**Assignment-6**

**Course**: High Performance Computing Lab

**PRN**: 22510034

**Name**: Om Kulkarni

**Batch**: B2

**Title:** Installation of MPI & Implementation of basic functions of MPI

MPI Installation Steps:

For Linux or Windows Subsystem for Linux

**Step-1:**

1. First, update the apt package to fetch the recent updates

sudo apt update

2. Run following command on terminal to install mpi from openmp

sudo apt install openmpi-bin openmpi-common libopenmpi-dev

3. Include in the Cpp.default.includePath in code editor to ensure it is detected:

/usr/lib/x86\_64-linux-gnu/openmpi/include

**Step-2**:   
To verify the installation, run the following command:  
 mpicc -show

**Step-3:**

To compile and run the program

* mpicc input\_file\_name.c -o output\_file\_name
* mpirun -np ”no\_of\_processors” ./hello\_world

**Problem Statement 1:**

Implement a simple Hello World program by setting the number of processes equal to 10

**SourceCode:**

#include<mpi.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

// initializing mpi environment

MPI\_Init(&argc, &argv);

// total number of processes

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

printf("Hello world from processor rank %d out of %d processors\n", world\_rank, world\_size);

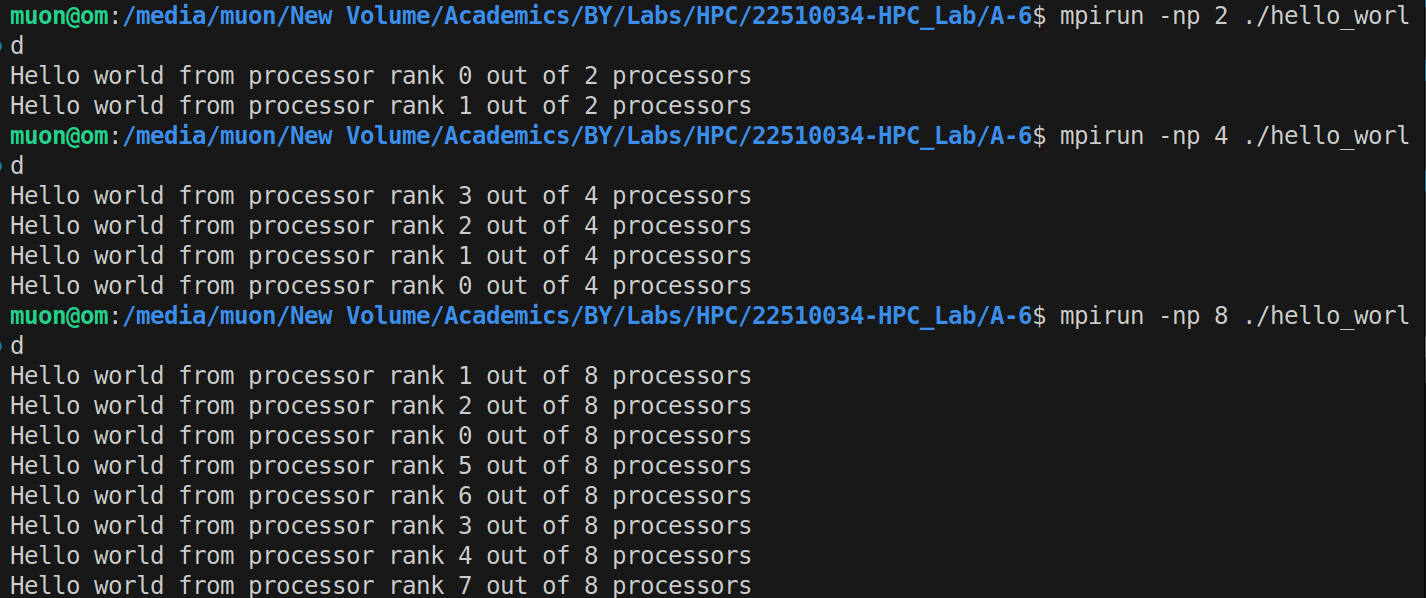
MPI\_Finalize();

return 0;

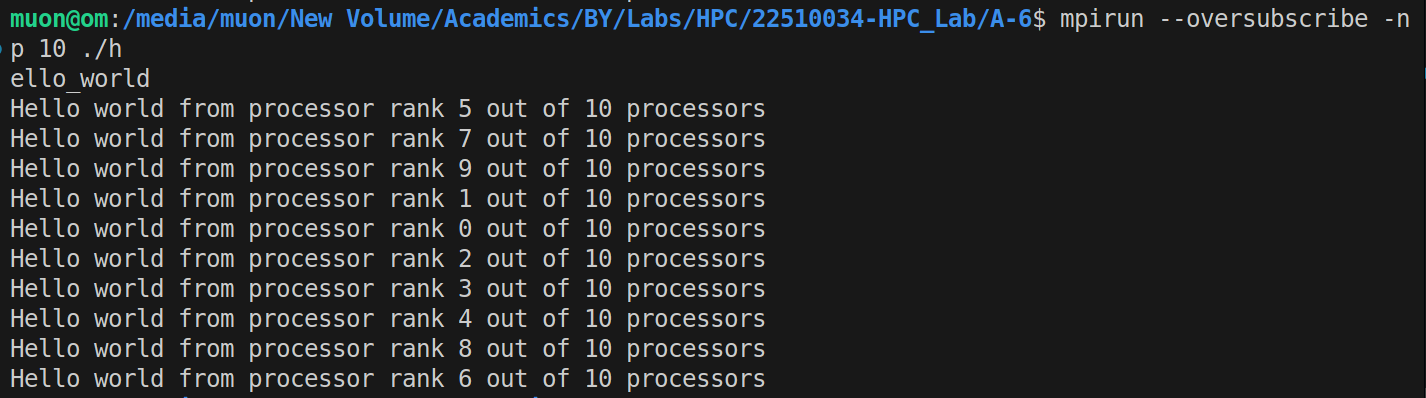
}

**Output:**

Up to those cores that my system supports



For more than 8 cores, we have to pass on the - - oversubscribe flag



**Analysis:**

MPI\_Init():

Initializes the MPI environment

MPI\_Comm\_size():  
Gets the total number of processes

MPI\_Comm\_rank():

Gets the current process’s rank (ID)

Each process runs the same code, but has a different rank

MPI\_Finalize():  
Cleans up the MPI environment

**Problem Statement 2:**

Implement a program to display the rank and the communicator group of five processes

**SourceCode:**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

MPI\_Init(&argc, &argv);

int world\_size, world\_rank;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

// associating a group with MPI\_COMM\_WORLD

MPI\_Group world\_group;

MPI\_Comm\_group(MPI\_COMM\_WORLD, &world\_group);

int group\_size, group\_rank;

MPI\_Group\_size(world\_group, &group\_size);

MPI\_Group\_rank(world\_group, &group\_rank);

printf("Process %d\n", world\_rank);

printf("World size: %d\n", world\_size);

printf("Group Rank: %d\n", group\_rank);

printf("Group Size: %d\n", group\_size);

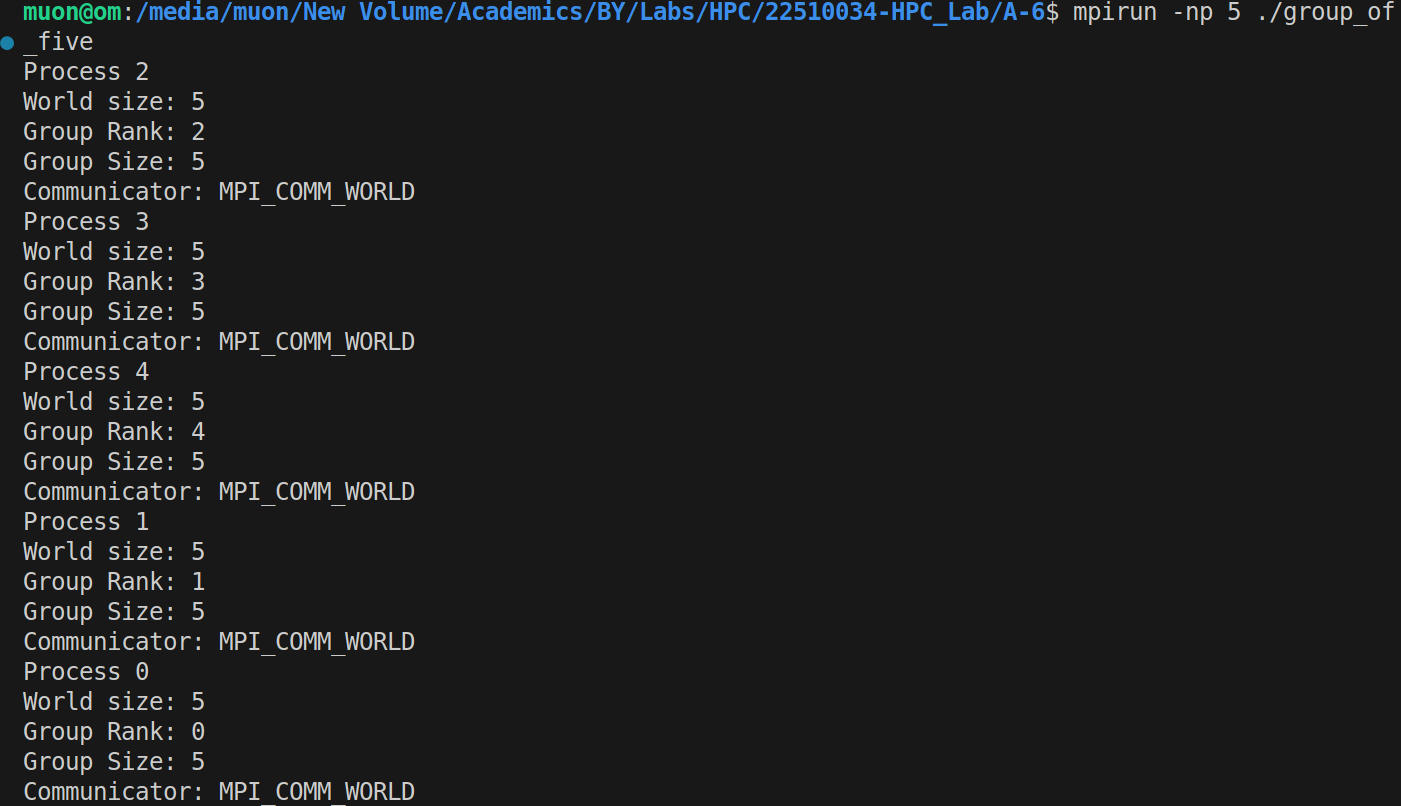
printf("Communicator: MPI\_COMM\_WORLD\n");

MPI\_Group\_free(&world\_group);

MPI\_Finalize();

return 0;

}

**Output:**  


**Analysis:**

MPI\_Comm\_group():

Gets the group associated with a communicator

MPI\_Group\_size():

Gets the number of processes in a group

MPI\_Group\_rank():  
Gets the rank of calling process in a group

MPI\_Group\_free():  
Frees the group object

Each process has both a communicator rank and a group rank.

Problem Statement-3:  
Implement an MPI program to give an example of Deadlock.

**SourceCode:**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int data = rank;

int received;

printf("Process %d: waiting to receive...\n", rank);

MPI\_Recv(&received, 1, MPI\_INT, 1-rank, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d: Now sending %d ...\n", rank, data);

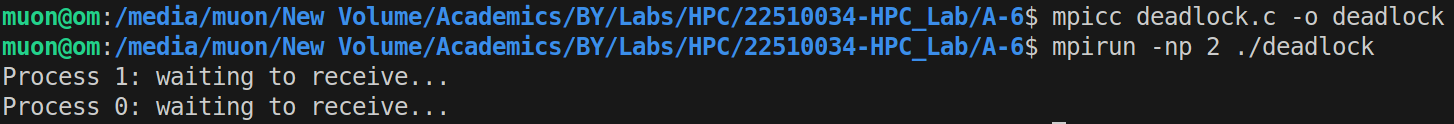
MPI\_Send(&data, 1, MPI\_INT, 1-rank, 0, MPI\_COMM\_WORLD);

printf("Process %d: Done\n", rank);

MPI\_Finalize();

return 0;

}

**Output:** 

The program simply hangs here as it’s a deadlock situation

**Analysis:**

Process 0 communicates with process 1 and vice versa

Process 0 waits to receive from process 1, and vice versa

Both processes are blocked at MPI\_Recv()

Neither can proceed to the MPI\_Send() line

Circular wait condition

MPI\_Recv():  
A blocking operation  
The calling process stops execution until it receives the expected message

Control doesn’t move to the next line until the receive completes

Synchronous nature:

Both sender and receiver must participate

**Problem Statement-4:**

Implement blocking MPI send & receive to demonstrate Nearest neighbor exchange of data in a ring topology.

**SourceCode:**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Ring topology: each process sends to next, receives from previous

int next = (rank + 1) % size;

int prev = (rank - 1 + size) % size;

int my\_data = rank \* 10;

int received\_data;

printf("Process %d: sending %d to Process %d\n", rank, my\_data, next);

// even ranks send first, odd ranks receive first

if (rank % 2 == 0) {

MPI\_Send(&my\_data, 1, MPI\_INT, next, 0, MPI\_COMM\_WORLD);

MPI\_Recv(&received\_data, 1, MPI\_INT, prev, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

} else {

MPI\_Recv(&received\_data, 1, MPI\_INT, prev, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Send(&my\_data, 1, MPI\_INT, next, 0, MPI\_COMM\_WORLD);

}

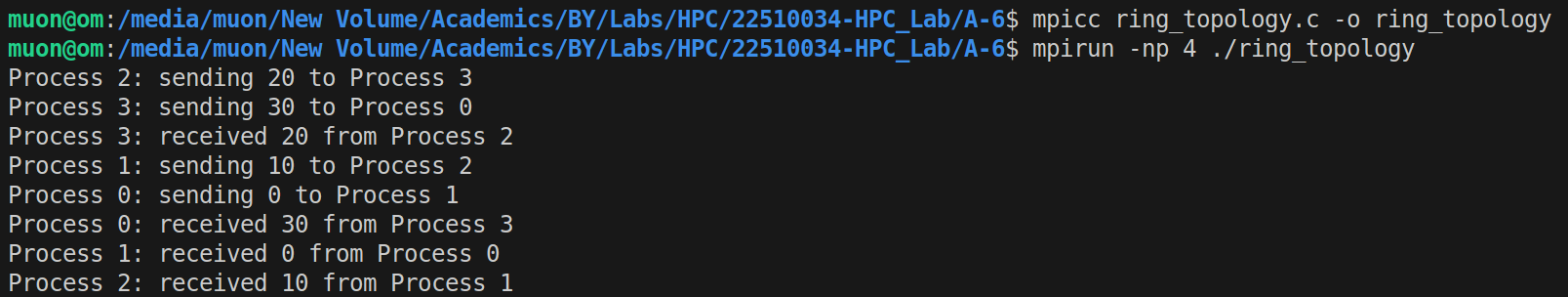
printf("Process %d: received %d from Process %d\n", rank, received\_data, prev);

MPI\_Finalize();

return 0;

}

**Output:**



**Analysis:**

Ring Topology: next = (rank + 1 ) % size and prev = (rank - 1+ size) % size

Nearest neighbour exchange:

Each process talks only to its immediate neighbours

Blocking send/receive:   
Using MPI\_Send(), MPI\_Receive

Even/Odd strategy to avoid the deadlock, by preventing all processes from waiting simultaneously

Modular arithmetic handles the wrap-around.

**Problem Statement-5:**

Write a MPI program to find the sum of all the elements of an array A of size

n. Elements of an array can be divided into two equals groups. The first [n/2]

elements are added by the first process, P0, and last [n/2] elements the by second process, P1. The two sums then are added to get the final result.

**SourceCode:**

**#include <mpi.h>**

**#include <stdio.h>**

**int main(int argc, char \*\*argv) {**

**MPI\_Init(&argc, &argv);**

**int rank;**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**int array[] = {1, 2, 3, 4, 5, 6, 7, 8};**

**int sum = 0;**

**if (rank == 0) {**

**// P0: Sum first half (0,1,2,3)**

**for (int i = 0; i < 4; i++) {**

**sum += array[i];**

**}**

**printf("P0 sum = %d\n", sum);**

**// Get P1's sum**

**int p1\_sum;**

**MPI\_Recv(&p1\_sum, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**printf("Total sum = %d\n", sum + p1\_sum);**

**} else {**

**// P1: Sum second half (4,5,6,7)**

**for (int i = 4; i < 8; i++) {**

**sum += array[i];**

**}**

**printf("P1 sum = %d\n", sum);**

**// Send to P0**

**MPI\_Send(&sum, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);**

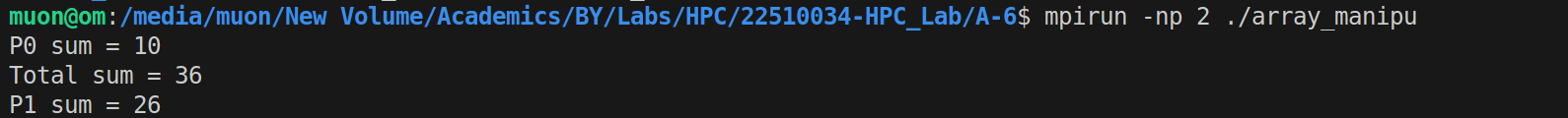
**}**

**MPI\_Finalize();**

**return 0;**

**}**

**Output:**

****

**Analysis:**

Array splitting into two equal halves

Process 1 sends its result to process 0

Process 0 combines both partial sums

P0: sums elements from 0 to (n/2)-1, receives P1’s sum and then calculates the final result:

P1: sums elements from n/2 to n-1, sends sum to P0

This is an example of point-to-point communication