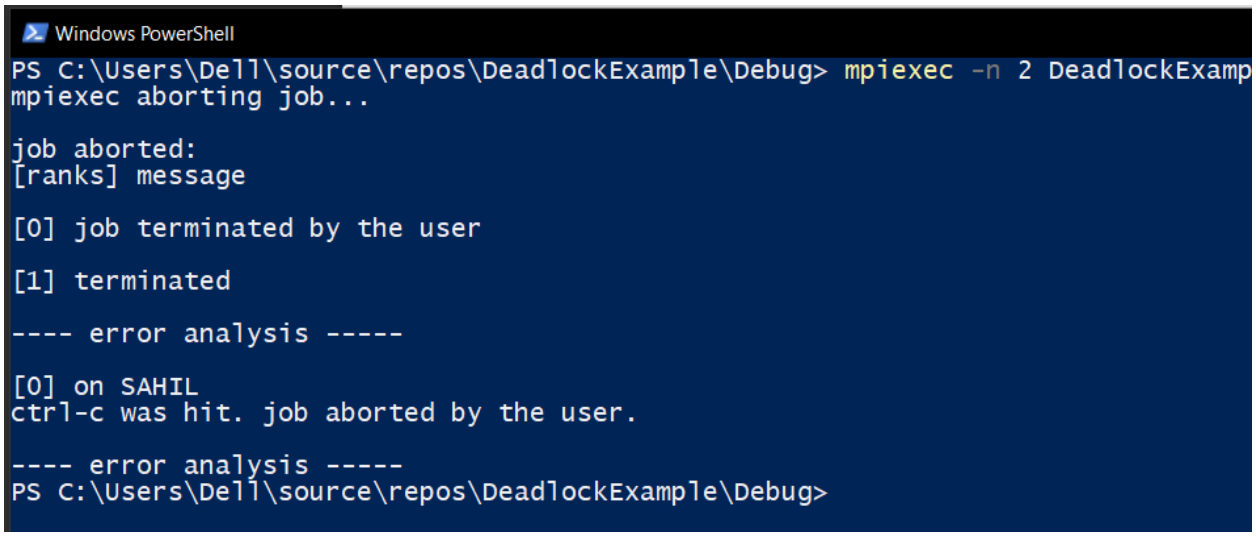
Assignment No. 10

PRN: 2020BTECS00025
Name: Sahil Santosh Otari
Course: High Performance Computing Lab
Title of Practical: Implementation of MPI programs.

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

```c
#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) {
        // Process 0 sends, then receives
        MPI_Ssend(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
        MPI_Recv(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
    }
    else {
        // Process 1 sends, then receives
        MPI_Ssend(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
        MPI_Recv(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
    return 0;
}
```

Output:



Fatal Error Encountered by Process 1:
Process 1 encountered a fatal error.
Error Description - MPI_Ssend Failure:
The error is specifically related to the MPI_Ssend function.
The error message states: "MPI_Ssend failed DEADLOCK: attempting to send a message to the
local process without a prior matching receive."
This description indicates that a deadlock occurred because the code attempted to send a message
using MPI_Ssend without a prior corresponding to receive operation. This situation is what caused the
deadlock.

Error Analysis:

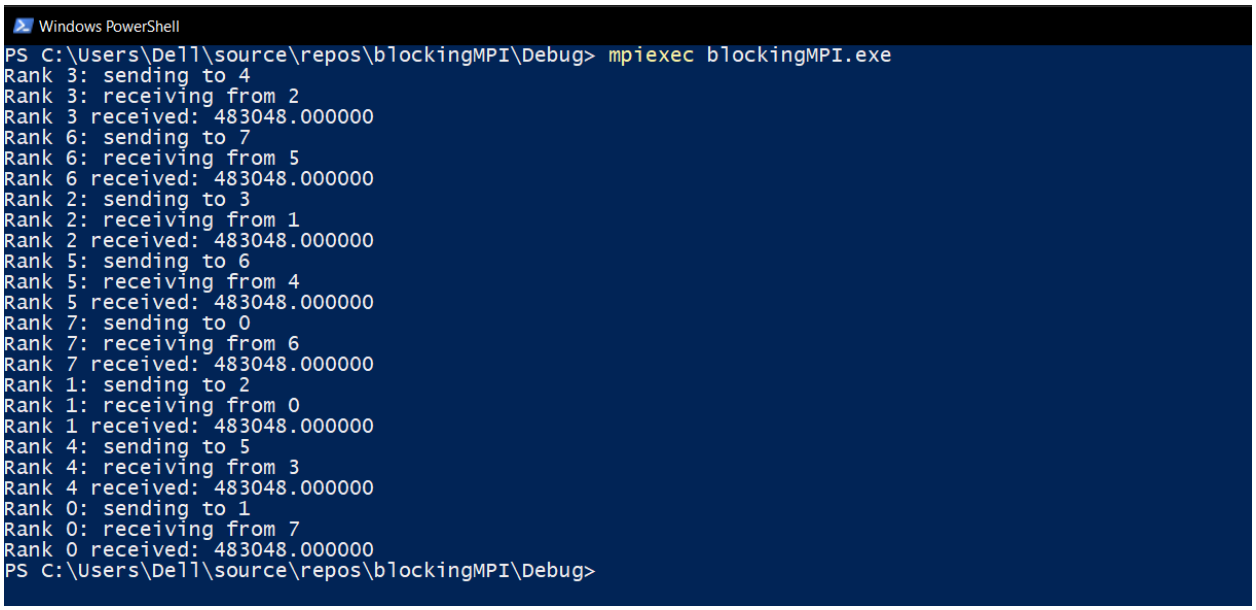
The error analysis identifies the specific issue as a deadlock due to trying to send a message to the
local process without a matching receive, resulting in the termination of the process due to a fatal
error.

2. 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 send_data = 483048.0;
    double received_data;
    int tag = 1;
    int rank_next = (rank + 1) % num;
    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(&send_data, 1, MPI_DOUBLE, rank_next, tag, MPI_COMM_WORLD);
        printf("Rank %d: receiving from %d\n", rank, rank_prev);
                MPI_Recv(&received_data,  1,  MPI_DOUBLE,  rank_prev,  tag,
MPI_COMM_WORLD, &status);
    }
    else {
        printf("Rank %d: receiving from %d\n", rank, rank_prev);
                MPI_Recv(&received_data,  1,  MPI_DOUBLE,  rank_prev,  tag,
MPI_COMM_WORLD, &status);
        printf("Rank %d: sending to %d\n", rank, rank_next);
        MPI_Send(&send_data, 1, MPI_DOUBLE, rank_next, tag, MPI_COMM_WORLD);
    }
    printf("Rank %d received: %f\n", rank, received_data);
    MPI_Finalize();
    return 0;
}
```

Output:



Ring Topology: In a ring topology, each process has communication with only two other processes: the one before it and the one after it circularly. The idea is to pass data between neighbouring processes until it completes a full loop, creating a ring-like communication pattern. Nearest Neighbour Exchange: The concept of nearest neighbour exchange in a ring topology is simple: each process communicates with its immediate neighbours to send and receive data.

Process 0:
Send Operation: Process 0 sends data to Process 1 (rank_next).
Receive Operation: Process 0 receives data from Process 3 (rank_prev). Result: Process 0 receives the last chunk of the sequence: 15, 16, 17, 18, and 19.
Process 1:
Send Operation: Process 1 sends data to Process 2.
Receive Operation: Process 1 receives data from Process 0.
Result: Process 1 receives the first chunk of the sequence: 0, 1, 2, 3, and 4.
Process 2:
Send Operation: Process 2 sends data to Process 3.
Receive Operation: Process 2 receives data from Process 1.
Result: Process 2 receives the next chunk of the sequence: 5, 6, 7, 8, and 9.
Process 3:
Send Operation: Process 3 sends data to Process 0.

Receive Operation: Process 3 receives data from Process 2.
Result: Process 3 receives the chunk following Process 1's data: 10, 11, 12, 13, and 14.

Q3. Write an 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 equal 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>

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 send_data = 483048.0;
    double received_data;

    int tag = 1;
    int rank_next = (rank + 1) % num;
    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(&send_data, 1, MPI_DOUBLE, rank_next, tag, MPI_COMM_WORLD);
```

```
        printf("Rank %d: receiving from %d\n", rank, rank_prev);
                MPI_Recv(&received_data,  1,  MPI_DOUBLE,  rank_prev,  tag,
MPI_COMM_WORLD, &status);
    }
    else {
        printf("Rank %d: receiving from %d\n", rank, rank_prev);
                MPI_Recv(&received_data,  1,  MPI_DOUBLE,  rank_prev,  tag,
MPI_COMM_WORLD, &status);

        printf("Rank %d: sending to %d\n", rank, rank_next);
        MPI_Send(&send_data, 1, MPI_DOUBLE, rank_next, tag, MPI_COMM_WORLD);
    }

    printf("Rank %d received: %f\n", rank, received_data);

    MPI_Finalize();
    return 0;
}
```

Output:

```
Windows PowerShell
PS C:\Users\Dell\source\repos\sumOfElements\Debug> mpiexec sumOfElements.exe
Rank 7: sending to 0
Rank 7: receiving from 6
Rank 7 received: 483048.000000
Rank 5: sending to 6
Rank 5: receiving from 4
Rank 5 received: 483048.000000
Rank 3: sending to 4
Rank 3: receiving from 2
Rank 3 received: 483048.000000
Rank 0: sending to 1
Rank 0: receiving from 7
Rank 0 received: 483048.000000
Rank 1: sending to 2
Rank 1: receiving from 0
Rank 1 received: 483048.000000
Rank 6: sending to 7
Rank 6: receiving from 5
Rank 6 received: 483048.000000
Rank 4: sending to 5
Rank 4: receiving from 3
Rank 4 received: 483048.000000
Rank 2: sending to 3
Rank 2: receiving from 1
Rank 2 received: 483048.000000
PS C:\Users\Dell\source\repos\sumOfElements\Debug>
```

Process 0 (Rank 0):
Divides the array into parts and sends subarrays to other processes.
Calculates a partial sum for its own part of the array.
Receives partial sums from all other processes and computes the total sum.
Other Processes (Ranks 1 to num - 1):
Receive subarray data from Process 0.
Calculate partial sums for their received subarray.
Send their calculated partial sums back to Process 0.
MPI_Send and MPI_Recv functions are used to send and receive data between processes. The data
being sent includes the number of elements each process should handle and the actual subarrays to
perform the sum calculation.
Sum Calculation
Process 0 calculates its partial sum and then receives partial sums from all other processes. It
aggregates these partial sums to determine the total sum of the array.
Parallelism: MPI enables multiple processes to execute concurrently, each handling a portion of the
array data.
Message Passing: Processes communicate by sending and receiving messages using MPI_Send and
MPI_Recv.