**Final Year B. Tech., Sem VII 2022-23**

**High Performance Computing Lab**

**PRN: 2020BTECS00206**

**Full Name: SAYALI YOGESH DESAI**

**Batch: B4**

**Assignment No. 6**

**Q1: Implement a MPI program to give an example of Deadlock.**

* **Code:**

#include "mpi.h"

#include <math.h>

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

MPI\_Status status;

int num;

MPI\_Init(&argc, &argv);

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

double d = 100.0;

int tag = 1;

if (num == 0) {

// synchronous Send

MPI\_Ssend(&d, 1, MPI\_DOUBLE, 1, tag,

MPI\_COMM\_WORLD);

MPI\_Recv(&d, 1, MPI\_DOUBLE, 1, tag,

MPI\_COMM\_WORLD, &status);

} else {

// Synchronous Send

MPI\_Ssend(&d, 1, MPI\_DOUBLE, 1, tag,

MPI\_COMM\_WORLD);

MPI\_Recv(&d, 1, MPI\_DOUBLE, 1, tag,

MPI\_COMM\_WORLD, &status);

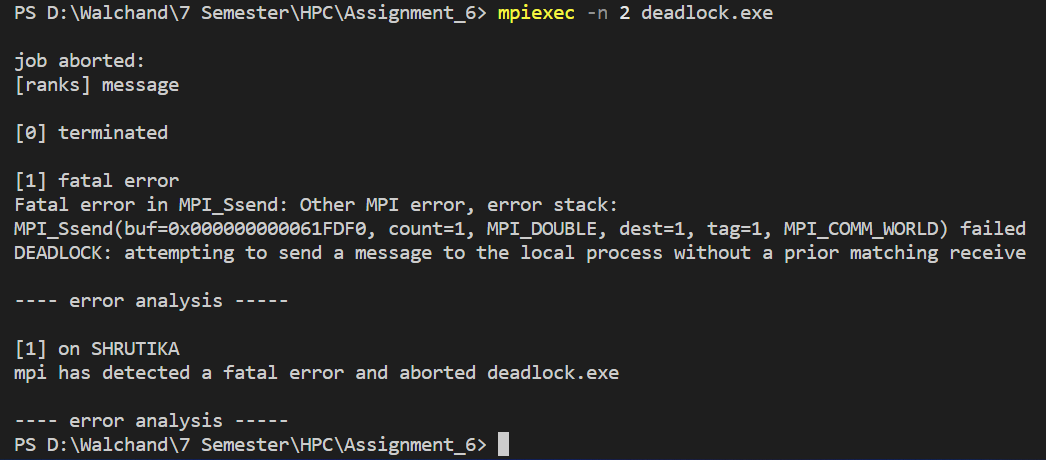
}

MPI\_Finalize();

return 0;

}

* **Output:**

****

**Q2. Implement blocking MPI send & receive to demonstrate Nearest neighbour exchange of data in a ring topology.**

* **Code:**

#include "mpi.h"

#include <stdio.h>

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

int rank;

int num;

MPI\_Init(&argc, &argv);

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

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

MPI\_Status status;

double d = 483048.0;

int tag = 1;

// calculating next rank

int rank\_next = (rank + 1) % num;

// prev process rank

int rank\_prev = rank == 0 ? num - 1 : rank - 1;

if (num % 2 == 0) {

printf("Rank %d: sending to %d\n", rank,

rank\_next);

MPI\_Send(&d, 1, MPI\_DOUBLE, rank\_next, tag,

MPI\_COMM\_WORLD);

printf("Rank %d: receiving from %d\n", rank,

rank\_prev);

MPI\_Recv(&d, 1, MPI\_DOUBLE, rank\_prev, tag,

MPI\_COMM\_WORLD, &status);

} else {

printf("Rank %d: receiving from %d\n", rank,

rank\_prev);

MPI\_Recv(&d, 1, MPI\_DOUBLE, rank\_prev, tag,

MPI\_COMM\_WORLD, &status);

printf("Rank %d: sending to %d\n", rank,

rank\_next);

MPI\_Send(&d, 1, MPI\_DOUBLE, rank\_next, tag,

MPI\_COMM\_WORLD);

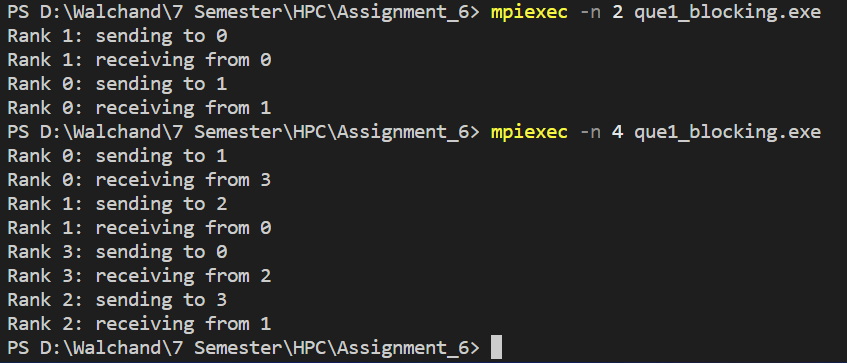
}

MPI\_Finalize();

return 0;

}

* **Output:**

****

**Q3. 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.**

* **Code:**

#include "mpi.h"

#include <stdio.h>

#define localSize 1000

int local[1000]; // to store the subarray data comming from process 0;

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

{

int rank;

int num;

int n = 10;

int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int per\_process, elements\_received;

MPI\_Init(&argc, &argv);

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

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

MPI\_Status status;

// process with rank 0 will divide data among all processes and add partial sums to get final sum

if (rank == 0)

{

int index, i;

per\_process = n / num;

if (num > 1) // if more than 1 processes available

{

// divide array data among processes

for (i = 1; i < num - 1; i++)

{

// calculating first index of subarray that need to be send to ith process

index = i \* per\_process;

// send no of elements and subarray of that lenght to each process

MPI\_Send(&per\_process, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

MPI\_Send(&arr[index], per\_process, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

}

// for last process send all remaining elements

index = i \* per\_process;

int ele\_left = n - index;

MPI\_Send(&ele\_left, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

MPI\_Send(&arr[index], ele\_left, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

}

// add numbers on process with rank 0

int sum = 0;

for (int i = 0; i < per\_process; i++)

{

sum += arr[i];

}

// add all partial sums from all processes

int tmp;

for (int i = 1; i < num; i++)

{

MPI\_Recv(&tmp, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &status);

int sender = status.MPI\_SOURCE;

sum += tmp;

}

printf("Sum of array = %d\n", sum);

}

else // if rank of process is not 0, then receive elements and calculate partial sums

{

// receive no of elements and elements form process 0 and store them on local array

MPI\_Recv(&elements\_received, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&local, elements\_received, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

// calculate partial local sum

int partial\_sum = 0;

for (int i = 0; i < elements\_received; i++)

{

partial\_sum += local[i];

}

// send calculated partial sum to process with rank 0

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

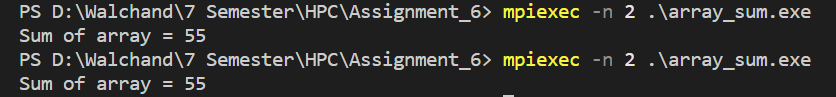
}

MPI\_Finalize();

return 0;

}

* **Output:**

****

**Github Link:** [**https://github.com/SayaliDesai4/HPC-Practicals**](https://github.com/SayaliDesai4/HPC-Practicals)