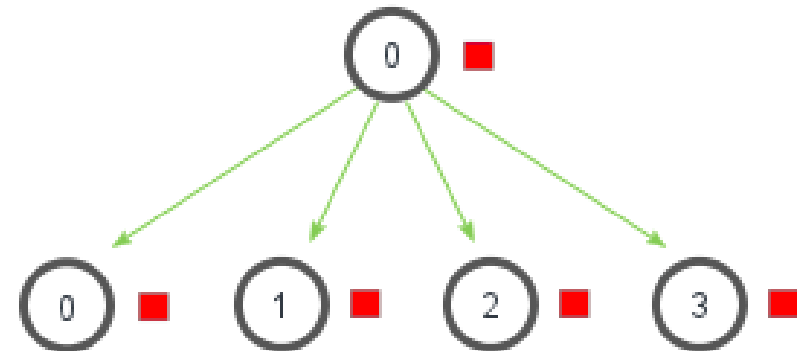
```
    data_received=comm.recv(source=source_process)
```


Collective communication using broadcast

- A communication method that involves all the processes belonging to a communicator is called a **collective communication**
- A collective communication generally involves more than two processes.
- Here 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**



Collective communication using broadcast

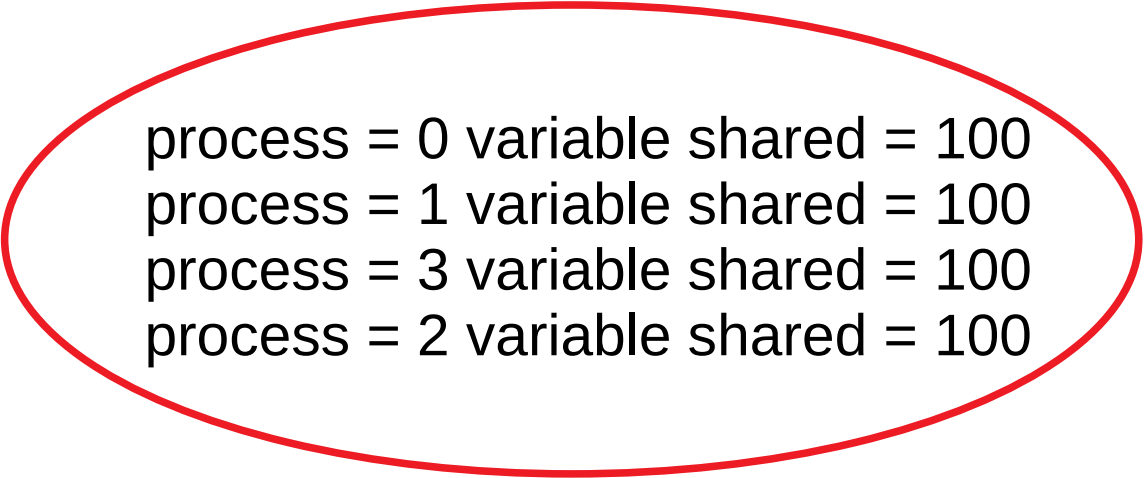
```
from mpi4py import MPI
```

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

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

```
else:  
    variable_to_share = None
```

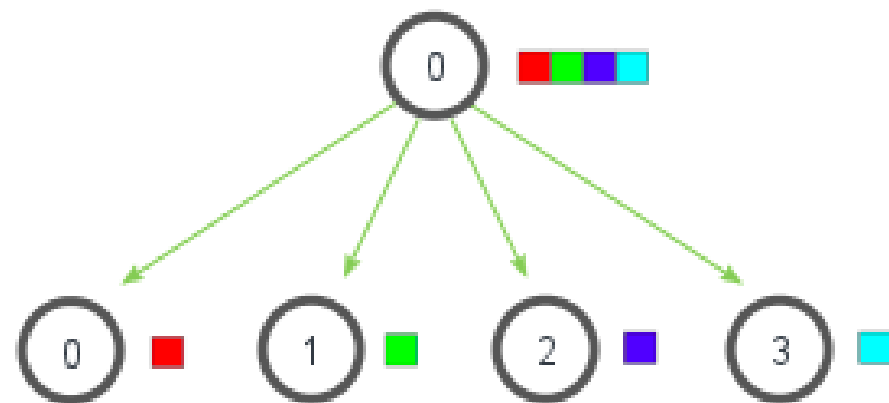
```
variable_to_share = comm.bcast(variable_to_share, root=0)  
print("process = %d" %rank + " variable shared = %d " \  
      %variable_to_share)
```



process = 0 variable shared = 100
process = 1 variable shared = 100
process = 3 variable shared = 100
process = 2 variable shared = 100

Collective communication using scatter

- The scatter functionality is very similar to a broadcast
- There is 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 `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)`



Collective communication using scatter

```
from mpi4py import MPI
```

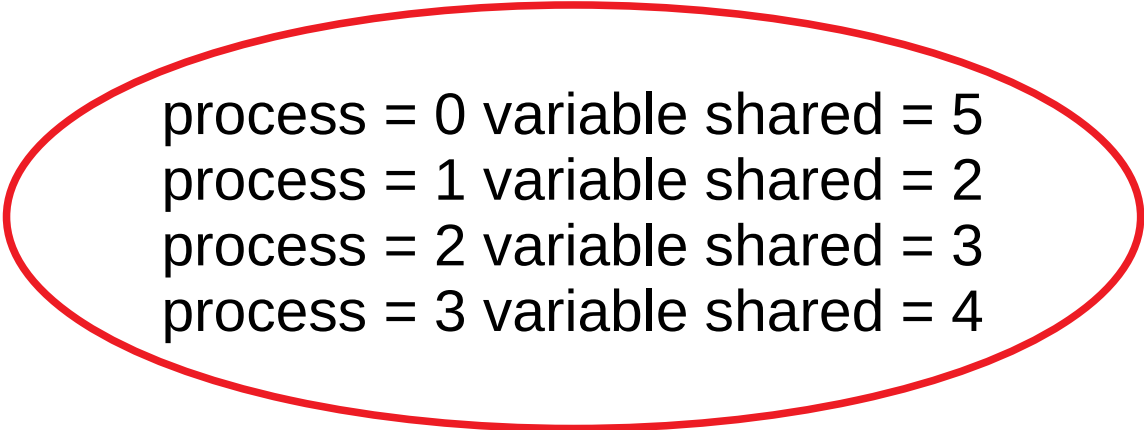
```
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()
```

```
if rank == 0:  
    array_to_share = [5, 2, 3, 4]
```

```
else:  
    array_to_share = None
```

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

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

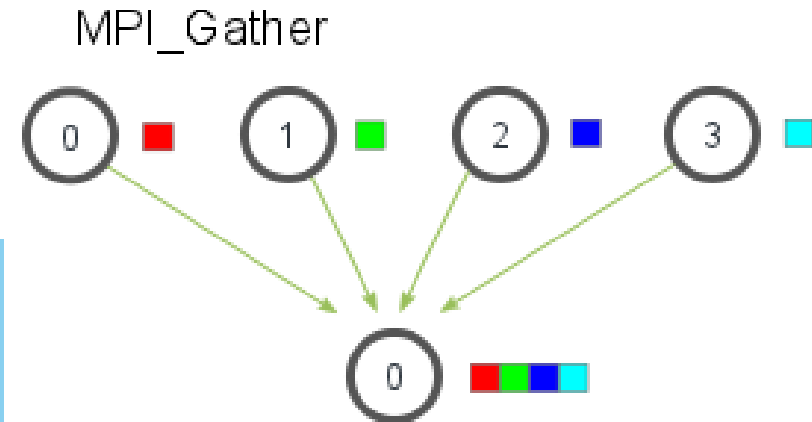


process = 0 variable shared = 5
process = 1 variable shared = 2
process = 2 variable shared = 3
process = 3 variable shared = 4

- One of the restrictions to `comm.scatter` is that you can scatter as many elements as the processors you have available or specify in the execution statement !

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)`



Collective communication using gather

```
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))
```

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

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

Collective communication using Alltoall

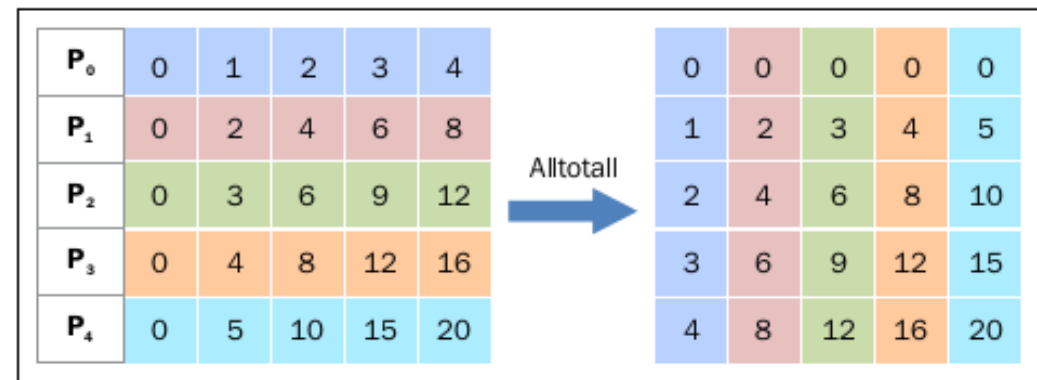
```
from mpi4py import MPI
import numpy
```

```
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))
```

process 0 sending [0 1 2 3] receiving [0 0 0 0]
process 2 sending [0 3 6 9] receiving [2 4 6 8]
process 3 sending [0 4 8 12] receiving [3 6 9 12]
process 1 sending [0 2 4 6] receiving [1 2 3 4]



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
- However, the output elements contain the reduced result
- In **mpi4py**, the reduction operation is defined through the following statement:
 - `comm.Reduce(sendbuf, recvbuf, rank_of_root_process, op = type_of_reduction_operation)`
- 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
- 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

The reduction operation

```
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(array_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)
```

process 0 sending [0 1 2]

process 1 sending [0 2 4]

process 3 sending [0 4 8]

process 2 sending [0 3 6]

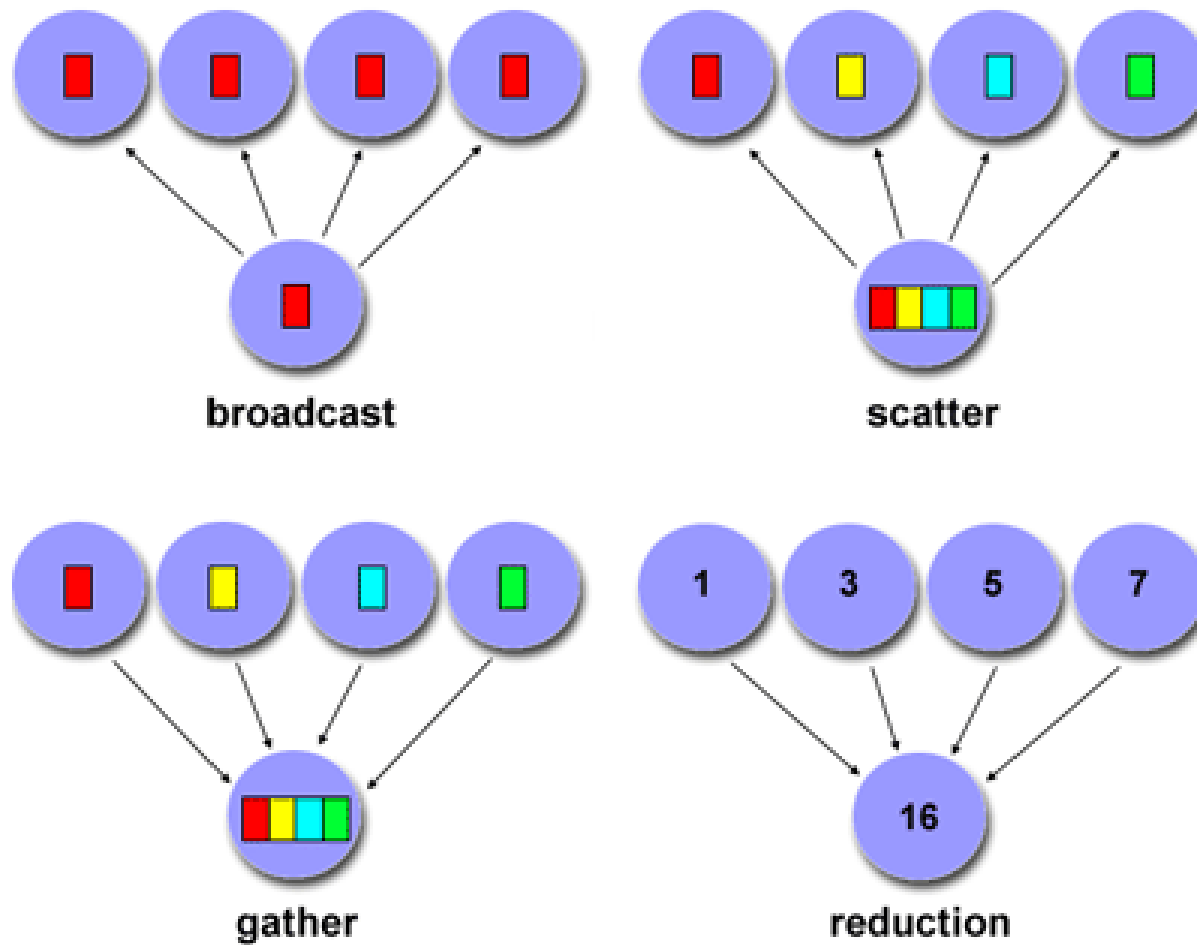
on task 2 after Reduce: data = [0 0 0]

on task 0 after Reduce: data = [0 10 20]

on task 1 after Reduce: data = [0 0 0]

on task 3 after Reduce: data = [0 0 0]

Overview



Comm.barrier()

- Collective communication implies a synchronization point among processes
- MPI has a special function that is dedicated to synchronizing processes: `Comm.Barrier()`
- The function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function

Comm.barrier()

