IT301 Assignment 8

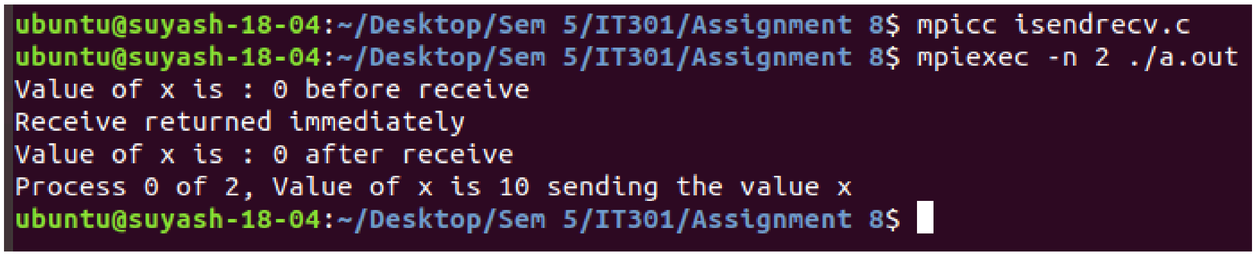
NAME: SUYASH CHINTAWAR
ROLL NO.: 191IT109
TOPIC: MPI PROGRAMMING - 2

NOTE: The codes have not been attached as already given in problems.

Q1. MPI non blocking Send and Receive(). Record the observation with and without MPI_Wait(). In each case observe whether the process was waiting for completing send/recv or continuing its execution.

(a) Note down results by commenting MPI_Wait()

Output:

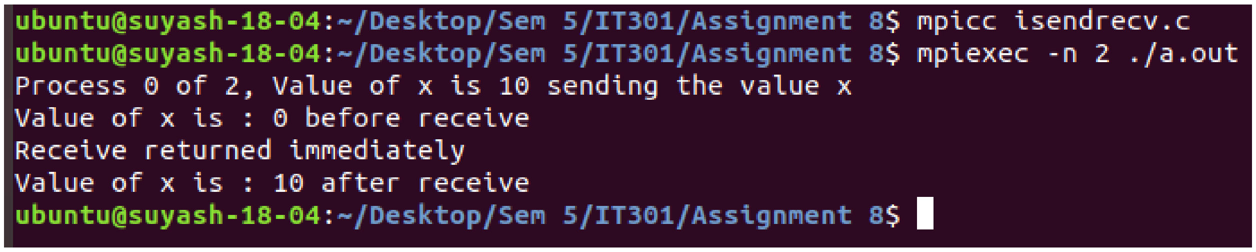


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpicc isendrecv.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 2 ./a.out
Value of x is : 0 before receive
Receive returned immediately
Value of x is : 0 after receive
Process 0 of 2, Value of x is 10 sending the value x
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: The value of x is not received by process 1. That is because it is a non-blocking type of send/receive. As the process does not wait for the communication to terminate, the value of x is not received.

(b) Note down result by uncommenting MPI_Wait()

Output:

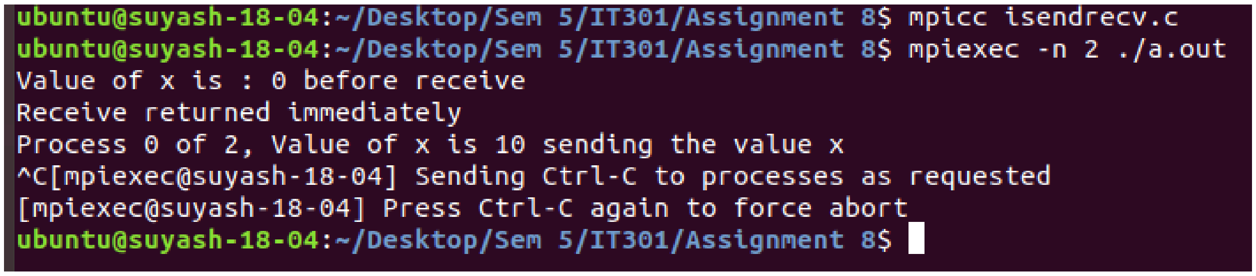


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpicc isendrecv.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 2 ./a.out
Process 0 of 2, Value of x is 10 sending the value x
Value of x is : 0 before receive
Receive returned immediately
Value of x is : 10 after receive
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: In this case, the processes wait for each other so that the transaction is carried out successfully. This ensures that the value sent process 0 is received by process 1 in the variable x and hence we see the same in the above screen shot.

(c) Note down the result by having a mismatched tag .

Output:

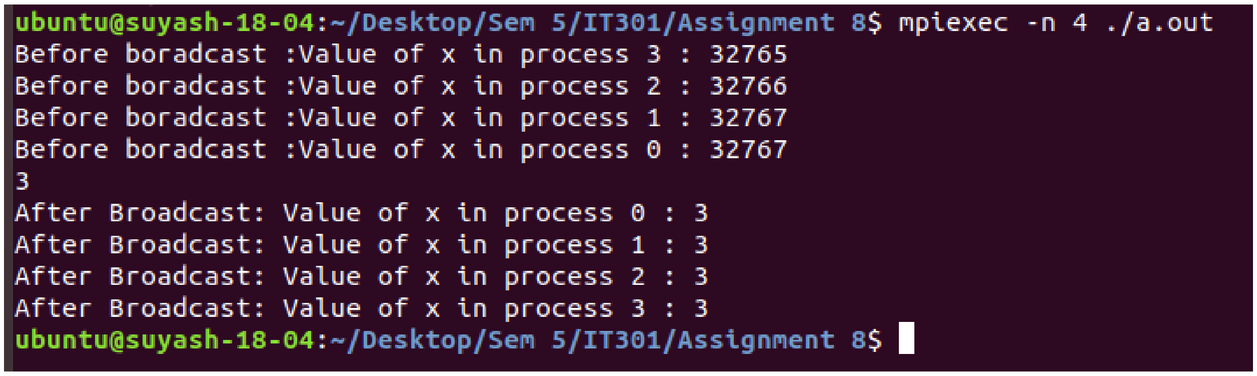


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpicc isendrecv.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 2 ./a.out
Value of x is : 0 before receive
Receive returned immediately
Process 0 of 2, Value of x is 10 sending the value x
^C[mpiexec@suyash-18-04] Sending Ctrl-C to processes as requested
[mpiexec@suyash-18-04] Press Ctrl-C again to force abort
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: When the MPI_Wait() function is uncommented, the receiving process waits for the sender, which, in this case, does not exist because of tag mismatch. This leads to deadlock and the program doesn't stop. However, when the MPI_Wait() function is commented, which restores the non-blocking property of the processes, the process does not wait for the value to be received and in that case the program stops without receiving any value. The output in this case will be similar to the output of 1(a).

Q2. Demonstration of Bcast()

Output:

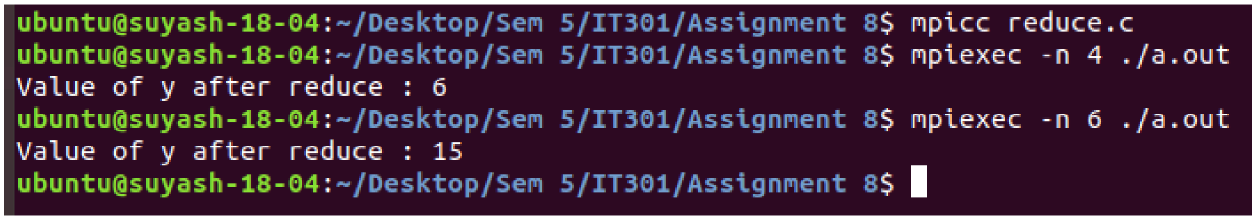


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 4 ./a.out
Before boradcast :Value of x in process 3 : 32765
Before boradcast :Value of x in process 2 : 32766
Before boradcast :Value of x in process 1 : 32767
Before boradcast :Value of x in process 0 : 32767
3
After Broadcast: Value of x in process 0 : 3
After Broadcast: Value of x in process 1 : 3
After Broadcast: Value of x in process 2 : 3
After Broadcast: Value of x in process 3 : 3
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: Bcast() function says that the sending value should be broadcasted to all the processes. This solves the problem of explicitly writing multiple receive statements for each of the processes. There is only one sender in this case (process 0). The value of x=3 is taken as input for the process 0, and we can see that before broadcasting the value of x is a garbage value. But after broadcasting, the value of the variable x is updated for every process.

Q3. Demonstration of Reduce(). Note down the observation and explain the result.

Output:

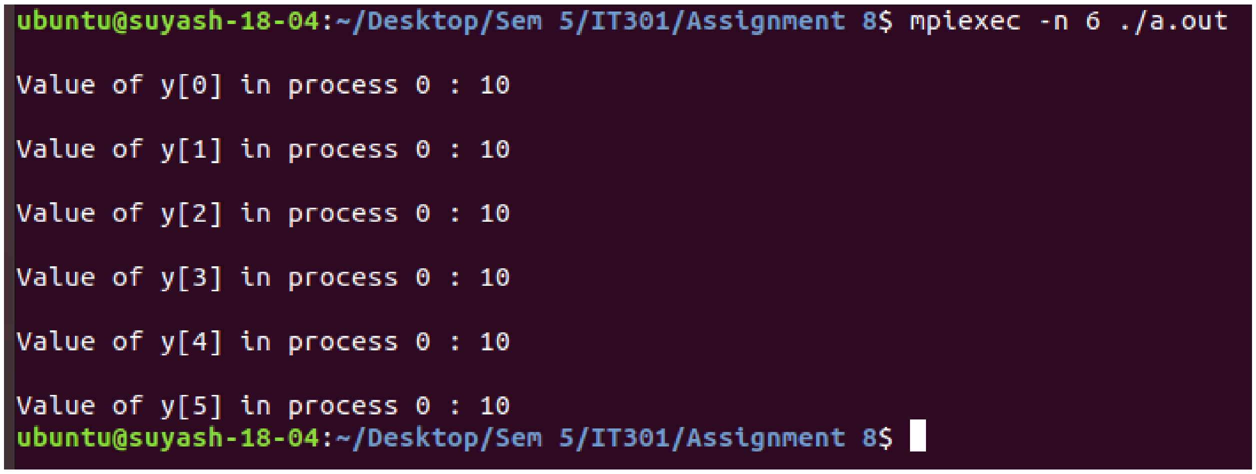


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpicc reduce.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 4 ./a.out
Value of y after reduce : 6
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 6 ./a.out
Value of y after reduce : 15
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: Reduce() function works similar to the reduction clause in openmp. Here there is only one process where the result after reduction is stored into. In our case it is process 0. The ranks of the processes are added using the "+" operator into process 0. Hence, when num_processes =4, the ranks 0,1,2,3 of the processes are added yielding the result to be 0+1+2+3 = 6.

Q4. Demonstration of MPI_Gather(). Note down the observation and explain the result.

Output:

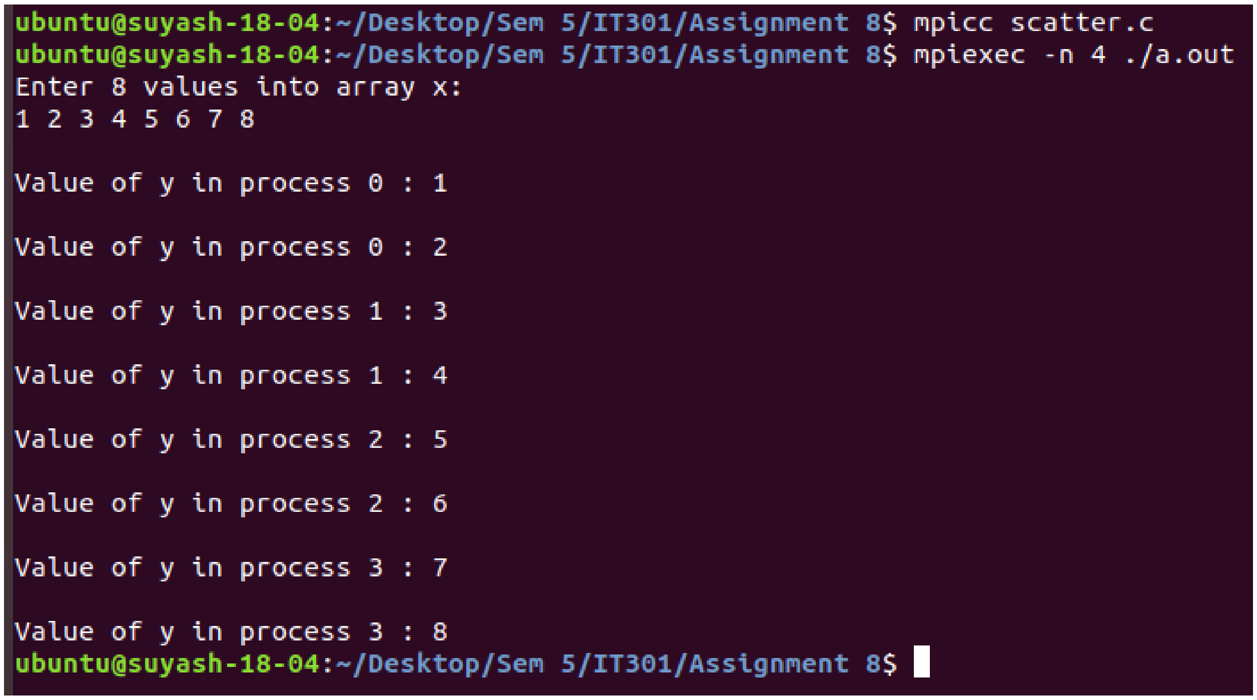


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 6 ./a.out

Value of y[0] in process 0 : 10

Value of y[1] in process 0 : 10

Value of y[2] in process 0 : 10

Value of y[3] in process 0 : 10

Value of y[4] in process 0 : 10

Value of y[5] in process 0 : 10
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: MPI_Gather() is used to accumulate the values of variables into one process. The variable x=10 is initialized for each process and is gathered into the array 'y' in process 0. If there are 6 processes, all the 6 array elements in 'y' gets the value of 10 (which is obtained from x=10)

Q5. Demonstration of MPI_Scatter(). The Program is hardcoded to work with 4 processes receiving two chunks from the array.

Output:



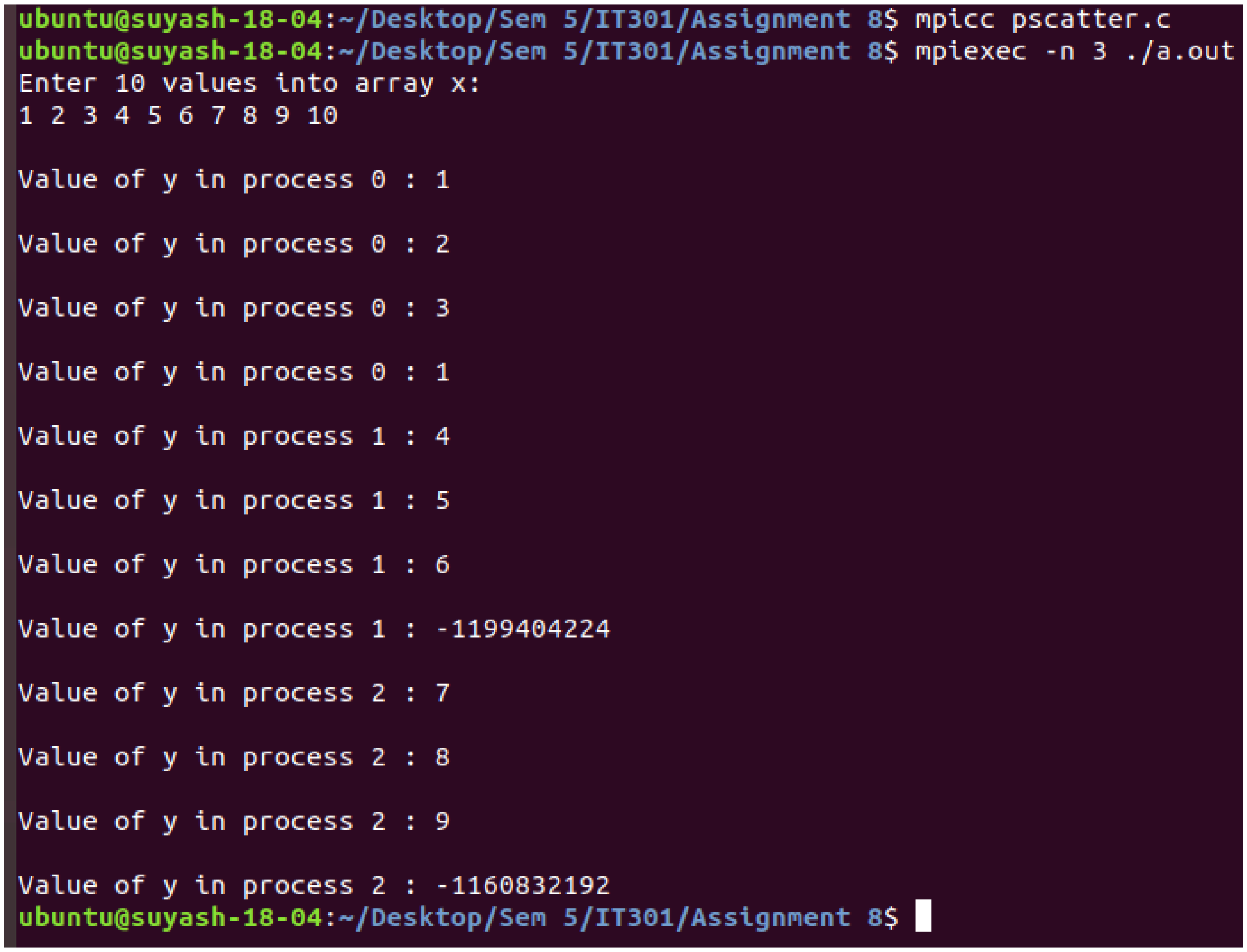
```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpicc scatter.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 4 ./a.out
Enter 8 values into array x:
1 2 3 4 5 6 7 8

Value of y in process 0 : 1

Value of y in process 0 : 2

Value of y in process 1 : 3

Value of y in process 1 : 4

Value of y in process 2 : 5

Value of y in process 2 : 6

Value of y in process 3 : 7

Value of y in process 3 : 8
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$
```

Observation: Scatter() works opposite to gather(). Here, the array is split into parts and the chunks are distributed among the processes. In our case, the 8-element array is split into 4 processes, each of which receives 2 elements.

Q6. Demonstration of MPI_Scatter() with partial scatter.
Note: Program is hardcoded to work with 3 processes receiving three chunks from the array. Note down the difference between program 5 and program 6.

(continued...)

Output:



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpicc pscatter.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ mpiexec -n 3 ./a.out
Enter 10 values into array x:
1 2 3 4 5 6 7 8 9 10

Value of y in process 0 : 1

Value of y in process 0 : 2

Value of y in process 0 : 3

Value of y in process 0 : 1

Value of y in process 1 : 4

Value of y in process 1 : 5

Value of y in process 1 : 6

Value of y in process 1 : -1199404224

Value of y in process 2 : 7

Value of y in process 2 : 8

Value of y in process 2 : 9

Value of y in process 2 : -1160832192
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 8$ 
```

Observation: This program is different from problem 5 as the scatter chunk size is 3 in this case. Also, as there are 10 elements in the original array, only 9 elements have been scattered because the distribution is only between 3 processes. The fourth component of the array y[3] gets garbage value as only 3 elements are scattered to the processes.

THANK YOU