

MPI - Python

Basic definitions

What is Multithreading?

- A single core(processor) in your computer can mimic multi(N)processor, This will allow you to run multiple(N) programs concurrently(not exactly!).
- Multiple processor can share same memory space. So that a single variable can be updated or used by multiple processor at the same time.

What is Multiprocessing?

- A single core(processor) is always single. So, for running program concurrently(exactly), you need to use multiple physical cores(processors)
- Multiple processor doesn't share memory. So, we need to make them talk among themselves.

High Performance Computing(HPC)

- Uses both of this concept efficiently

Topics covered

- Using the mpi4py Python module
- Implementing point-to-point communication
- Avoiding deadlock problems
- Collective communication using a broadcast
- Collective communication using the scatter function
- Collective communication using the gather function
- Collective communication using Alltoall
- The reduction operation
- Optimizing communication

Installation Procedure

- The installation procedure of mpi4py on a Windows machine is as follows:
 - **C:>pip install mpi4py**
- Anaconda users must type the following:
 - **C:>conda install mpi4py**
- This implies that the notation used to run the mpi4py examples is as follows:
 - **C:>mpiexec -n x python mpi4py_script_name.py**
- The mpiexec command is the typical way to start parallel jobs: x is the total number of processes to use, while mpi4py_script_name.py is the name of the script to be executed.

Strengths of MPI

- **Standardization:** It is supported by all **High-Performance Computing (HPC)** platforms.
- **Portability:** The changes applied to the source code are minimal, which is useful if you decide to use the application on a different platform that also supports the same standard.
- **Performance:** Manufacturers can create implementations optimized for a specific type of hardware and get better performance.
- **Functionality:** Over 440 routines are defined in MPI-3, but many parallel programs can be written using fewer than even 10 routines.

MPI basic functions

- Import the mpi4py library:

```
from mpi4py import MPI
```

- If we have a number (p of processes) that runs a program, then the processes will have a rank that goes from 0 to $p-1$. In particular, in order to assess the rank of each process, we must use the COMM_WORLD MPI function in particular. This function is called a **communicator**, as it defines its own set of all processes that can communicate together:

```
comm = MPI.COMM_WORLD
```

- Finally, the following Get_rank() function returns rank of the process calling it:

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

SPMD model

- MPI belongs to the **Single Program Multiple Data (SPMD)** programming technique.
- SPMD is a programming technique in which all processes execute the same program, each on different data. The distinction in executions between different processes occurs by differentiating the flow of the program, based on the local rank of the process.
- SPMD is a programming technique in which a single program is executed by several processes at the same time, but each process can operate on different data.
- At the same time, the processes can execute both the same instruction and different instructions.

Modules of MPI

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

Point-to-Point Communication

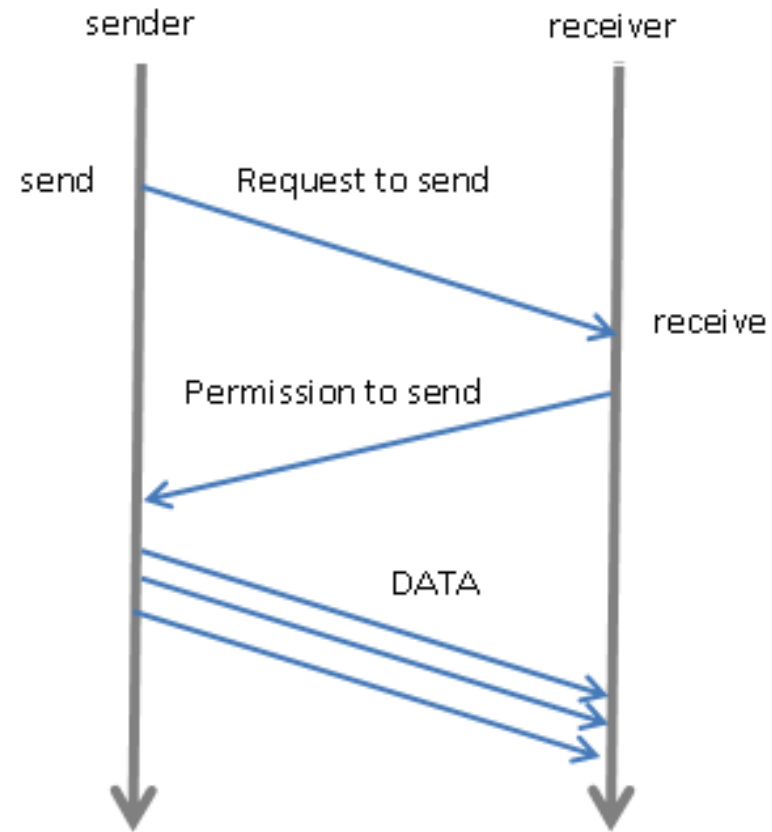
The mpi4py Python module enables point-to-point communication via two functions:

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

The `Comm` parameter, which is short for *communicator*, defines the group of processes that

- may communicate through message passing using `comm = MPI.COMM_WORLD`

Point-to-Point Communication



Point-to-Point Communication

- 3 arguments for communication

(source=0, dest=1, tag=1001)

source = From where you are communicating?

dest = To whom you are communicating?

tag = Are you communicating to me or to someone else?

- Point to Point types
 - Blocking
 - Non-Blocking

Example 1

```
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

```
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = None
    print('Before receiving: ', data)
    data = comm.recv(source=0, tag=11)
    print('After receiving: ', data)
```

Example 2

```
from mpi4py import MPI
```

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

```
if rank == 0:  
    data = {'a': 7, 'b': 3.14}  
    req = comm.isend(data, dest=1, tag=11)  
    req.wait()  
elif rank == 1:  
    req = comm.irecv(source=0, tag=11)  
    data = req.wait()
```

```
from mpi4py import MPI
```

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

```
if rank == 0:  
    data = {'a': 7, 'b': 3.14}  
    req = comm.isend(data, dest=1, tag=11)  
    req.wait()  
elif rank == 1:  
    data = None  
    print('Before receiving: ', data)  
    req = comm.irecv(source=0, tag=11)  
    print('After receiving: ', data)  
    data = req.wait()  
    print('After checking:', data)
```

Example 3

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
rank = comm.Get_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)
```

**mpiexec -n 9 python
pointToPointCommunication.py**

Example 3 - Output

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

my rank is : 7

my rank is : 5

my rank is : 2

my rank is : 6

my rank is : 3

my rank is : 1

sending data hello :to process 8

my rank is : 0

sending data 10000000 to process 4

my rank is : 4

data received is = 10000000

my rank is : 8

data1 received is = hello

Example 4

```
from mpi4py import MPI
import numpy

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

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)

# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```


Blocking Functions

The `comm.send()` and `comm.recv()` functions are blocking functions, which means that they block the caller until the buffered data involved can be used safely.

- **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.
- **Synchronous mode:** The function only gets terminated when the corresponding receive function begins receiving the message

Avoid Deadlocks in point-point comm

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)
if rank==0:
    data_send= "a"
    destination_process = 1
    source_process = 1
    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)
```

Avoid Deadlocks in point-point comm

```
if rank==1:
    data_send= "b"
    destination_process = 0
    source_process = 0
    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)
```

send_recv() – to avoid deadlocks

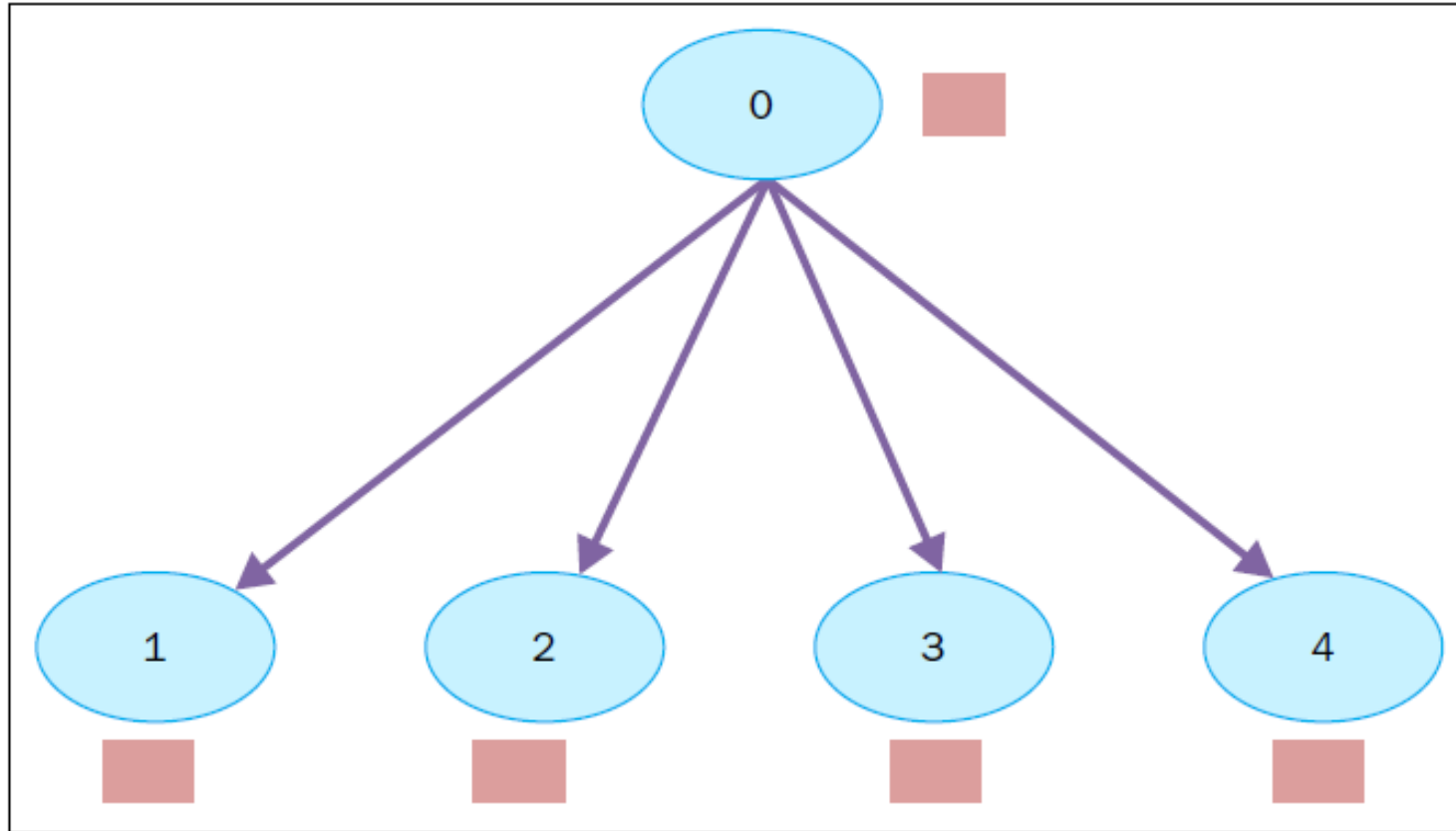
```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.sendrecv(data_send,dest=destination_process,
                                source =source_process)
if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.sendrecv(data_send,dest=destination_process,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

Collective communication routines

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

Broadcast Example

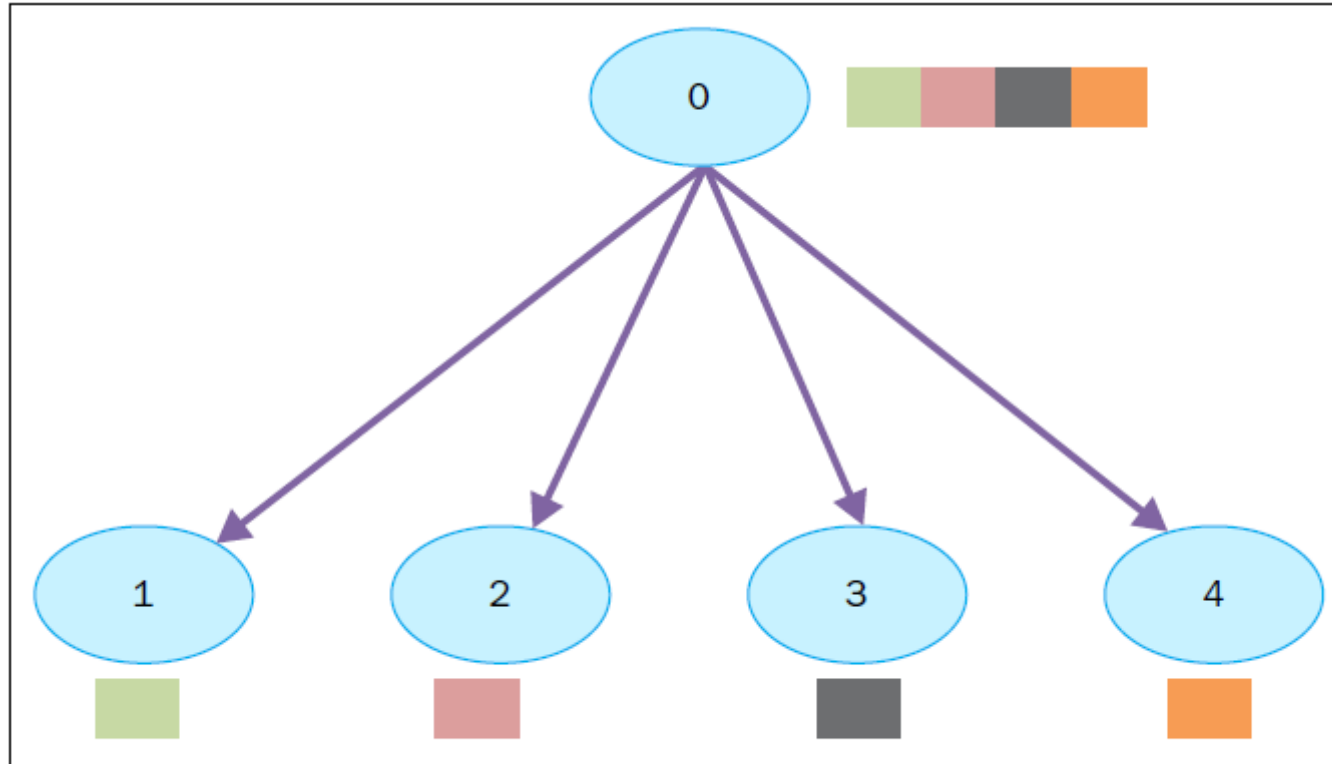


Broadcasting data from process 0 to processes 1, 2, 3, and 4

Broadcast Example

```
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)
```

Scatter Example



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

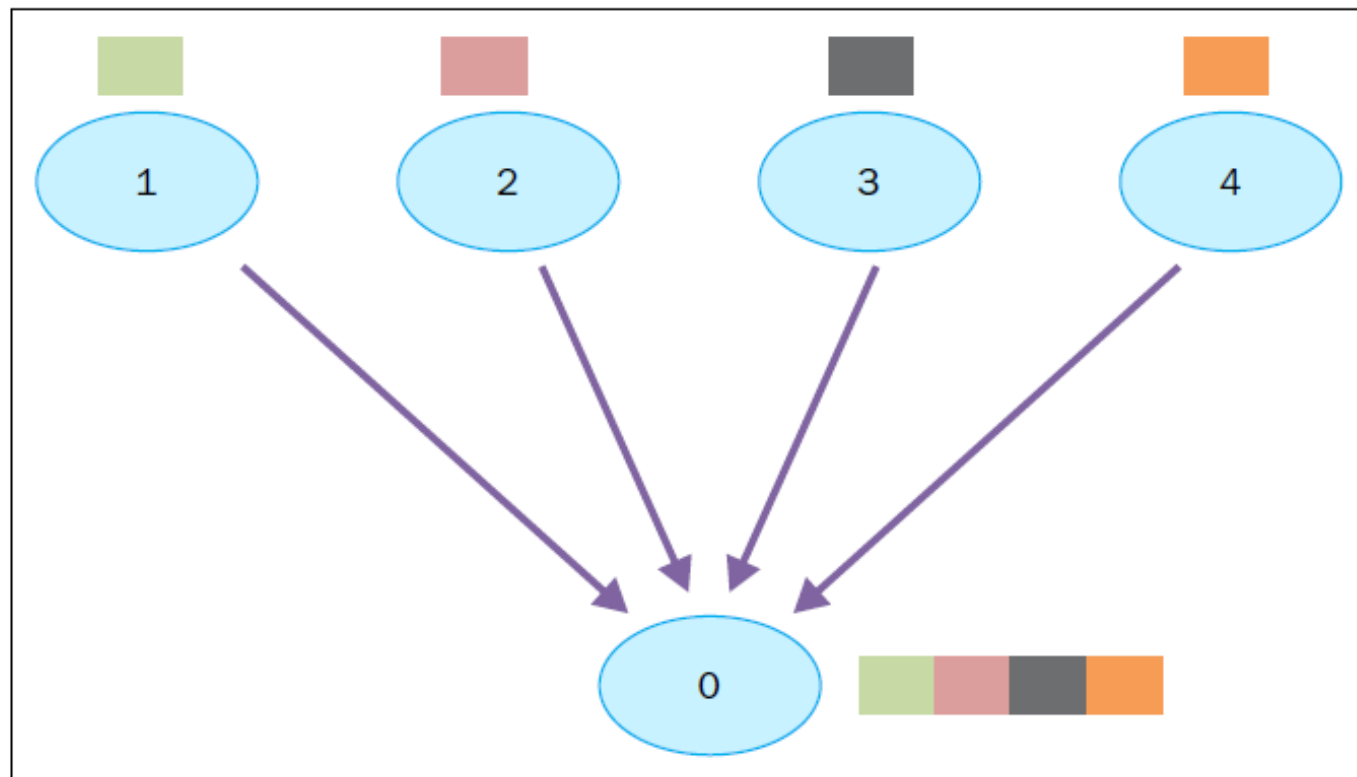
Scatter Example

```
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 " % recvbuf)
```

Scatter Example

- 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`.

Gather Example



Gathering data from processes 1, 2, 3, 4

Gather Example

```
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))
```

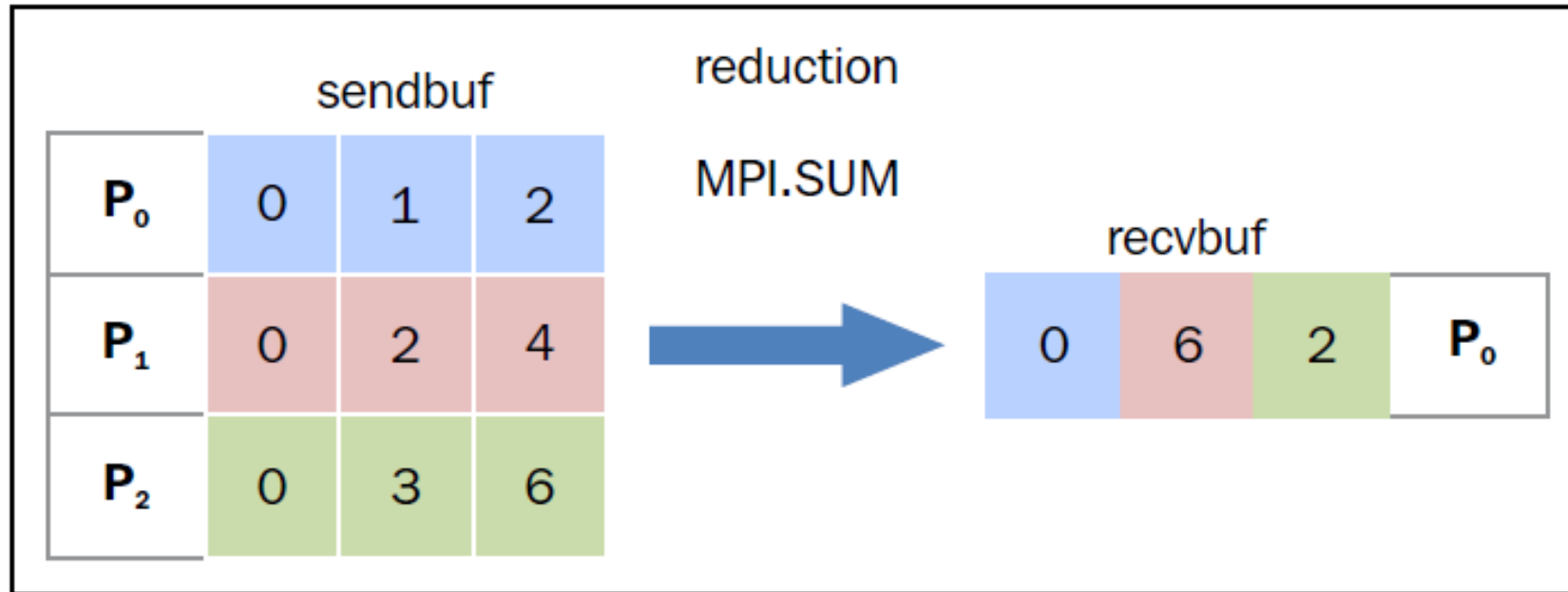
Reduce Example

- `comm.Reduce(sendbuf, recvbuf, rank_of_root_process, op = type_of_reduction_operation)`
- 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

Reduce Example

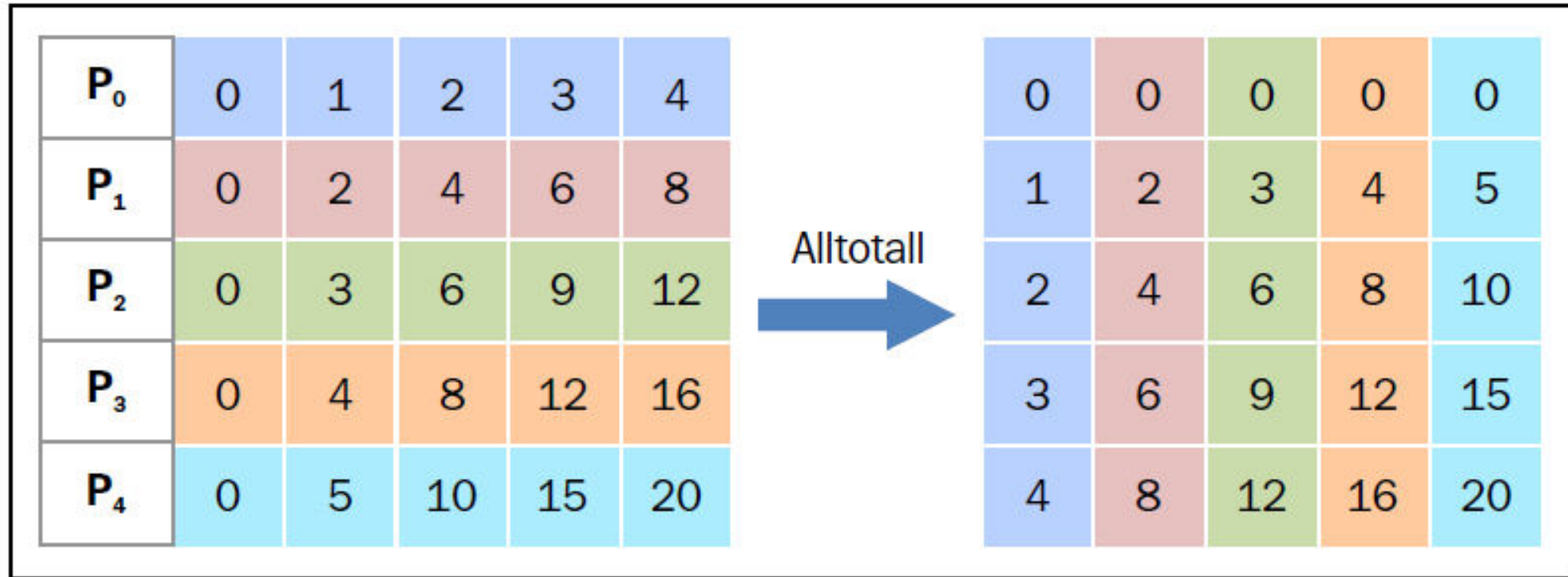
```
import numpy
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.size
rank = comm.rank
array_size = 3
recvdata = numpy.zeros(array_size,dtype=numpy.int)
senddata = (rank+1)*numpy.arange(array_size,dtype=numpy.int)
print(" process %s sending %s " %(rank , senddata))
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
print ('on task',rank,'after Reduce: data = ',recvdata)
```

Reduce Example



The reduction collective communication

All to All Example



The Alltoall collective communication

All to All Example

```
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))
```

Lab 2

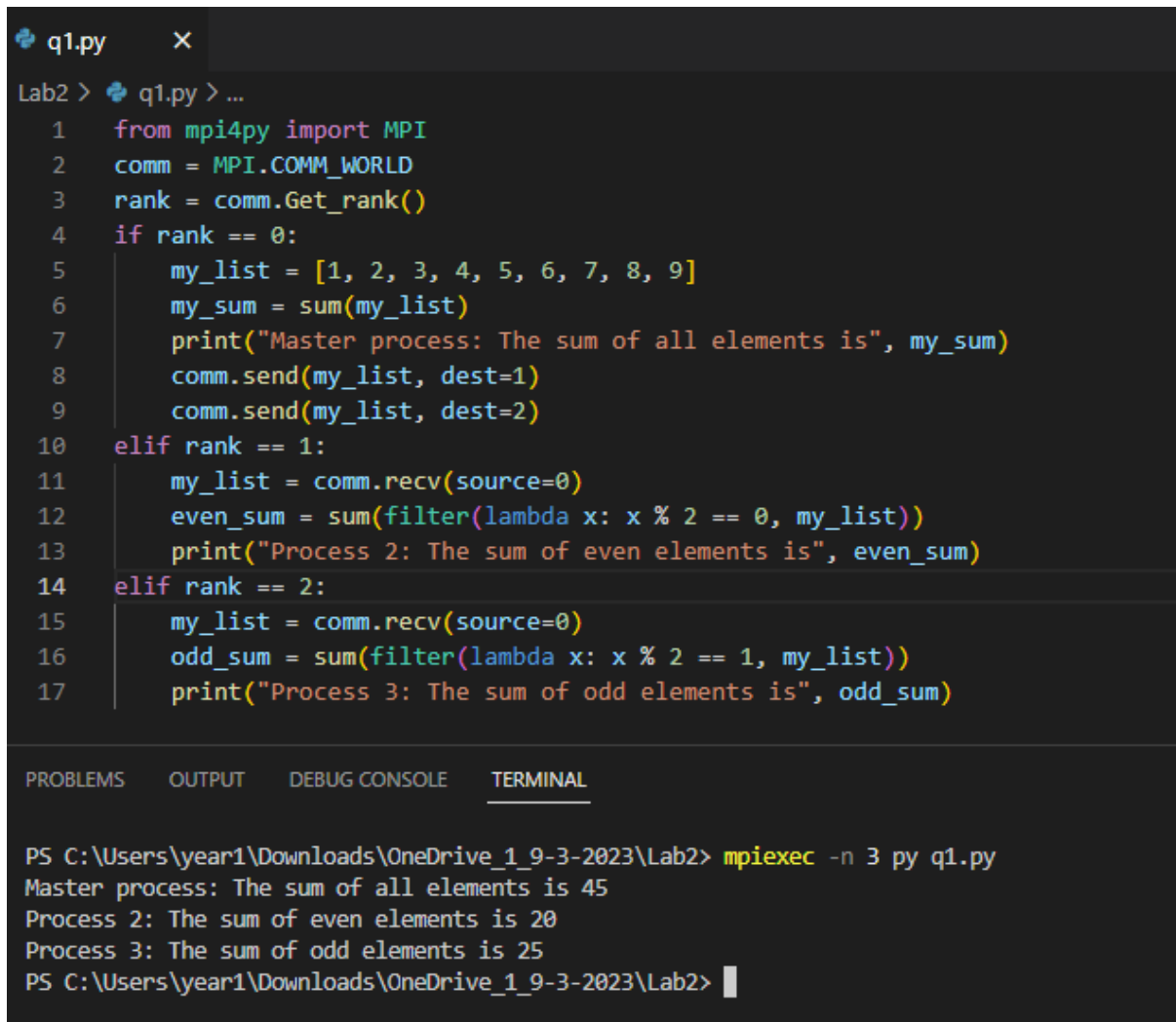
Question 1:

Create a system with 3 processes, where the master process reads a list (of integers) and prints the sum of the list elements. It sends the list to the processes 2 and 3 with 2 finding and printing the sum of even elements in the list and 3 printing the sum of the odd elements in the list.

Code:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    my_sum = sum(my_list)
    print("Master process: The sum of all elements is", my_sum)
    comm.send(my_list, dest=1)
    comm.send(my_list, dest=2)
elif rank == 1:
    my_list = comm.recv(source=0)
    even_sum = sum(filter(lambda x: x % 2 == 0, my_list))
    print("Process 2: The sum of even elements is", even_sum)
elif rank == 2:
    my_list = comm.recv(source=0)
    odd_sum = sum(filter(lambda x: x % 2 == 1, my_list))
    print("Process 3: The sum of odd elements is", odd_sum)
```

Output:



```
q1.py x
Lab2 > q1.py > ...
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  rank = comm.Get_rank()
4  if rank == 0:
5      my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
6      my_sum = sum(my_list)
7      print("Master process: The sum of all elements is", my_sum)
8      comm.send(my_list, dest=1)
9      comm.send(my_list, dest=2)
10 elif rank == 1:
11     my_list = comm.recv(source=0)
12     even_sum = sum(filter(lambda x: x % 2 == 0, my_list))
13     print("Process 2: The sum of even elements is", even_sum)
14 elif rank == 2:
15     my_list = comm.recv(source=0)
16     odd_sum = sum(filter(lambda x: x % 2 == 1, my_list))
17     print("Process 3: The sum of odd elements is", odd_sum)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab2> mpiexec -n 3 py q1.py
Master process: The sum of all elements is 45
Process 2: The sum of even elements is 20
Process 3: The sum of odd elements is 25
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab2> 
```

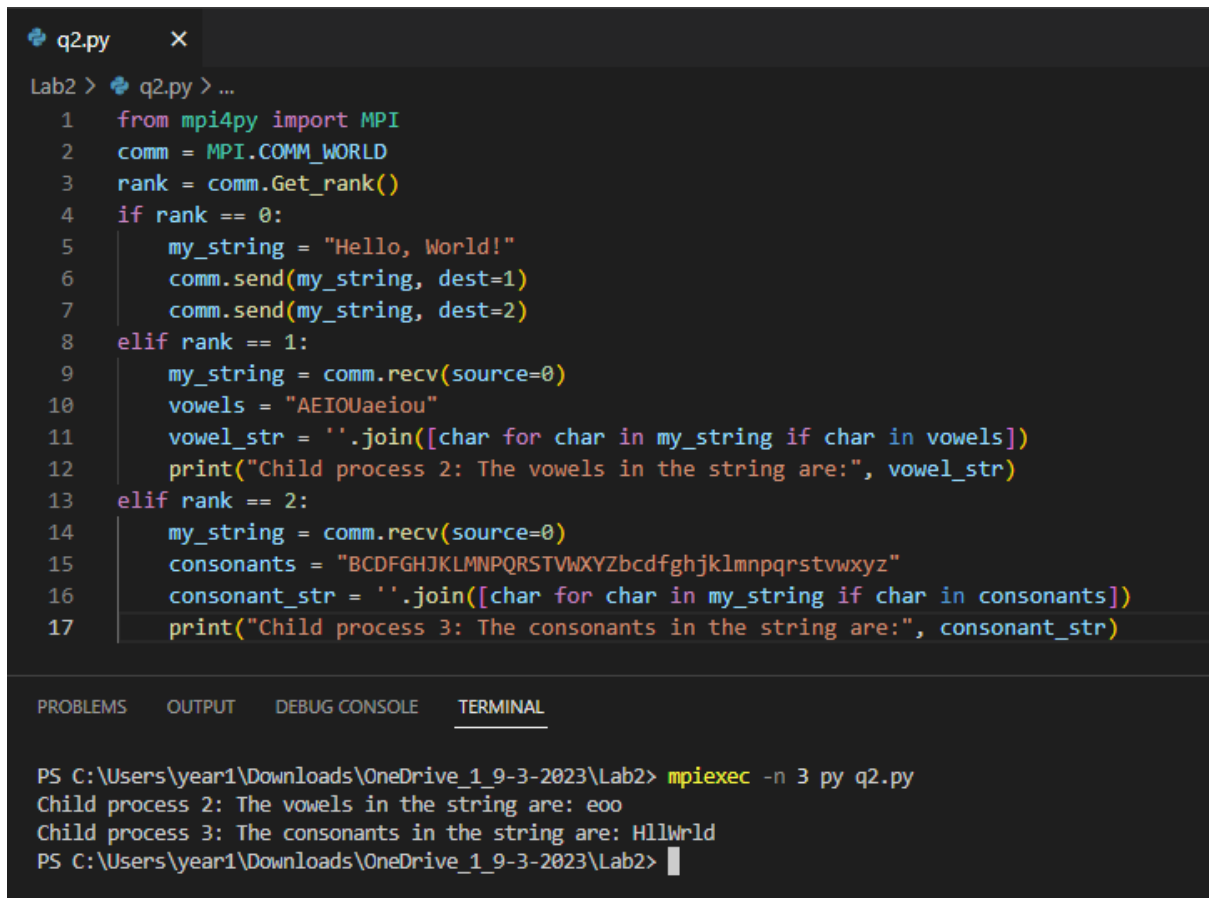
Question 2:

The master process sends a string to the child processes 2 and 3. The child process 2 prints the vowels in the string while the process 3 prints the consonants in the string.

Code:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    my_string = "Hello, World!"
    comm.send(my_string, dest=1)
    comm.send(my_string, dest=2)
elif rank == 1:
    my_string = comm.recv(source=0)
    vowels = "AEIOUaeiou"
    vowel_str = ''.join([char for char in my_string if char in vowels])
    print("Child process 2: The vowels in the string are:", vowel_str)
elif rank == 2:
    my_string = comm.recv(source=0)
    consonants = "BCDFGHJKLMNPQRSTVWXYZbcdfghjklmnpqrstvwxyz"
    consonant_str = ''.join([char for char in my_string if char in
consonants])
    print("Child process 3: The consonants in the string are:", consonant_str)
```

Output:



The screenshot shows a code editor with a file named `q2.py`. The code is a Python script using `mpi4py` to create three processes. Process 0 sends the string "Hello, World!" to processes 1 and 2. Process 1 extracts the vowels from the string, and process 2 extracts the consonants. The terminal output shows the execution of `mpiexec -n 3 py q2.py`, resulting in the following messages:

```
Lab2 > q2.py > ...
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  rank = comm.Get_rank()
4  if rank == 0:
5      my_string = "Hello, World!"
6      comm.send(my_string, dest=1)
7      comm.send(my_string, dest=2)
8  elif rank == 1:
9      my_string = comm.recv(source=0)
10     vowels = "AEIOUaeiou"
11     vowel_str = ''.join([char for char in my_string if char in vowels])
12     print("Child process 2: The vowels in the string are:", vowel_str)
13 elif rank == 2:
14     my_string = comm.recv(source=0)
15     consonants = "BCDFGHJKLMNPQRSTVWXYZbcd fghjklmnpqrstvwxyz"
16     consonant_str = ''.join([char for char in my_string if char in consonants])
17     print("Child process 3: The consonants in the string are:", consonant_str)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab2> mpiexec -n 3 py q2.py
Child process 2: The vowels in the string are: eoo
Child process 3: The consonants in the string are: HllWrld
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab2>
```

Question 3:

Consider a system of 2 processes. The master process generates an array of random numbers of size n. It shares the array with the slave. Slave is asked to do the sum of numbers. The result returned by the slave is printed by the master process. The master process is simultaneously counting the numbers less than 50 in the array and printing the same.

Code:

```
from mpi4py import MPI
import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    n = 10
    arr = [random.randint(1, 100) for _ in range(n)]
    print("Master process: Generated array of random numbers:", arr)
    comm.send(arr, dest=1)
    count = 0
    for num in arr:
        if num < 50:
            count += num
    print("Master process: Count of numbers less than 50:", count)
elif rank == 1:
    arr = comm.recv(source=0)
    my_sum = sum(arr)
    print("Slave process: Sum of numbers in the array:", my_sum)
    comm.send(my_sum, dest=0)
```

Output:

```
q3.py x
Lab2 > q3.py > ...
1  from mpi4py import MPI
2  import random
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5  if rank == 0:
6      n = 10
7      arr = [random.randint(1, 100) for _ in range(n)]
8      print("Master process: Generated array of random numbers:", arr)
9      comm.send(arr, dest=1)
10     count = 0
11     for num in arr:
12         if num < 50:
13             count += num
14     print("Master process: Count of numbers less than 50:", count)
15 elif rank == 1:
16     arr = comm.recv(source=0)
17     my_sum = sum(arr)
18     print("Slave process: Sum of numbers in the array:", my_sum)
19     comm.send(my_sum, dest=0)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab2> mpiexec -n 2 py q3.py
Master process: Generated array of random numbers: [64, 78, 61, 82, 25, 81, 52, 14, 93, 62]
Master process: Count of numbers less than 50: 39
Slave process: Sum of numbers in the array: 612
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab2> 
```

Lab 3

Question 1:

Create n processes in an environment where each process generates an array of size x/n , x being the size of the array held by the master process, which is initially empty. Each slave is generating the numbers and sending them to the master. Master gathers them in an array and prints the count of elements which are multiples of 5. The random numbers can be between 1 and 1000.

Code:

```
from mpi4py import MPI
import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
n = comm.Get_size()
x = 16
if rank == 0:
    my_array = [0] * x
else:
    my_array = None
my_array = [random.randint(1, 1000) for _ in range(x//n)]
my_array = comm.gather(my_array, root=0)
if rank == 0:
    master_array = [num for sublist in my_array for num in sublist]
    count = len([num for num in master_array if num % 5 == 0])
    print("Master Process: ")
    print(f"Count of multiples of 5: {count}")
    print("Array elements:", master_array)
    print("Multiples of 5: ", [num for num in master_array if num % 5 == 0])
```


Output:

```
q1.py  X
q1.py > ...
1  from mpi4py import MPI
2  import random
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5  n = comm.Get_size()
6  x = 16
7  if rank == 0:
8      my_array = [0] * x
9  else:
10     my_array = None
11 my_array = [random.randint(1, 1000) for _ in range(x//n)]
12 my_array = comm.gather(my_array, root=0)
13 if rank == 0:
14     master_array = [num for sublist in my_array for num in sublist]
15     count = len([num for num in master_array if num % 5 == 0])
16     print("Master Process: ")
17     print(f"Count of multiples of 5: {count}")
18     print("Array elements:", master_array)
19     print("Multiples of 5: ", [num for num in master_array if num % 5 == 0])

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3> mpiexec -n 4 py q1.py
Master Process:
Count of multiples of 5: 5
Array elements: [487, 498, 143, 210, 95, 764, 678, 988, 705, 841, 488, 219, 810, 139, 305, 859]
Multiples of 5:  [210, 95, 705, 810, 305]
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3> |
```

Question 2:

Create an environment with n processes. The master process generates random numbers of size 1000. The range of random numbers is between 1 and 50. All the even numbered slaves excluding the master is finding the sum of even numbers in the array. All the odd numbered processes are finding the sum of odd elements in the array. They divide the work equally among them to find the sum. The even sum and odd sum are finally added by the master process which displays the final output. Use appropriate routines for effective transmission of data.

Code:

This is an incorrect answer:

```
from mpi4py import MPI
import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
if rank == 0:
    arr = [random.randint(1, 50) for i in range(1000)]
    for i in range(1, size):
        comm.send(arr, dest=i)
    esum = 0
    osum = 0
    for i in range(1, size):
        if i % 2 == 0:
            esum += comm.recv(source=i)
        else:
            osum += comm.recv(source=i)
    print("Master process Output: ")
    print("Sum of even numbers: ", esum)
    print("Sum of odd numbers: ", osum)
    print("Total sum: ", esum + osum)
else:
    arr = comm.recv(source=0)
    print(f"Slave process {rank} Output: ")
    if rank % 2 == 0:
        filtered_arr = [num for num in arr if num % 2 == 0]
        print("Sum of elements recieved (even):",sum(filtered_arr))
    else:
        filtered_arr = [num for num in arr if num % 2 != 0]
        print("Sum of elements recieved (odd):",sum(filtered_arr))
    comm.send(sum(filtered_arr), dest=0)
```

Output:

```
q2.py x
q2.py > ...
2 import random
3 comm = MPI.COMM_WORLD
4 rank = comm.Get_rank()
5 size = comm.Get_size()
6 if rank == 0:
7     arr = [random.randint(1, 50) for i in range(1000)]
8     for i in range(1, size):
9         comm.send(arr, dest=i)
10    esum = 0
11    osum = 0
12    for i in range(1, size):
13        if i % 2 == 0:
14            esum += comm.recv(source=i)
15        else:
16            osum += comm.recv(source=i)
17    print("Master process Output: ")
18    print("Sum of even numbers: ", esum)
19    print("Sum of odd numbers: ", osum)
20    print("Total sum: ", esum + osum)
21 else:
22     arr = comm.recv(source=0)
23     print(f"Slave process {rank} Output: ")
24     if rank % 2 == 0:
25         filtered_arr = [num for num in arr if num % 2 == 0]
26         print("Sum of elements recieved (even):",sum(filtered_arr))
27     else:
28         filtered_arr = [num for num in arr if num % 2 != 0]
29         print("Sum of elements recieved (odd):",sum(filtered_arr))
30     comm.send(sum(filtered_arr), dest=0)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3> mpiexec -n 4 py q2.py
Slave process 1 Output:
Sum of elements recieved (odd): 13180
Slave process 2 Output:
Sum of elements recieved (even): 12898
Slave process 3 Output:
Sum of elements recieved (odd): 13180
Master process Output:
Sum of even numbers: 12898
Sum of odd numbers: 26360
Total sum: 39258
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3> |
```

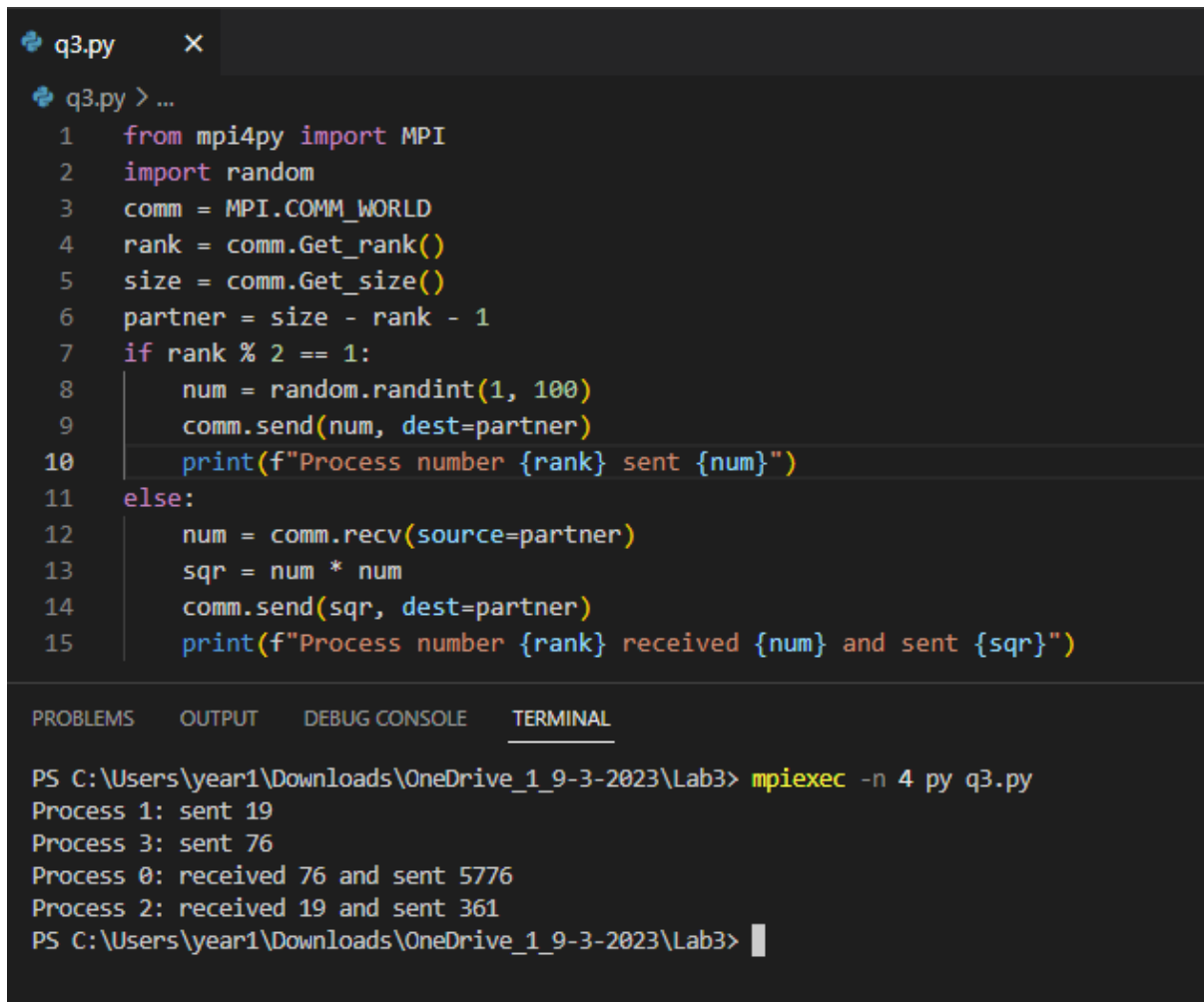
Question 3:

Create an environment where process numbered 0 pairs up with process numbered n-1, process numbered 1 with n-2 etc. For simplicity, you can consider that the system contains even number of processes. The pairs exchange messages between each other. In each pair, the odd numbered process sends a random number (between 1 and 100) and the even numbered process in the pair finds the square of the number received and sends the result to the sender. The received value is printed by the process. Use appropriate routine to do the mentioned task.

Code:

```
from mpi4py import MPI
import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
partner = size - rank - 1
if rank % 2 == 1:
    num = random.randint(1, 100)
    comm.send(num, dest=partner)
    print(f"Process number {rank} sent {num}")
else:
    num = comm.recv(source=partner)
    sqr = num * num
    comm.send(sqr, dest=partner)
    print(f"Process number {rank} received {num} and sent {sqr}")
```

Output:



The screenshot displays a code editor window titled 'q3.py' with a dark theme. The code is a Python script using the mpi4py library for MPI. It defines a communication world, gets the rank and size of the world, and then uses a conditional statement to either send a random number to a partner process or receive one and calculate its square. The script is executed in a terminal window below the editor, showing the output for four processes.

```
q3.py > ...  
1  from mpi4py import MPI  
2  import random  
3  comm = MPI.COMM_WORLD  
4  rank = comm.Get_rank()  
5  size = comm.Get_size()  
6  partner = size - rank - 1  
7  if rank % 2 == 1:  
8      num = random.randint(1, 100)  
9      comm.send(num, dest=partner)  
10     print(f"Process number {rank} sent {num}")  
11 else:  
12     num = comm.recv(source=partner)  
13     sqr = num * num  
14     comm.send(sqr, dest=partner)  
15     print(f"Process number {rank} received {num} and sent {sqr}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3> mpiexec -n 4 py q3.py  
Process 1: sent 19  
Process 3: sent 76  
Process 0: received 76 and sent 5776  
Process 2: received 19 and sent 361  
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3>
```

Question 4:

Write parallel programs to compute $n!$ in each of the following ways and assess their performance in terms of speed up factor. The number n may be even or odd but is a positive constant.

a. Compute $n!$ using 2 concurrent processes, each computing approximately half of the complete sequence. A master process then combines the two partial results.

b. Compute $n!$ using a producer process and a consumer process connected. The producer produces the numbers 1, 2, 3... n in a sequence. The consumer accepts the sequence of numbers from the producer and accumulates the result (i.e.) $1 \times 2 \times 3 \dots$

Code:

q1_a.py:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
n = 5
start = MPI.Wtime()
if rank == 0:
    mid = n // 2
    nums1 = []
    nums2 = []
    for i in range(1, mid):
        nums1.append(i)
    comm.send(nums1, dest=1)

    for i in range(mid+1, n+1):
        nums2.append(i)
    comm.send(nums2, dest=2)

    r1 = comm.recv(source=1)
    r2 = comm.recv(source=2)
    res = r1 * r2
    stop = MPI.Wtime()
    print(f"Factorial of {n} is: {res}")
    print(f"Time taken: {stop-start} seconds")
elif rank == 1:
    nums1 = comm.recv(source=0)
```

```
fact1 = 1
for i in nums1:
    fact1 *= i
comm.send(fact1, dest=0)
elif rank == 2:
    nums2 = comm.recv(source=0)
    fact1 = 1
    for i in nums2:
        fact1 *= i
    comm.send(fact1, dest=0)
```

q1_b.py:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
n = 5
start = MPI.Wtime()

# Consumer (Master)
if rank == 0:
    nums = comm.recv(source=1)
    fact = 1
    for i in nums:
        fact *= i
    stop = MPI.Wtime()
    print(f"Factorial of {n} is: {fact}")
    print(f"Time taken: {stop-start} seconds")

# Producer (Slave)
elif rank == 1:
    nums = []
    for i in range(1, n+1):
        nums.append(i)
    comm.send(nums, dest=0)
```

Name - Shubham Garg

Reg No – BL.EN.U4CSE20158

Output:

```
q1_a.py > ...
4 n = 5
5 start = MPI.Wtime()
6 if rank == 0:
7     mid = n // 2
8     nums1 = []
9     nums2 = []
10    for i in range(1,mid):
11        nums1.append(i)
12    comm.send(nums1, dest=1)
13
14    for i in range(mid+1,n+1):
15        nums2.append(i)
16    comm.send(nums2, dest=2)
17
18    r1 = comm.recv(source=1)
19    r2 = comm.recv(source=2)
20    res = r1 * r2
21    stop = MPI.Wtime()
22    print(f"Factorial of {n} is: {res}")
```

PROBLEMS OUTPUT TERMINAL COMMENTS DEBUG CONSOLE

```
PS D:\Sem 6\DS\Lab 4> mpiexec -n 3 py q1_a.py
Factorial of 5 is: 60
Time taken: 0.0005104000010760501 seconds
● PS D:\Sem 6\DS\Lab 4> mpiexec -n 3 py q1_b.py
○ Factorial of 5 is: 120
Time taken: 0.000228899996727705 seconds
PS D:\Sem 6\DS\Lab 4> █
```


Lab 4

Question 1:

Write parallel programs to compute $n!$ in each of the following ways and assess their performance in terms of speed up factor. The number n may be even or odd but is a positive constant.

- Compute $n!$ using 2 concurrent processes, each computing approximately half of the complete sequence. A master process then combines the two partial results.
- Compute $n!$ using a producer process and a consumer process connected. The producer produces the numbers 1, 2, 3... n in a sequence. The consumer accepts the sequence of numbers from the producer and accumulates the result (i.e.) $1 \times 2 \times 3 \dots$

Code:

q1_a.py:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
n = 5
start = MPI.Wtime()
if rank == 0:
    mid = n // 2
    nums1 = []
    nums2 = []
    for i in range(1, mid):
        nums1.append(i)
        comm.send(nums1, dest=1)

    for i in range(mid, n+1):
        nums2.append(i)
        comm.send(nums2, dest=2)

    r1 = comm.recv(source=1)
    r2 = comm.recv(source=2)
    res = r1 * r2
    stop = MPI.Wtime()
    print(f"Factorial of {n} is: {res}")
    print(f"Time taken: {stop-start} seconds")
elif rank == 1:
    nums1 = comm.recv(source=0)
    fact1 = 1
```

```
    for i in nums1:
        fact1 *= i
    comm.send(fact1, dest=0)
elif rank == 2:
    nums2 = comm.recv(source=0)
    fact1 = 1
    for i in nums2:
        fact1 *= i
    comm.send(fact1, dest=0)
```

q1_b.py:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
n = 5
start = MPI.Wtime()

# Consumer (Master)
if rank == 0:
    nums = comm.recv(source=1)
    fact = 1
    for i in nums:
        fact *= i
    stop = MPI.Wtime()
    print(f"Factorial of {n} is: {fact}")
    print(f"Time taken: {stop-start} seconds")

# Producer (Slave)
elif rank == 1:
    nums = []
    for i in range(1,n+1):
        nums.append(i)
    comm.send(nums, dest=0)
```

Output:



```
q1_a.py x q1_b.py
Lab4 > q1_a.py > ...
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  rank = comm.Get_rank()
4  n = 5
5  start = MPI.Wtime()
6  if rank == 0:
7      mid = n // 2
8      nums1 = []
9      nums2 = []
10     for i in range(1,mid):
11         nums1.append(i)
12     comm.send(nums1, dest=1)
13
14     for i in range(mid,n+1):
15         nums2.append(i)
16     comm.send(nums2, dest=2)
17
18     r1 = comm.recv(source=1)
19     r2 = comm.recv(source=2)
20     res = r1 * r2
21     stop = MPI.Wtime()
22     print(f"Factorial of {n} is: {res}")
23     print(f"Time taken: {stop-start} seconds")
24 elif rank == 1:
25     nums1 = comm.recv(source=0)
26     fact1 = 1
27     for i in nums1:
28         fact1 *= i
29     comm.send(fact1, dest=0)
30 elif rank == 2:
31     nums2 = comm.recv(source=0)
32     fact1 = 1
33     for i in nums2:
34         fact1 *= i
35     comm.send(fact1, dest=0)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3\Lab4> mpiexec -n 4 py q1_a.py
Factorial of 5 is: 120
Time taken: 0.002913600002836226 seconds
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3\Lab4> mpiexec -n 4 py q1_b.py
Factorial of 5 is: 120
Time taken: 0.000879799998932937 seconds
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3\Lab4> 
```

Question 2:

You have been given a data set represented as a $n \times n$ matrix. The target is to add all the elements of the matrix using parallel computation. There are P processors which can perform the computation parallelly. Assume that the master process holds the input matrix and it sends smaller matrices to P slaves to do the computation. Each slave performs the addition of smaller matrices and sends the resultant sum to the master. The master adds up all the results to get the final sum. Please note that the slaves alone perform the addition of the matrices, the master adds the partial sums to get the result.

Code:

```
from mpi4py import MPI
import random

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

n = 4
matrix = None

if rank == 0:
    matrix = [[random.randint(1, 100) for _ in range(n)] for _ in range(n)]
    print("Input matrix:")
    for row in matrix:
        print(row)

    flat_matrix = [elem for row in matrix for elem in row]
    num_chunks = num_processors - 1
    chunk_size = len(flat_matrix) // num_chunks
    chunks = [flat_matrix[i:i + chunk_size] for i in range(0, len(flat_matrix),
chunk_size)]
else:
    chunks = None

chunk = comm.scatter(chunks, root=0)

partial_sum = sum(chunk) if chunk is not None else None

result_sum = comm.gather(partial_sum, root=0)

if rank == 0:
    total_sum = sum(result_sum[1:])
    print("Total sum:", total_sum)
```

Output:

```
q2.py x
Lab4 > q2.py > ...
1  from mpi4py import MPI
2  import random
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  num_processors = comm.Get_size()
7
8  n = 4
9  matrix = None
10
11 if rank == 0:
12     matrix = [[random.randint(1, 100) for _ in range(n)] for _ in range(n)]
13     print("Input matrix:")
14     for row in matrix:
15         print(row)
16
17     flat_matrix = [elem for row in matrix for elem in row]
18     num_chunks = num_processors - 1
19     chunk_size = len(flat_matrix) // num_chunks
20     chunks = [flat_matrix[i:i + chunk_size] for i in range(0, len(flat_matrix), chunk_size)]
21 else:
22     chunks = None
23
24 chunk = comm.scatter(chunks, root=0)
25
26 partial_sum = sum(chunk) if chunk is not None else None
27
28 result_sum = comm.gather(partial_sum, root=0)
29
30 if rank == 0:
31     total_sum = sum(result_sum[1:])
32     print("Total sum:", total_sum)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3\Lab4> mpiexec -n 4 py q2.py
Input matrix:
[66, 49, 60, 59]
[42, 97, 8, 71]
[60, 85, 18, 84]
[82, 91, 1, 83]
Total sum: 680
PS C:\Users\year1\Downloads\OneDrive_1_9-3-2023\Lab3\Lab4> 
```

Lab 5

Question 1:

You have an image which has to be relocated from (0,0) to (10, 10) and also rotated to an angle of 30 degrees. You are using p processes to do the task using a row major division. Perform the task using embarrassingly parallel computation where the slaves update the new computed data to the master and the master just overwrites the updated data to the array maintained. The slaves just compute the new values and display it on the screen. Print the speed up factor and efficiency.

Code:

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

WIDTH = 1000
HEIGHT = 1000

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

theta = np.radians(30)

rows_per_process = int(math.ceil(float(HEIGHT) / size))
start_row = rank * rows_per_process
end_row = min((rank + 1) * rows_per_process, HEIGHT)

image = np.zeros((HEIGHT, WIDTH))

image[:, :] = np.random.rand(HEIGHT, WIDTH)

start_time = MPI.Wtime()

new_data = np.zeros((rows_per_process, WIDTH))
for i in range(start_row, end_row):
    for j in range(WIDTH):
        x = j * np.cos(theta) - i * np.sin(theta) + 10
        y = j * np.sin(theta) + i * np.cos(theta) + 10
```

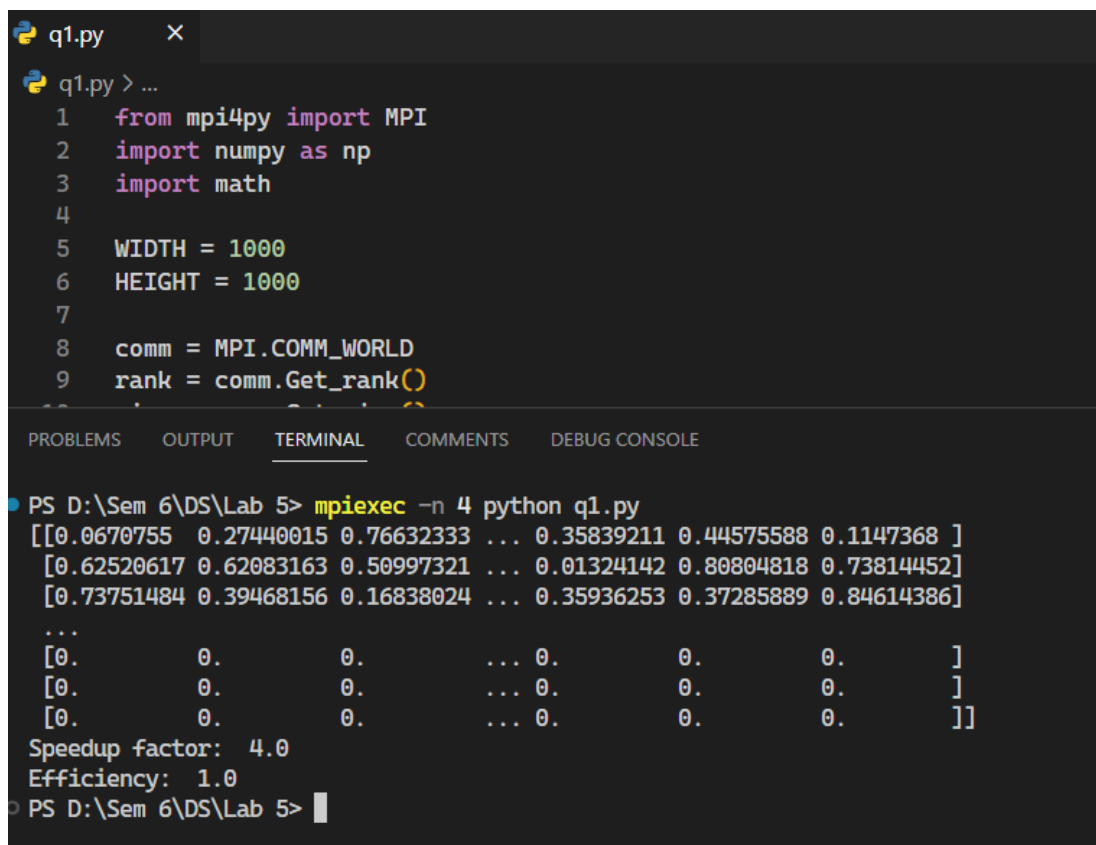
```
        if x >= 0 and x < WIDTH and y >= 0 and y < HEIGHT:
            new_data[i - start_row, j] = image[int(y), int(x)]

if rank != 0:
    comm.send(new_data, dest=0)
else:
    for i in range(1, size):
        slave_new_data = comm.recv(source=i)
        slave_start_row = i * rows_per_process
        slave_end_row = min((i + 1) * rows_per_process, HEIGHT)
        image[slave_start_row:slave_end_row, :] = slave_new_data

    print(image)

end_time = MPI.Wtime()
elapsed_time = end_time - start_time
sequential_time = elapsed_time * size
speedup_factor = sequential_time / elapsed_time
efficiency = speedup_factor / size
print("Speedup factor: ", speedup_factor)
print("Efficiency: ", efficiency)
```

Output:



```
q1.py  x
q1.py > ...
1  from mpi4py import MPI
2  import numpy as np
3  import math
4
5  WIDTH = 1000
6  HEIGHT = 1000
7
8  comm = MPI.COMM_WORLD
9  rank = comm.Get_rank()
...
PROBLEMS  OUTPUT  TERMINAL  COMMENTS  DEBUG CONSOLE
PS D:\Sem 6\DS\Lab 5> mpiexec -n 4 python q1.py
[[[0.0670755  0.27440015 0.76632333 ... 0.35839211 0.44575588 0.1147368 ]
 [0.62520617 0.62083163 0.50997321 ... 0.01324142 0.80804818 0.73814452]
 [0.73751484 0.39468156 0.16838024 ... 0.35936253 0.37285889 0.84614386]
 ...
 [0.         0.         0.         ... 0.         0.         0.         ]
 [0.         0.         0.         ... 0.         0.         0.         ]
 [0.         0.         0.         ... 0.         0.         0.         ]]]
Speedup factor: 4.0
Efficiency: 1.0
PS D:\Sem 6\DS\Lab 5>
```

Question 2:

Assume that you are finding the sum of 'n' numbers using 8 CPUs. The numbers are held by the master process P0 and the task is assigned using binary divide and conquer approach. The processes divide the numbers and find the partial sum. The final sum is conquered at each level and the result is printed by the master process.

Code:

```
from mpi4py import MPI
import numpy as np

def divide_and_conquer_sum(rank, size, data):
    partial_sum = np.sum(data)

    step = 1
    while step < size:
        if rank % (2 * step) == 0:
            other_partial_sum = rank + step
            if other_partial_sum < size:
                recv_data = np.empty(1, dtype=np.float64)
                MPI.COMM_WORLD.Recv(recv_data,
source=other_partial_sum)
                partial_sum += recv_data[0]

            elif (rank - step) % (2 * step) == 0:
                MPI.COMM_WORLD.Send(np.array([partial_sum],
dtype=np.float64), dest=rank - step)
                break

        step *= 2

    return partial_sum

def main():
    MPI.COMM_WORLD.Barrier()
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    n = 100
    np.random.seed(rank)
```



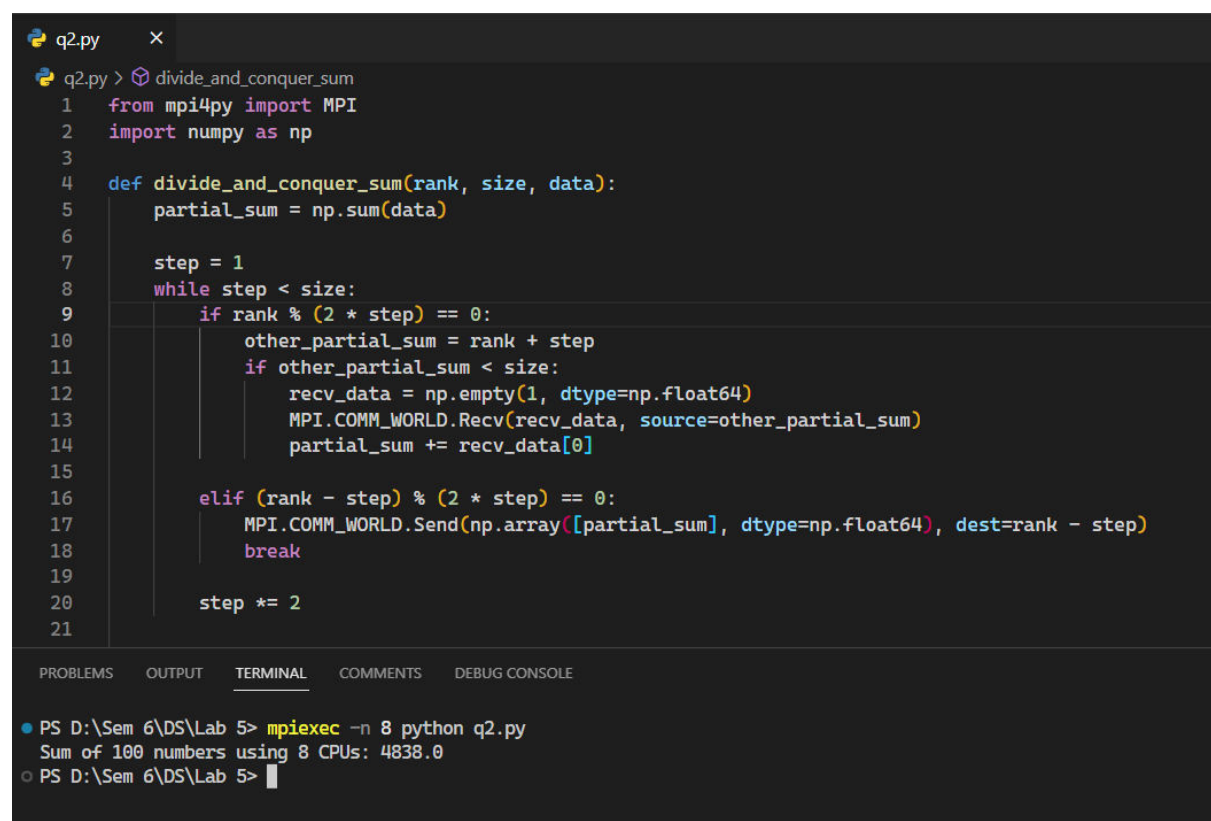
```
numbers_per_process = int(n / size)
data = np.random.randint(0, 100, size=numbers_per_process)

partial_sum = divide_and_conquer_sum(rank, size, data)

if rank == 0:
    print(f"Sum of {n} numbers using {size} CPUs: {partial_sum}")

if __name__ == "__main__":
    main()
```

Output:



```
q2.py x
q2.py > divide_and_conquer_sum
1  from mpi4py import MPI
2  import numpy as np
3
4  def divide_and_conquer_sum(rank, size, data):
5      partial_sum = np.sum(data)
6
7      step = 1
8      while step < size:
9          if rank % (2 * step) == 0:
10             other_partial_sum = rank + step
11             if other_partial_sum < size:
12                 recv_data = np.empty(1, dtype=np.float64)
13                 MPI.COMM_WORLD.Recv(recv_data, source=other_partial_sum)
14                 partial_sum += recv_data[0]
15
16             elif (rank - step) % (2 * step) == 0:
17                 MPI.COMM_WORLD.Send(np.array([partial_sum], dtype=np.float64), dest=rank - step)
18                 break
19
20             step *= 2
21
22
PROBLEMS  OUTPUT  TERMINAL  COMMENTS  DEBUG CONSOLE
● PS D:\Sem 6\DS\Lab 5> mpiexec -n 8 python q2.py
Sum of 100 numbers using 8 CPUs: 4838.0
○ PS D:\Sem 6\DS\Lab 5> █
```

Question 3:

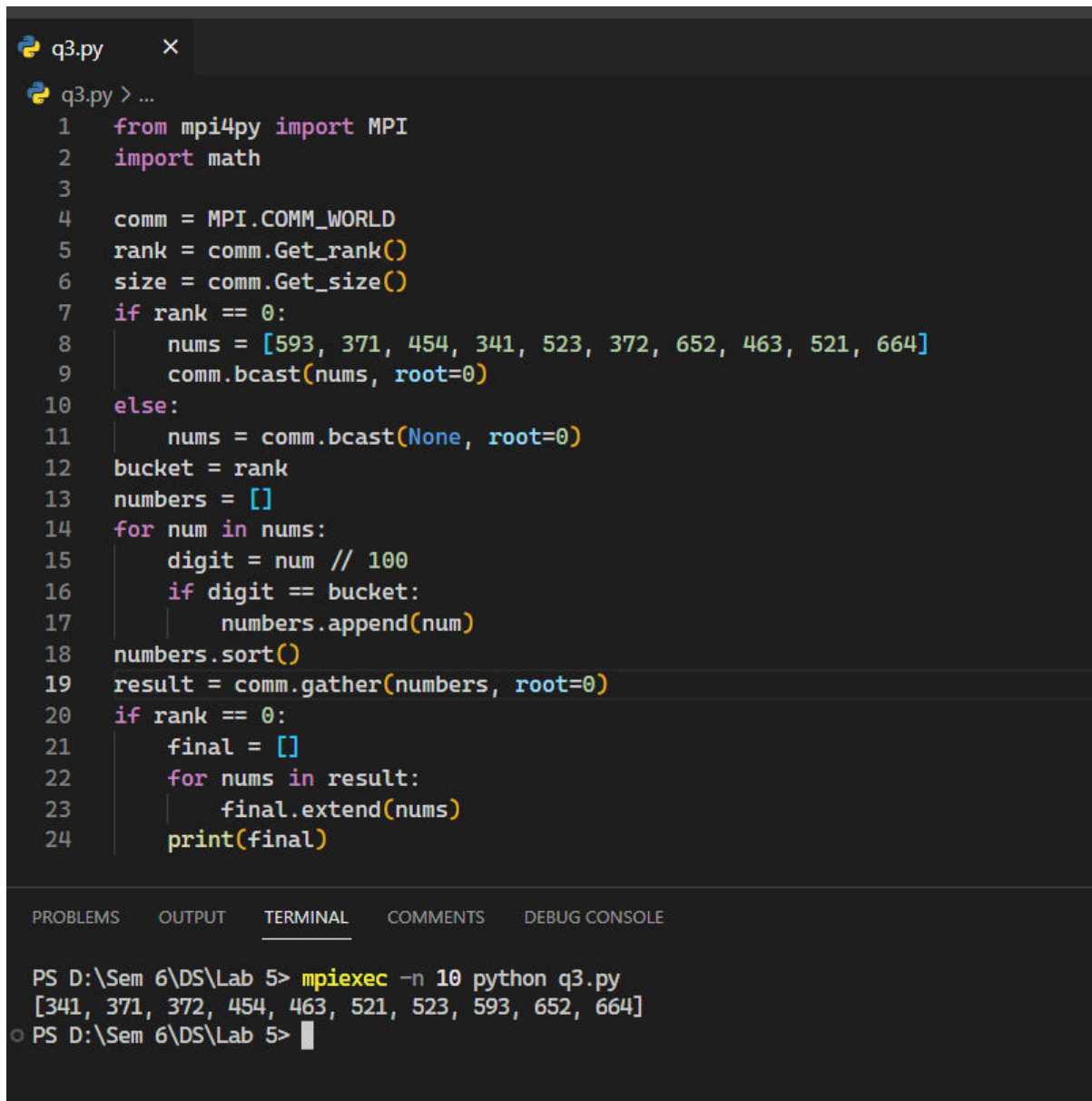
Sort the given 'n' numbers using bucket sort and portioning approach learnt in the class. The numbers falling into a single bucket are sorted using the inbuilt sort() function. The numbers are initially with the master and the numbers are broadcasted. There are 10 processors numbered 0 to 9 who store the numbers in each bucket based on the digit. Assume the number of digits in the input is to a maximum of 3.

Code:

```
from mpi4py import MPI
import math

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
if rank == 0:
    nums = [593, 371, 454, 341, 523, 372, 652, 463, 521, 664]
    comm.bcast(nums, root=0)
else:
    nums = comm.bcast(None, root=0)
bucket = rank
numbers = []
for num in nums:
    digit = num // 100
    if digit == bucket:
        numbers.append(num)
numbers.sort()
result = comm.gather(numbers, root=0)
if rank == 0:
    final = []
    for nums in result:
        final.extend(nums)
    print(final)
```

Output:



```
q3.py x
q3.py > ...
1  from mpi4py import MPI
2  import math
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  size = comm.Get_size()
7  if rank == 0:
8      nums = [593, 371, 454, 341, 523, 372, 652, 463, 521, 664]
9      comm.bcast(nums, root=0)
10 else:
11     nums = comm.bcast(None, root=0)
12     bucket = rank
13     numbers = []
14     for num in nums:
15         digit = num // 100
16         if digit == bucket:
17             numbers.append(num)
18     numbers.sort()
19     result = comm.gather(numbers, root=0)
20     if rank == 0:
21         final = []
22         for nums in result:
23             final.extend(nums)
24         print(final)
```

PROBLEMS OUTPUT TERMINAL COMMENTS DEBUG CONSOLE

```
PS D:\Sem 6\DS\Lab 5> mpiexec -n 10 python q3.py
[341, 371, 372, 454, 463, 521, 523, 593, 652, 664]
PS D:\Sem 6\DS\Lab 5> █
```