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

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

**Course:** High Performance Computing Lab

**Practical No. 10**

**Name:** Sumit Narake

**PRN:**2020BTECS00023

**Batch :** B2

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

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

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

    int rank, size;

    int data;

    MPI\_Status status;

    MPI\_Init(&argc, &argv);

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

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

    if (size < 2) {

        fprintf(stderr, "This program requires at least two processes.\n");

        MPI\_Finalize();

        return 1;

    }

    if (rank == 0) {

        // Process 0 send data to Process 1

        data = 42;

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

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

        // Process 0 now waits for a message from Process 1

        MPI\_Recv(&data, 1, MPI\_INT, 1, 1, MPI\_COMM\_WORLD, &status);

    } else if (rank == 1) {

        // Process 1 send data to Process 0

        data = 99;

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

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

        // Process 1 now waits for a message from Process 0

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

    }

    MPI\_Finalize();

    return 0;

}

**Output:**

****

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

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

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

    int rank, size;

    int data;

    MPI\_Status status;

    MPI\_Init(&argc, &argv);

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

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

    if (size < 2) {

        fprintf(stderr, "This program requires at least two processes.\n");

        MPI\_Finalize();

        return 1;

    }

    data = rank;

    int left\_neighbor = (rank - 1 + size) % size;

    int right\_neighbor = (rank + 1) % size;

    // Send data to the right neighbor

    MPI\_Send(&data, 1, MPI\_INT, right\_neighbor, 0, MPI\_COMM\_WORLD);

    // Receive data from the left neighbor

    MPI\_Recv(&data, 1, MPI\_INT, left\_neighbor, 0, MPI\_COMM\_WORLD, &status);

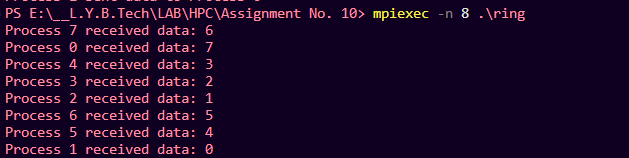
    printf("Process %d received data: %d\n", rank, data);

    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.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

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

    int rank, size, n, local\_n;

    int\* A = NULL;

    int local\_sum = 0, total\_sum = 0;

    MPI\_Init(&argc, &argv);

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

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

    if (size != 2) {

        fprintf(stderr, "This program requires exactly two processes.\n");

        MPI\_Finalize();

        return 1;

    }

    if (rank == 0) {

        printf("Enter the size of the array (even number): ");

        scanf("%d", &n);

        if (n % 2 != 0) {

            fprintf(stderr, "The size of the array must be even.\n");

            MPI\_Finalize();

            return 0;

        }

        A = (int\*)malloc(sizeof(int) \* n);

        for (int i = 0; i < n; i++) {

            A[i] = i + 1;

        }

        // Send the size of the array to P1

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

    } else if (rank == 1) {

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

    }

    // Calculate local\_n (size of the subarray)

    local\_n = n / 2;

    // Allocate memory for the local subarray

    int\* local\_A = (int\*)malloc(sizeof(int) \* local\_n);

    // Scatter the data from the master (P0) to the workers (P0 and P1)

    MPI\_Scatter(A, local\_n, MPI\_INT, local\_A, local\_n, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Compute the local sum for the subarray

    for (int i = 0; i < local\_n; i++) {

        local\_sum += local\_A[i];

    }

    // Reduce the local sums to get the total sum

    MPI\_Reduce(&local\_sum, &total\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    if (rank == 0) {

        printf("\nThe sum of all elements in the array is: %d\n", total\_sum);

    }

    if (A != NULL) {

        free(A);

    }

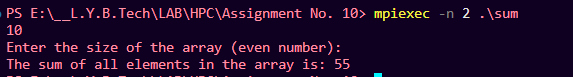
    free(local\_A);

    MPI\_Finalize();

    return 0;

}

**Output:**

****