Class: Final Year (Computer Science and Engineering)

Year: 2023-24 **Semester:** 1

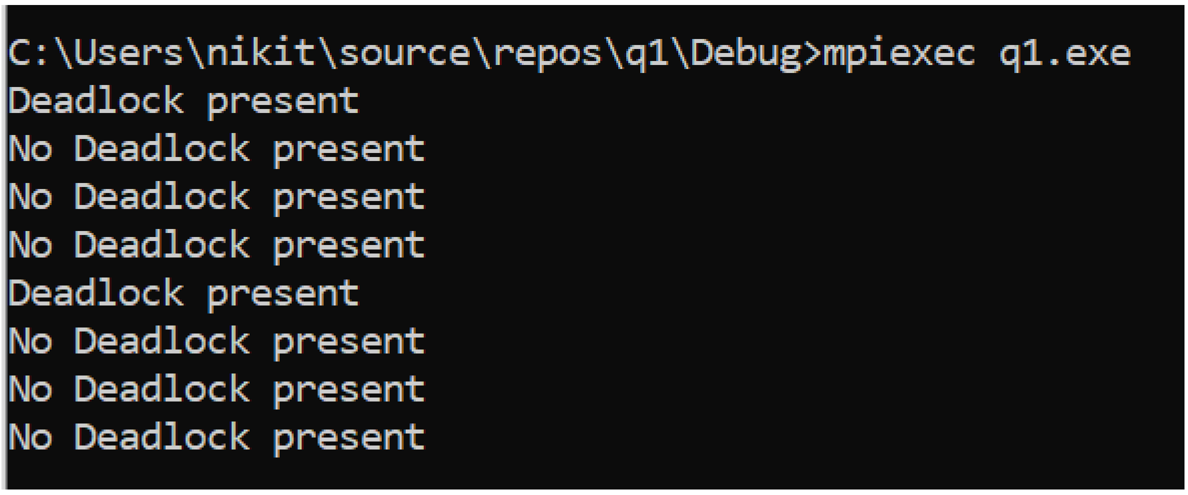
Course: High Performance Computing Lab

Practical No.10

PRN No: 2020BTECS00041
Name: Nikita Shivchandra Khot

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

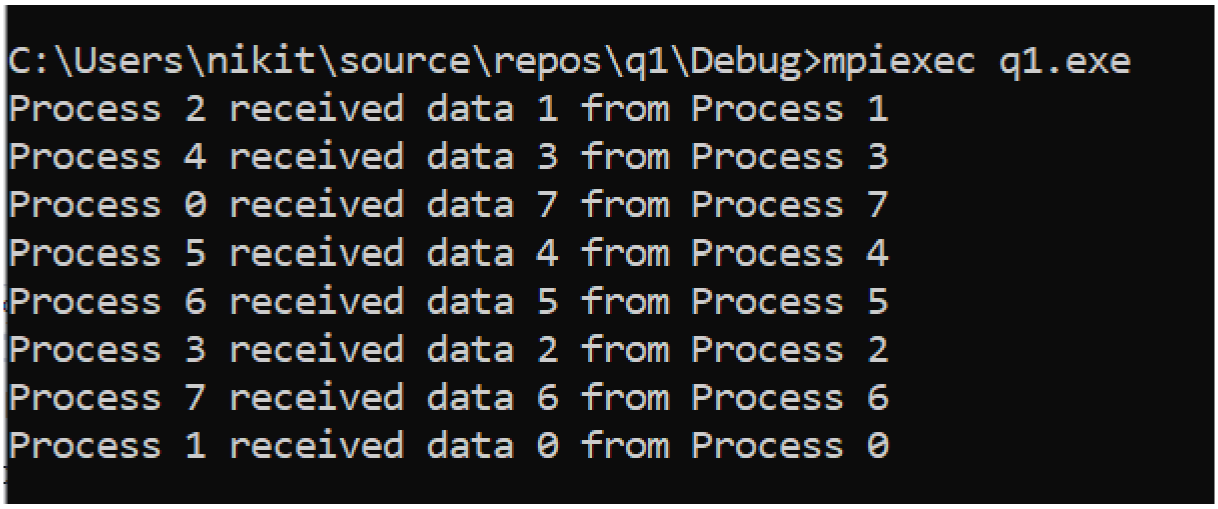
```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2)
    {
        printf("This example requires at least 2 processes.\n");
        MPI_Finalize();
        return 1;
    }
    bool ans = false;

    if (rank == 0)
    {
        // Process 0 sends a message but does not receive it.
        int data = 42;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        // Process 0 is now waiting for message from Process 1, which will never
arrive.
        ans = true;
    }
    else if (rank == 1) {
        // Process 1 sends message but does not receive it.
        int data = 99;
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        // Process 1 is now waiting for message from Process 0, which will never
arrive.
        ans = true;
    }
    // Both processes are now in state of deadlock and will not proceed further.
    if (ans) {
        printf("Deadlock present\n");
    }
    else {
        printf("No Deadlock present\n");
    }
    MPI_Finalize();
    return 0;
}
```



```
C:\Users\nikit\source\repos\q1\Debug>mpiexec q1.exe
Deadlock present
No Deadlock present
No Deadlock present
No Deadlock present
Deadlock present
No Deadlock present
No Deadlock present
No Deadlock present
```

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

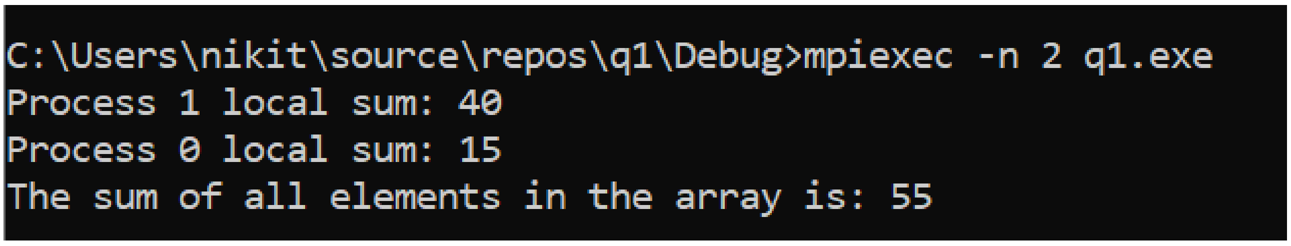
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int rank, size;
    int data, received_data;
    MPI_Status status;

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        printf("This program requires at least 2 processes.\n");
        MPI_Finalize();
        return 1;
    }

    // Data to send (use the rank of the process)
    data = rank;

    // Neighbors in the ring topology
    int left_neighbor = (rank - 1 + size) % size;
    int right_neighbor = (rank + 1) % size;

    // Perform nearest neighbor data exchange in a ring
    MPI_Send(&data, 1, MPI_INT, right_neighbor, 0, MPI_COMM_WORLD);
    MPI_Recv(&received_data, 1, MPI_INT, left_neighbor, 0, MPI_COMM_WORLD, &status);

    // Print the result
    printf("Process %d received data %d from Process %d\n", rank, received_data,
left_neighbor);

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```



```
C:\Users\nikit\source\repos\q1\Debug>mpiexec q1.exe
Process 2 received data 1 from Process 1
Process 4 received data 3 from Process 3
Process 0 received data 7 from Process 7
Process 5 received data 4 from Process 4
Process 6 received data 5 from Process 5
Process 3 received data 2 from Process 2
Process 7 received data 6 from Process 6
Process 1 received data 0 from Process 0
```

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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int n = 10; // The size of the array
    int* A = (int*)malloc(n * sizeof(int));
    // Initialize the array with some data (for example, 1 to n)
    for (int i = 0; i < n; i++) {
        A[i] = i + 1;
    }
    int local_sum = 0;
    int global_sum = 0;
    // Determine the number of elements to be processed by each process
    int local_n = n / 2;
    // Calculate the sum of elements for each process
    for (int i = rank * local_n; i < (rank + 1) * local_n; i++) {
        local_sum += A[i];
    }

    // Use MPI_Reduce to calculate the global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    printf("Process %d local sum: %d\n", rank, local_sum);

    if (rank == 0) {
        printf("The sum of all elements in the array is: %d\n", global_sum);
    }

    free(A);
    MPI_Finalize();
    return 0;
}
```



```
C:\Users\nikit\source\repos\q1\Debug>mpiexec -n 2 q1.exe
Process 1 local sum: 40
Process 0 local sum: 15
The sum of all elements in the array is: 55
```