**Class:** Final Year (Computer Science and Engineering)

**Year:** 2022-23 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 6**

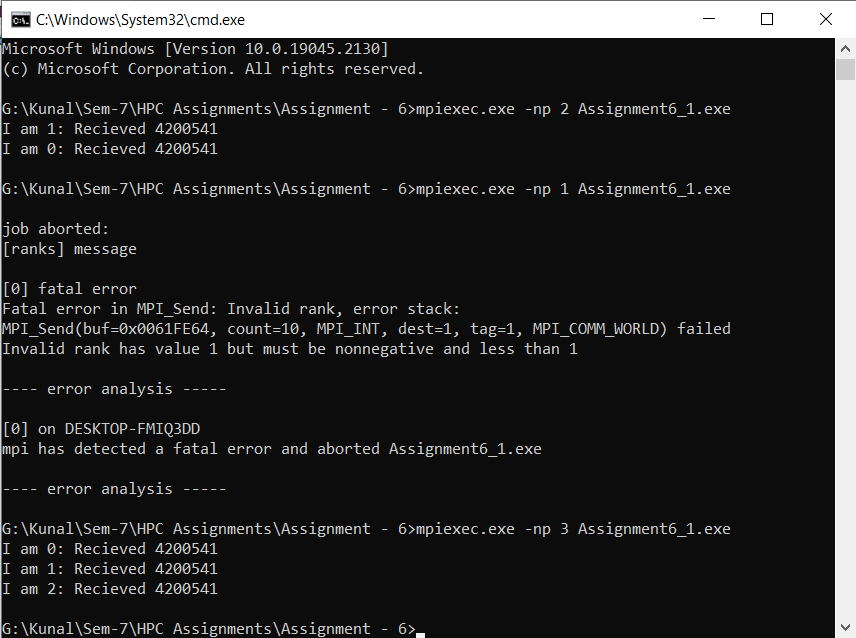
**Exam Seat No: 2019BTECS00064**

**Name – Kunal Santosh Kadam**

**Problem Statement 1:**

Implement a MPI program to give an example of Deadlock.

**Screenshot #:**



**Information #:**

#include <stdlib.h>

#include <stdio.h>

#include <mpi.h>

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

{

//Initialize the MPI environment

MPI\_Init(NULL,NULL);

//Get the rank of process

int rank;

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

int a[10],b[10];

MPI\_Status status;

if(rank == 0)

{

MPI\_Send(a, 10, MPI\_INT, 1, 1, MPI\_COMM\_WORLD);

MPI\_Send(b, 10, MPI\_INT, 1, 2, MPI\_COMM\_WORLD);

}

else if(rank == 1)

{

MPI\_Recv(b, 10, MPI\_INT, 0, 2, MPI\_COMM\_WORLD, &status);

MPI\_Recv(a, 10, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, &status);

}

printf("I am %d: Recieved %d\n", rank, b[0]);

//Finalize the MPI environment

MPI\_Finalize();

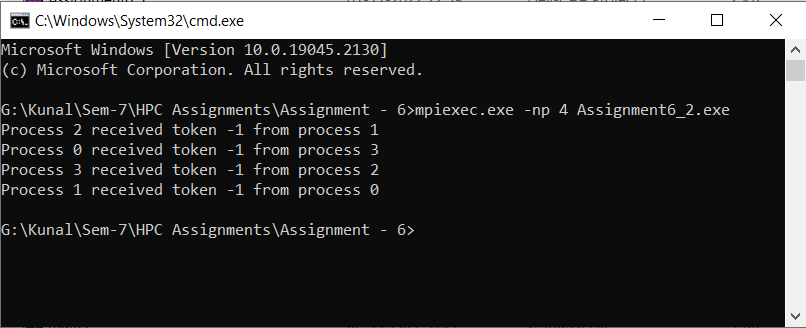
return 0;

}

**Problem Statement 2:**

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

**Screenshot #:**



**Information #:**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

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

{

// Initialize the MPI environment

MPI\_Init(NULL, NULL);

// Find out rank, size

int world\_rank;

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

int world\_size;

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

int token;

// Receive from the lower process and send to the higher process. Take care

// of the special case when you are the first process to prevent deadlock.

if (world\_rank != 0)

{

MPI\_Recv(&token, 1, MPI\_INT, world\_rank - 1, 0, MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n", world\_rank, token,

world\_rank - 1);

}

else

{

// Set the token's value if you are process 0

token = -1;

}

MPI\_Send(&token, 1, MPI\_INT, (world\_rank + 1) % world\_size, 0,

MPI\_COMM\_WORLD);

// Now process 0 can receive from the last process. This makes sure that at

// least one MPI\_Send is initialized before all MPI\_Recvs (again, to prevent

// deadlock)

if (world\_rank == 0)

{

MPI\_Recv(&token, 1, MPI\_INT, world\_size - 1, 0, MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n", world\_rank, token,

world\_size - 1);

}

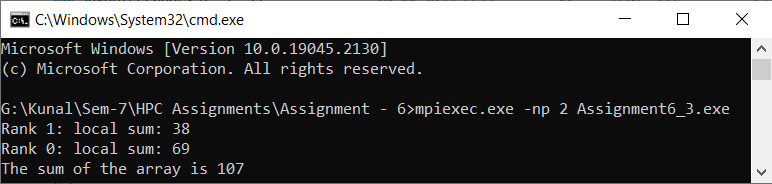
MPI\_Finalize();

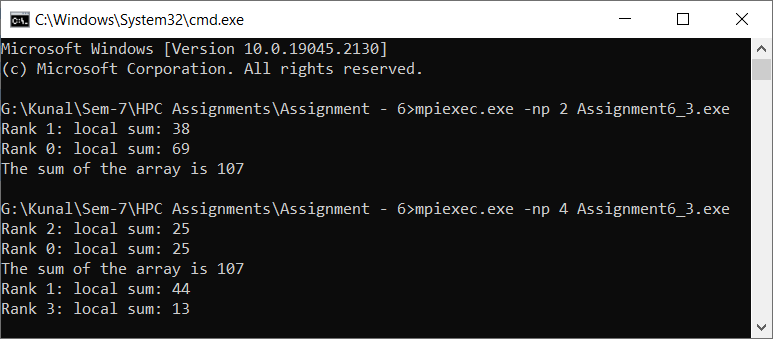
}

**Problem Statement 3:**

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.

**Screenshot #:**





**Information #:**

#include<stdio.h>

#include<mpi.h>

#define arr\_size 15

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

int i,j;

int rank, size;

MPI\_Init(&argc, &argv);

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

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

//Code that will execute inside process 0 or rank 0

if(rank == 0){

int arr[]= {12,4,6,3,21,15,3,5,7,8,9,1,5,3,5};

int global\_sum = 0, local\_sum = 0, recv\_local\_sum;

//If the array size is perfectly divisible by number of process.

if(arr\_size%size == 0){

int array\_element\_per\_process = arr\_size/size;

int sub\_arr[array\_element\_per\_process];

for(i=1; i<size; i++){

//Copying the sub array

for(j=0; j<array\_element\_per\_process;j++){

sub\_arr[j] = arr[i\*array\_element\_per\_process+j];

}

//Sending array chunk of equal size to all the process.

MPI\_Send(sub\_arr, array\_element\_per\_process, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

MPI\_Send(&array\_element\_per\_process, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

}

//Calculating the local sum of rank 0 itself

for(j=0; j<array\_element\_per\_process; j++){

local\_sum += arr[j];

}

printf("Rank %d: local sum: %d\n", rank, local\_sum);

global\_sum += local\_sum;

//When the array size is not perfectly divisible by number of process.

}else{

int array\_element\_per\_process = arr\_size/size + 1;

int sub\_arr[array\_element\_per\_process];

for(i=1; i<size; i++){

if(i == size - 1){

//last sub array will have the size less than other process array size

int total\_array\_size\_of\_last\_process = arr\_size - array\_element\_per\_process \* i;

for(j=0; j< total\_array\_size\_of\_last\_process; j++){

sub\_arr[j] = arr[i\*array\_element\_per\_process+j];

}

MPI\_Send(&sub\_arr, total\_array\_size\_of\_last\_process, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

MPI\_Send(&total\_array\_size\_of\_last\_process, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

}else{

//Copying the sub array

for(j=0; j<array\_element\_per\_process;j++){

sub\_arr[j] = arr[i\*array\_element\_per\_process+j];

}

MPI\_Send(&sub\_arr, array\_element\_per\_process, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

MPI\_Send(&array\_element\_per\_process, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

}

}

//Calculating the local sum of rank 0 itself

for(j=0; j<array\_element\_per\_process; j++){

local\_sum += arr[j];

}

printf("Rank %d: local sum: %d\n", rank, local\_sum);

global\_sum += local\_sum;

}

//calculating the global sum of the array

//Receving the local sum from the other process and updating the global sum

for(i=1; i<size; i++){

MPI\_Recv(&recv\_local\_sum, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

global\_sum += recv\_local\_sum;

}

//Printing the output

printf("The sum of the array is %d\n", global\_sum);

//Code that will get executed inside other than process 0 or rank 0.

}else{

//The other process will receive the chunck of array

int array\_element\_per\_process = arr\_size/size + 1;

int recv\_sub\_arr[array\_element\_per\_process];

int recv\_array\_element\_per\_process, local\_sum = 0;

MPI\_Recv(recv\_sub\_arr, recv\_array\_element\_per\_process, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(&recv\_array\_element\_per\_process, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

//Calculating local sum for the sub array

for(j=0; j<recv\_array\_element\_per\_process; j++){

local\_sum += recv\_sub\_arr[j];

}

//Printing the local sum

printf("Rank %d: local sum: %d\n", rank, local\_sum);

//Sending back the local sum to the rank 0 or process 0.

MPI\_Send(&local\_sum, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}