Walchand College of Engineering, Sangli

Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

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

Course: High Performance Computing Lab

Practical No. 6

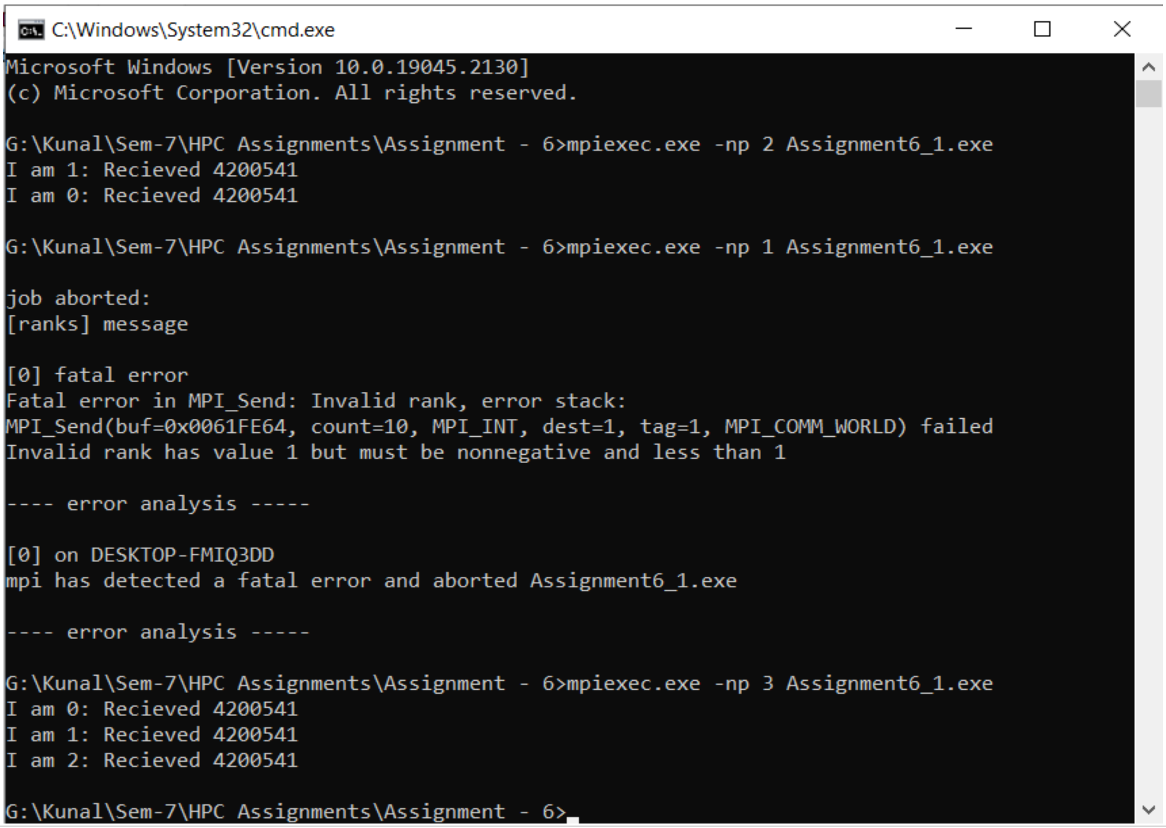
Exam Seat No: 2019BTECS00064

Name – Kunal Santosh Kadam

Problem Statement 1:

Implement a MPI program to give an example of Deadlock.

Screenshot #:



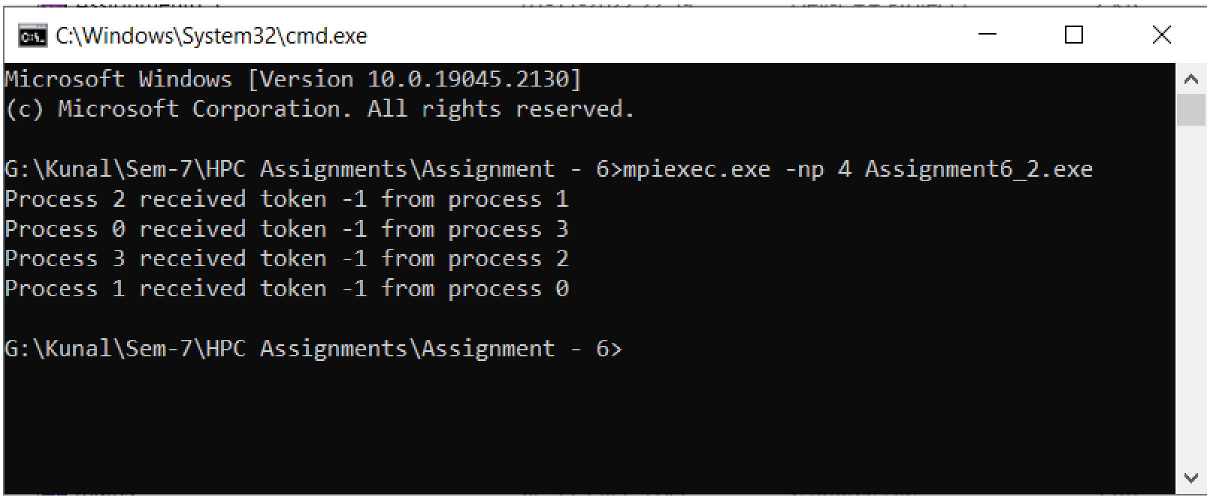
Information #:

```
#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 #:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2130]
(c) Microsoft Corporation. All rights reserved.

G:\Kunal\Sem-7\HPC Assignments\Assignment - 6>mpiexec.exe -np 2 Assignment6_3.exe
Rank 1: local sum: 38
Rank 0: local sum: 69
The sum of the array is 107
```

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2130]
(c) Microsoft Corporation. All rights reserved.

G:\Kunal\Sem-7\HPC Assignments\Assignment - 6>mpiexec.exe -np 2 Assignment6_3.exe
Rank 1: local sum: 38
Rank 0: local sum: 69
The sum of the array is 107

G:\Kunal\Sem-7\HPC Assignments\Assignment - 6>mpiexec.exe -np 4 Assignment6_3.exe
Rank 2: local sum: 25
Rank 0: local sum: 25
The sum of the array is 107
Rank 1: local sum: 44
Rank 3: local sum: 13
```

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;
}
```