**High Performance Computing Lab**

**Practical No. 7**

Name – Yash Nawale

Batch – B4

PRN – 21510074

**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

#include <mpi.h>

#include <stdio.h>

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

    MPI\_Init(NULL, NULL);

    int world\_size;

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

    int world\_rank;

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

    printf("Hello world from processor rank %d out of %d processors\n", world\_rank, world\_size);

    MPI\_Finalize();

}

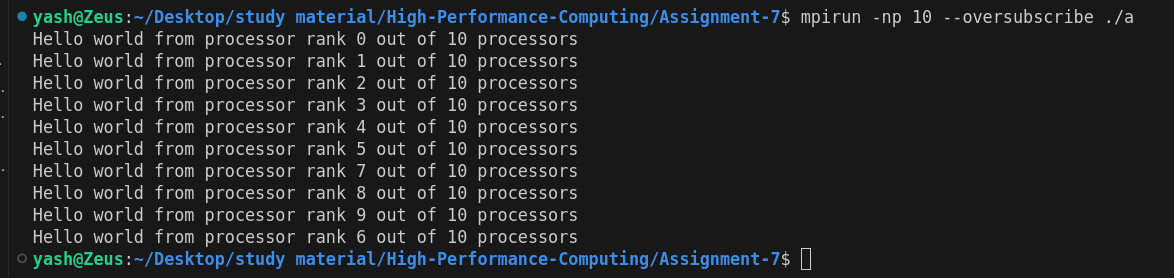
    // Get the name of the processor

    // char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

    // int name\_len;

    // MPI\_Get\_processor\_name(processor\_name, &name\_len);

**Screenshots:**

****

**Information 1:**

**1. #include <mpi.h>**

* This includes the MPI (Message Passing Interface) library, providing the necessary functions and constants to enable MPI communication in the program.

**2. #include <stdio.h>**

* This includes the standard input/output library in C, allowing the program to use functions like printf for displaying output.

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

* The main function of the program that takes argc (argument count) and argv (argument vector), standard parameters for C programs. These arguments can be passed through the command line and are required by MPI to manage the initialization process.

**4. MPI\_Init(&argc, &argv)**

* MPI Initialization: This function initializes the MPI environment. It is mandatory to call MPI\_Init before any other MPI function.
* This sets up all necessary communication settings and allows the program to use multiple processes.

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

* Rank: This line retrieves the rank (ID) of the current process in the communicator (MPI\_COMM\_WORLD), and stores it in the variable rank.
  + The rank is a unique ID for each process, used to identify it. It ranges from 0 to size-1 (where size is the total number of processes).
  + For example, if you are running the program with 4 processes, the ranks will be 0, 1, 2, and 3.

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

* Size: This line gets the total number of processes in the communicator (MPI\_COMM\_WORLD) and stores it in the variable size.
  + size represents the total number of processes involved in this program.
  + For example, if you run the program with 4 processes, size will be 4.

**7. printf("Hello world from process %d of %d\n", rank, size)**

* This line prints a "Hello world" message from each process. Each process outputs its rank and the total number of processes (size).

**8. MPI\_Finalize()**

* This function finalizes the MPI environment and is mandatory to call at the end of the program.
* After MPI\_Finalize(), no other MPI functions can be called.
* It ensures that all processes terminate correctly and releases any resources used by MPI.

**9. What is Oversubscription?**

Oversubscription occurs when you run more MPI processes than the available CPU cores. While this is permissible, it can lead to:

* Increased Context Switching: The operating system switches between processes, which can reduce performance.
* Resource Contention: Multiple processes compete for CPU time, memory, and other resources.
* Potential Performance Degradation: Especially noticeable in compute-intensive applications.

**10.When to Use Oversubscription:**

* Testing and Development: When you want to simulate larger MPI applications on a single machine.
* Educational Purposes: Learning MPI concepts without requiring multiple physical machines.
* Lightweight Applications: Programs that are not heavily CPU-bound can tolerate oversubscription better.

**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(NULL, NULL);

    int world\_rank;

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

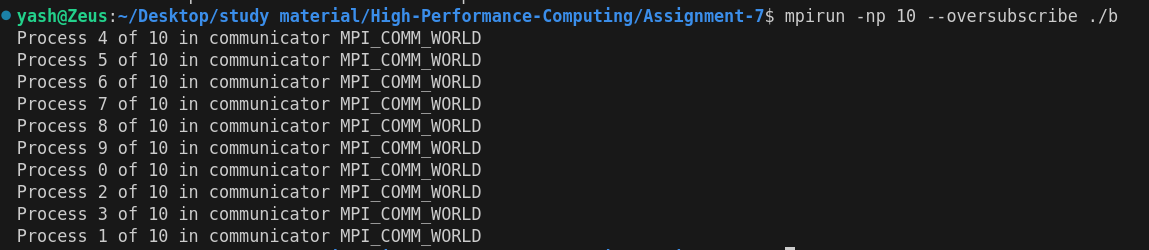
    int world\_size;

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

    printf("Process %d of %d in communicator MPI\_COMM\_WORLD\n", world\_rank, world\_size);

    MPI\_Finalize();

}



**Information:**

**1.Include Headers:**

#include <mpi.h>

#include <stdio.h>

* <mpi.h>: This header file includes definitions for MPI functions and constants, enabling message passing in parallel programs.
* <stdio.h>: This standard input/output library allows the program to use functions like printf for displaying output.

**2. Main Function:**

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

* The main function begins, taking argc (argument count) and argv (argument vector) as parameters. These are standard parameters in C for handling command-line arguments.

**3. Initialize MPI:**

MPI\_Init(&argc, &argv);

* This function initializes the MPI environment. It must be called before any other MPI functions to set up the necessary resources for parallel execution.

**4. Get Rank of the Current Process:**

int rank;

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

* Rank: This retrieves the rank (unique identifier) of the current process within the MPI\_COMM\_WORLD communicator and stores it in the variable rank.
* The rank ranges from 0 to size-1, where size is the total number of processes.

**5. Get Total Number of Processes:**

int size;

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

* This function retrieves the total number of processes in the MPI\_COMM\_WORLD communicator and stores it in size. This allows the program to know how many processes are involved in the execution.

**6. Process Count Check:**

if (size != 5) {

if (rank == 0) {

printf("This program must be run with exactly 5 processes.\n");

}

MPI\_Finalize();

return 1; // Exit with an error code

}

* This block checks if the total number of processes (size) is equal to 5. If not:
  + The process with rank 0 (typically the main process) prints a message indicating that exactly 5 processes are required.
  + The program finalizes the MPI environment using MPI\_Finalize() and exits with an error code (1). This prevents execution of the main logic if the process count requirement is not met.

**7. Display Rank and Communicator Group:**

printf("Process %d of %d is part of communicator group MPI\_COMM\_WORLD\n", rank, size);

* Each process prints its rank and the total number of processes. This line will be executed only if the number of processes is exactly 5.
* The output format is Process X of Y is part of communicator group MPI\_COMM\_WORLD, where X is the rank and Y is the total number of processes.

**8. Finalize MPI:**

MPI\_Finalize();

* This function finalizes the MPI environment. It is called at the end of the program to clean up resources and ensure that the MPI environment is properly shut down.

**9. Key Points :**

* **Communicator**: A communicator allows processes to communicate with each other. MPI\_COMM\_WORLD is the default communicator that includes all the processes launched in an MPI program.
* **Rank**: Each process is assigned a unique rank, which is used to identify it during communication.
* **Size**: The total number of processes participating in the program, which is crucial for coordination in parallel programming.

10. **mpirun --oversubscribe**:

This command allows you to run more processes than there are physical CPU cores.

[**GitHub link :-**](https://github.com/Shreyak810/HPC-LAB/tree/main/Assignment-7)

[**https://github.com/YashNawale26/High-Performance-Computing**](https://github.com/YashNawale26/High-Performance-Computing)