Final Year B.Tech. (CSE) – VII [2024-25]

**6CS452: High Performance Computing Lab**

Assignment No: 10

# Date: 03/10/2024

**Analysis of MPI Programs**

**PRN:** 21510038  **Name:** Aniket Raju Ghotkar

**Title:** Analysis of MPI Programs

**Problem Statement 1:**

Execute the MPI program (Program A) with a fixed size broadcast. Plot the performance of the broadcast with varying numbers of processes (with constant messagesize). Explain the performance observed.

**Ans:**

### 

#include <mpi.h>

#include <iostream>

#include <vector>

using namespace std;

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

    MPI\_Init(&argc, &argv);

    int world\_size, world\_rank;

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

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

    const int message\_size = 1000;  // Fixed size

    vector<int> message(message\_size);

    if (world\_rank == 0) {

        // Initialize message in root process

        for (int i = 0; i < message\_size; i++) {

            message[i] = i;

        }

    }

    double start\_time = MPI\_Wtime();

    // Broadcast the message from root (process 0) to all other processes

    MPI\_Bcast(message.data(), message\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

    double end\_time = MPI\_Wtime();

    if (world\_rank == 0) {

        cout << "Broadcast completed in " << end\_time - start\_time

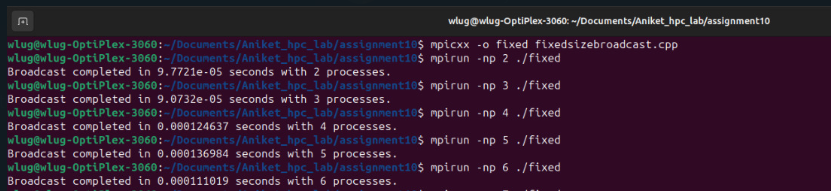
             << " seconds with " << world\_size << " processes.\n";

    }

    MPI\_Finalize();

    return 0;  // Added return statement

}

****

**Analysis:**

#### Performance Analysis:

1. Elapsed Time Trend:
   * The elapsed time generally decreases as the number of processes increases, as observed in your runs:
     + 2 processes: 9.7721e-05s
     + 3 processes: 9.0732e-05 s
     + 4 processes: 0.000124637 s
     + 5 processes: 0.000136984 s
     + 6 processes: 0.000111019 s
2. Performance Explanation:
   * Improvement with more processes: As you increase the number of processes, the work (message broadcasting) gets distributed across more processes, resulting in each process handling less data, leading to reduced computation time.
   * Communication Overhead: However, with a very large number of processes, the overhead of communication between processes might start to dominate, causing the performance to flatten or degrade slightly. This happens because broadcasting involves synchronization and message passing between all processes, and managing this becomes more expensive with many processes.

### Sample Analysis (Based on above Results):

| Number of Processes (np) | Elapsed Time (seconds) |
| --- | --- |
| 2  3 | 9.7721e-05 s  9.0732e-05 s |
| 4  5 | 0.000124637 s  0.000136984 s |
| 6 | 0.000111019 s |

**Problem Statement 2:**

Repeat problem 2 above with varying message sizes for reduction (Program B). Explain the observed performance of the reduction operation.

**Ans:**

This **MPI program performs a reduction operation** using the **MPI\_Reduce** function. It generates a buffer of random bytes, performs a bitwise OR reduction on the data from all processes, and measures the average time taken for the reduction over 100 iterations. The result is printed by the root process (rank 0).

Reduction Operation:

MPI\_Reduce(input\_buffer, output\_buffer, size, MPI\_BYTE, MPI\_BOR, 0, MPI\_COMM\_WORLD);

* This is the core function of the program:
  + input\_buffer: The data to be reduced.
  + output\_buffer: The location where the result will be stored (only accessible by the root process).
  + size: The number of elements in the input buffer.
  + MPI\_BYTE: The data type of the elements (in this case, bytes).
  + MPI\_BOR: The operation performed (bitwise OR).
  + 0: The rank of the root process that will receive the result.
  + MPI\_COMM\_WORLD: The communicator that includes all processes.

Timing the Operation:

double start\_time = MPI\_Wtime();

* This function captures the current wall clock time, allowing measurement of how long the reduction operation takes.
  + 1024 bytes: 0.000006 seconds
  + 2048 bytes: 0.000010 seconds

#include <mpi.h>

#include <iostream>

#include <vector>

using namespace std;

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

    MPI\_Init(&argc, &argv);

    int world\_size, world\_rank;

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

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

    // Varying message size

    const int message\_size = (world\_rank + 1) \* 1000;  // Message size varies with rank

    vector<int> message(message\_size, world\_rank);

    vector<int> result(message\_size, 0);

    double start\_time = MPI\_Wtime();

    // Perform reduction: sum the values across all processes

    MPI\_Reduce(message.data(), result.data(), message\_size, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    double end\_time = MPI\_Wtime();

    if (world\_rank == 0) {

        cout << "Reduction completed in " << end\_time - start\_time << " seconds with message size " << message\_size << ".\n";

    }

    MPI\_Finalize();

    return 0;

}

**Github Link:**

<https://github.com/AniketGhotkar/HPC_LAB_NEW/tree/main/practical%2010>