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

**Course:** High Performance Computing Lab

## Practical No.8

PRN : 21510074

Name : Yash Nawale

Batch : B4

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

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define MESSAGE\_SIZE 10000000

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

    MPI\_Init(&argc, &argv);

    int world\_rank;

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

    int\* message = (int\*)malloc(MESSAGE\_SIZE \* sizeof(int));

    if (world\_rank == 0) {

        MPI\_Send(message, MESSAGE\_SIZE, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

        printf("Process 0 sent message to Process 1\n");

        MPI\_Recv(message, MESSAGE\_SIZE, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        printf("Process 0 received message from Process 1\n");

    } else if (world\_rank == 1) {

        MPI\_Send(message, MESSAGE\_SIZE, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

        printf("Process 1 sent message to Process 0\n");

        MPI\_Recv(message, MESSAGE\_SIZE, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        printf("Process 1 received message from Process 0\n");

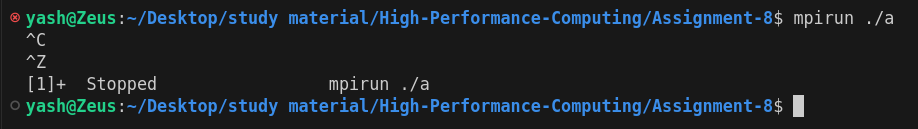
    }

    free(message);

    MPI\_Finalize();

    return 0;

}



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

#include <mpi.h>

#include <stdio.h>

#include <string.h>

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

{

    int rank, size;

    char send\_data[100];

    char recv\_data[100];

    int dest, source;

    MPI\_Init(&argc, &argv);

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

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

    dest = (rank + 1) % size;

    source = (rank - 1 + size) % size;

    // Sprintf Simiar to printf par stores string instead of printing

    sprintf(send\_data, "Hello %d", dest);

    MPI\_Send(send\_data, strlen(send\_data) + 1, MPI\_CHAR, dest, 0, MPI\_COMM\_WORLD);

    MPI\_Recv(recv\_data, 100, MPI\_CHAR, source, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

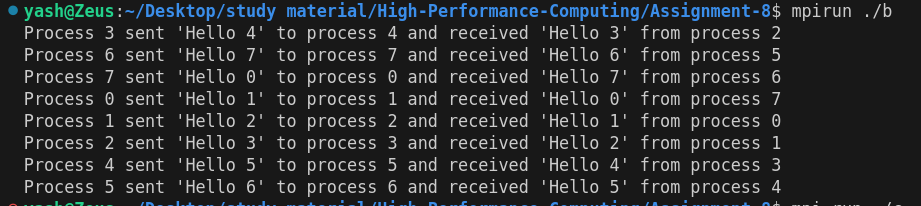
    printf("Process %d sent '%s' to process %d and received '%s' from process %d\n",

           rank, send\_data, dest, recv\_data, source);

    MPI\_Finalize();

    return 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.**

#include <mpi.h>

#include <stdio.h>

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

    MPI\_Init(&argc, &argv);

    int world\_rank, world\_size;

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

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

    int n = 8;

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

    int local\_sum = 0;

    int half = n / 2;

    if (world\_rank == 0) {

        for (int i = 0; i < half; i++) {

            local\_sum += A[i];

        }

        printf("Process 0 partial sum: %d\n", local\_sum);

    }

    else if (world\_rank == 1) {

        for (int i = half; i < n; i++) {

            local\_sum += A[i];

        }

        printf("Process 1 partial sum: %d\n", local\_sum);

    }

    int total\_sum = 0;

    if (world\_rank == 0) {

        int other\_sum = 0;

        MPI\_Recv(&other\_sum, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        total\_sum = local\_sum + other\_sum;

        printf("Final sum calculated by Process 0: %d\n", total\_sum);

    } else if (world\_rank == 1) {

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

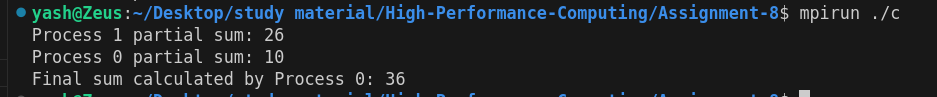
    }

    MPI\_Finalize();

    return 0;

}

Output :-



Github link :-

<https://github.com/YashNawale26/High-Performance-Computing>