**Class:** Final Year B.Tech(Computer Science and Engineering)

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

PRN: 22510070

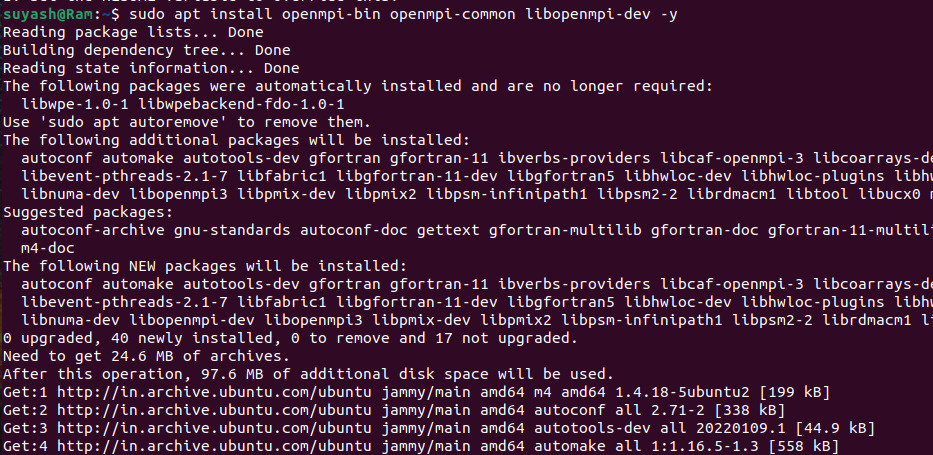
Name:Suyash Yadav

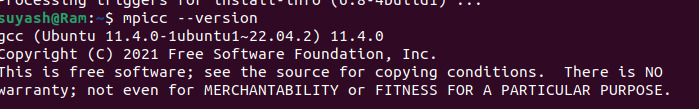
**Practical No. 6**

**Exam Seat No:22510070**

**Title of practical:**

**Installation of MPI & Implementation of basic functions of MPI**





**Problem Statement 1:**

**Implement a simple hello world program by setting number of processes equal to 10**

**Screenshots:**

**#include <mpi.h>**

**#include <stdio.h>**

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

**int rank, size;**

**MPI\_Init(&argc, &argv); // Initialize MPI**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Get process rank**

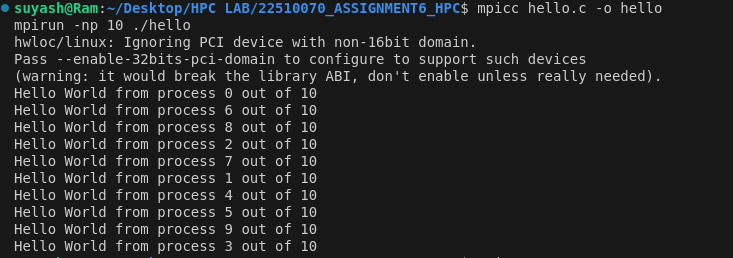
**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Get total number of processes**

**printf("Hello World from process %d out of %d\n", rank, size);**

**MPI\_Finalize(); // Finalize MPI**

**return 0;**

**}**



**Information 1:**

** The Hello World program is the simplest way to test MPI.**

** It ensures MPI installation and environment are working properly.**

** Each process prints its rank and the total number of processes, showing parallel execution.**

** The order of messages may vary due to concurrent execution.**

**Problem Statement 2:**

**Implement a program to display rank and communicator group of five processes**

**Screenshots:**

**#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 color = (world\_rank < (world\_size + 1) / 2) ? 0 : 1;**

**MPI\_Comm newcomm;**

**MPI\_Comm\_split(MPI\_COMM\_WORLD, color, world\_rank, &newcomm);**

**int new\_rank, new\_size;**

**MPI\_Comm\_rank(newcomm, &new\_rank);**

**MPI\_Comm\_size(newcomm, &new\_size);**

**printf("World rank %d/%d -> color=%d -> newcomm rank %d/%d\n",**

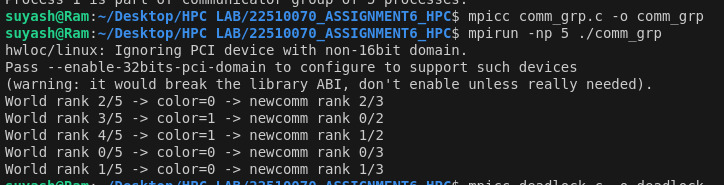
**world\_rank, world\_size, color, new\_rank, new\_size);**

**MPI\_Comm\_free(&newcomm);**

**MPI\_Finalize();**

**return 0;**

**}**



**Info:**

** Rank identifies each process within its communicator.**

** Communicator is a collection of processes that can communicate with each other.**

** MPI\_COMM\_WORLD is the default communicator containing all processes.**

** By running with 5 processes, we can confirm how MPI assigns ranks and handles groups.**

** This problem helps understand process identification and group association in MPI**

**Q3: Implement a MPI program to give an example of Deadlock.**

**Program and screenshots**

**#include <mpi.h>**

**#include <stdio.h>**

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

**int rank;**

**MPI\_Init(&argc, &argv);**

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

**int data;**

**if (rank == 0) {**

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

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

**} else if (rank == 1) {**

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

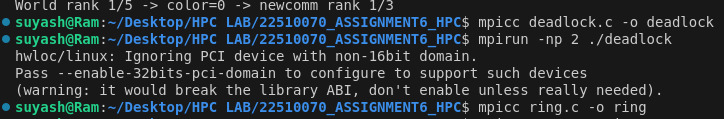
**MPI\_Recv(&data, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**}**

**MPI\_Finalize();**

**return 0;**

**}**



**Info:**

** Deadlock is a state where two or more processes wait forever for each other to complete an operation.**

** In MPI, deadlocks often occur if both processes use MPI\_Send first, and no one is available to MPI\_Recv.**

** This exercise demonstrates what happens when communication is incorrectly ordered.**

** It is useful for learning about safe communication patterns and how to avoid deadlocks in MPI programs.**

**Q4. Implement blocking MPI send & receive to demonstrate Nearest neighbor exchange of data in a ring topology.**

**Program and screenshots**

**#include <mpi.h>**

**#include <stdio.h>**

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

**int rank, size, send\_val, recv\_val;**

**MPI\_Init(&argc, &argv);**

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

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

**send\_val = rank; // each process sends its rank**

**int next = (rank + 1) % size;**

**int prev = (rank - 1 + size) % size;**

**MPI\_Send(&send\_val, 1, MPI\_INT, next, 0, MPI\_COMM\_WORLD);**

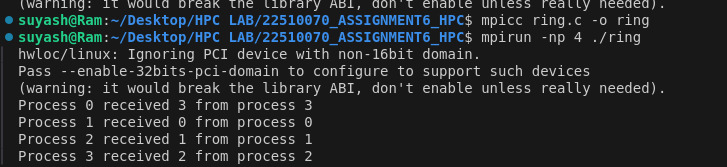
**MPI\_Recv(&recv\_val, 1, MPI\_INT, prev, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**printf("Process %d received %d from process %d\n", rank, recv\_val, prev);**

**MPI\_Finalize();**

**return 0;**

**}**



**Information / Theory:**

* **A ring topology is a structure where each process communicates with its immediate left and right neighbors.**
* **Each process sends its data to the right neighbor and receives from the left neighbor.**
* **This problem demonstrates blocking communication using MPI\_Send and MPI\_Recv.**
* **Such patterns are used in algorithms like distributed matrix multiplication, stencil computations, and scientific simulations.**

**Q5. 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 and screenshots**

**#include <mpi.h>**

**#include <stdio.h>**

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

**int rank, size, n = 8; // array size (must be even)**

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

**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) {**

**if(rank==0) printf("Run with 2 processes!\n");**

**MPI\_Finalize();**

**return 0;**

**}**

**if (rank == 0) {**

**for (int i = 0; i < n/2; i++) local\_sum += A[i];**

**} else {**

**for (int i = n/2; i < n; i++) local\_sum += A[i];**

**}**

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

**if (rank == 0) {**

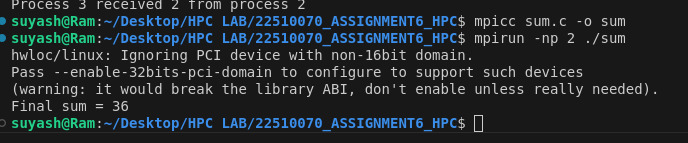
**printf("Final sum = %d\n", total\_sum);**

**}**

**MPI\_Finalize();**

**return 0;**

**}**



**Information / Theory:**

* **Parallel computation can divide work among processes for faster execution.**
* **Here, the array is split into two equal halves.**
  + **Process P0 adds the first half.**
  + **Process P1 adds the second half.**
* **The partial sums are combined using MPI\_Reduce or explicit send/receive.**
* **Demonstrates how MPI enables divide-and-conquer strategies for data processing.**
* **Such patterns are useful in large-scale data analysis, simulations, and numerical computations.**

**Github Link: https://github.com/Suyashyadav07/HPC**