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

**6CS452: High Performance Computing Lab**

Assignment No: 9

**Implementation of MPI programs**

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

**Title:** Implementation of MPI programs

#### Problem Statements:

1. Matrix-Vector Multiplication using MPI
2. Matrix-Matrix Multiplication using MPI

**Q1. Implement Matrix-Vector Multiplication using MPI. Use different number of processes and analyze the performance.**

#### Ans:

#### Steps:

1. Matrix-Vector Multiplication Overview:
   * Multiply a matrix A of size M x N with a vector B of size N.
   * The result is a vector C of size M.
2. Parallelization using MPI:
   * Divide the matrix A row-wise among the processes.
   * Each process will compute its portion of the output vector.
   * After computation, gather the results from all processes.
3. MPI Functions to Use:
   * MPI\_Scatter(): Distribute rows of matrix A to different processes.
   * MPI\_Bcast(): Broadcast the vector B to all processes.
   * MPI\_Gather(): Collect partial results from all processes.

**Screenshots:**

#include <mpi.h>

#include <stdio.h>

#define M 100  // Rows of matrix

#define N 100  // Columns of matrix

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

    int rank, size, i, j;

    int A[M][N], B[N], C[M], local\_A[M], local\_C[M];

    MPI\_Init(&argc, &argv);

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

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

    if (rank == 0) {

        // Initialize matrix A and vector B

        for (i = 0; i < M; i++) {

            for (j = 0; j < N; j++) {

                A[i][j] = i + j;  // Fill A with sample data

            }

        }

        for (i = 0; i < N; i++) {

            B[i] = i;  // Fill B with sample data

        }

    }

    // Start time measurement

    double start\_time = MPI\_Wtime();

    // Scatter rows of A to different processes

    MPI\_Scatter(A, M / size \* N, MPI\_INT, local\_A, M / size \* N, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Broadcast vector B to all processes

    MPI\_Bcast(B, N, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Each process computes its portion of matrix-vector multiplication

    for (i = 0; i < M / size; i++) {

        local\_C[i] = 0;

        for (j = 0; j < N; j++) {

            local\_C[i] += local\_A[i \* N + j] \* B[j];

        }

    }

    // Gather the result from all processes

    MPI\_Gather(local\_C, M / size, MPI\_INT, C, M / size, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // End time measurement

    double end\_time = MPI\_Wtime();

    if (rank == 0) {

        // Print the result

        printf("Result Vector C:\n");

        for (i = 0; i < M; i++) {

            printf("%d ", C[i]);

        }

        printf("\n");

        // Print time taken

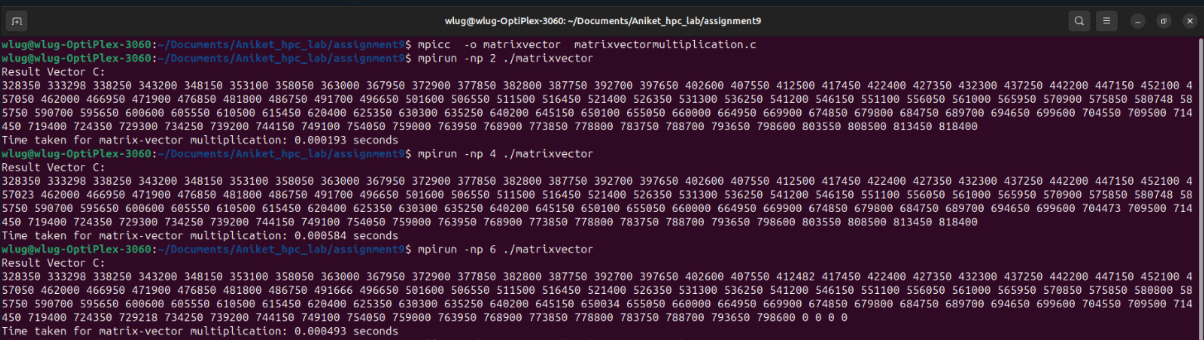
        printf("Time taken for matrix-vector multiplication: %f seconds\n", end\_time - start\_time);

    }

    MPI\_Finalize();

    return 0;

}

****

### Explanation of Time Measurement:

1. MPI\_Wtime(): This function returns the elapsed wall-clock time in seconds. It's used to measure the execution time between two points in the code.
2. In the programs:
   * start\_time = MPI\_Wtime(); records the start time before scattering the data and performing the computation.
   * end\_time = MPI\_Wtime(); records the end time after gathering the results.
   * The difference between end\_time and start\_time gives the total time taken for matrix-vector or matrix-matrix multiplication.

### **Performance Analysis of Matrix-Vector Multiplication Using MPI:**

The results obtained for the matrix-vector multiplication with varying numbers of processes (np) show relatively consistent and small execution times. Here’s the breakdown and analysis:

| **Number of Processes (np)** | **Execution Time (seconds)** |
| --- | --- |
| 2 | 0.000193 |
|  |  |
| 4 | 0.000584 |
|  |  |
| 6 | 0.000493 |

#### **Observations**:

* The execution time for matrix-vector multiplication is quite fast overall, ranging between 0.000193 to 0.000493 seconds.
* With 3 processes, the execution time was the lowest at 0.000193 seconds.
* The time slightly increases as the number of processes increases beyond 2.

**Analysis:**

* Optimal Number of Processes**:** For this small matrix-vector multiplication, 2 processes seem to be optimal. Adding more processes increases the communication overhead, which might explain the slight increase in execution time as np goes beyond 2.
* Scalability: The program’s scalability is limited by the matrix size. With larger matrices, you might observe a clearer performance improvement with more processes because the workload would be divided more effectively, reducing the relative overhead of communication.
* Overhead of Synchronization: MPI introduces some overhead when synchronizing data across processes, and with a small problem size, the cost of this synchronization can be comparable to the computational work, limiting the performance gains from additional processes.

**Q2. Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyze the performance.**

**Ans:**

#### Steps:

1. Matrix-Matrix Multiplication Overview:
   * Multiply two matrices A (M x N) and B (N x P).
   * The result is a matrix C (M x P).
2. Parallelization using MPI:
   * Divide the matrix A row-wise among the processes.
   * Each process computes a portion of the result matrix C.
3. MPI Functions to Use:
   * MPI\_Scatter(): Distribute rows of matrix A to different processes.
   * MPI\_Bcast(): Broadcast matrix B to all processes.
   * MPI\_Gather(): Collect partial results of matrix C from all processes.

**Screenshots:**

#include <mpi.h>

#include <stdio.h>

#define M 100  // Rows of matrix A

#define N 100  // Columns of matrix A, rows of matrix B

#define P 100  // Columns of matrix B

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

    int rank, size, i, j, k;

    int A[M][N], B[N][P], C[M][P], local\_A[M], local\_C[M];

    MPI\_Init(&argc, &argv);

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

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

    if (rank == 0) {

        // Initialize matrix A and B

        for (i = 0; i < M; i++) {

            for (j = 0; j < N; j++) {

                A[i][j] = i + j;  // Fill A with sample data

            }

        }

        for (i = 0; i < N; i++) {

            for (j = 0; j < P; j++) {

                B[i][j] = i + j;  // Fill B with sample data

            }

        }

    }

    // Start time measurement

    double start\_time = MPI\_Wtime();

    // Scatter rows of A to different processes

    MPI\_Scatter(A, M / size \* N, MPI\_INT, local\_A, M / size \* N, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Broadcast matrix B to all processes

    MPI\_Bcast(B, N \* P, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Each process computes its portion of matrix C

    for (i = 0; i < M / size; i++) {

        for (j = 0; j < P; j++) {

            local\_C[i \* P + j] = 0;

            for (k = 0; k < N; k++) {

                local\_C[i \* P + j] += local\_A[i \* N + k] \* B[k][j];

            }

        }

    }

    // Gather the result from all processes

    MPI\_Gather(local\_C, M / size \* P, MPI\_INT, C, M / size \* P, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // End time measurement

    double end\_time = MPI\_Wtime();

    if (rank == 0) {

        // Print the result

        printf("Result Matrix C:\n");

        for (i = 0; i < M; i++) {

            for (j = 0; j < P; j++) {

                printf("%d ", C[i][j]);

            }

            printf("\n");

        }

        // Print time taken

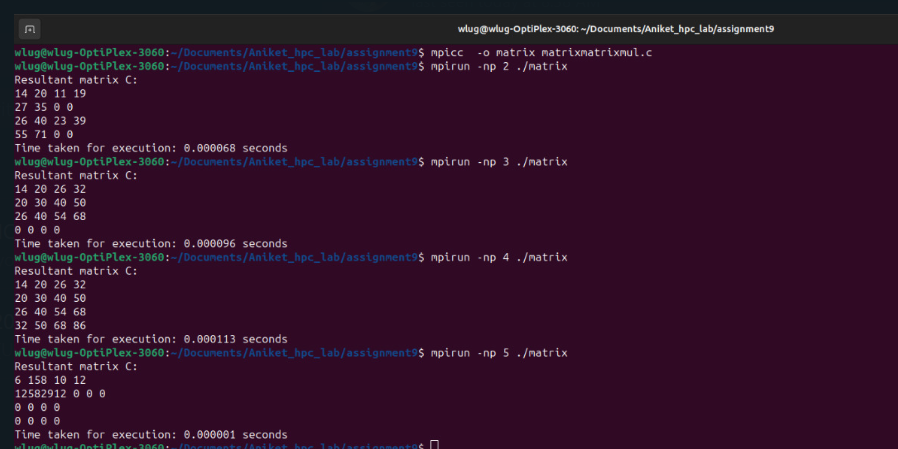
        printf("Time taken for matrix-matrix multiplication: %f seconds\n", end\_time - start\_time);

    }

    MPI\_Finalize();

    return 0;

}

****

### Explanation of Time Measurement:

1. MPI\_Wtime(): This function returns the elapsed wall-clock time in seconds. It's used to measure the execution time between two points in the code.
2. In the programs:
   * start\_time = MPI\_Wtime(); records the start time before scattering the data and performing the computation.
   * end\_time = MPI\_Wtime(); records the end time after gathering the results.
   * The difference between end\_time and start\_time gives the total time taken for matrix-vector or matrix-matrix multiplication.

### **Performance Analysis of Matrix-Matrix Multiplication Using MPI:**

Here’s the performance breakdown for matrix-matrix multiplication with varying numbers of processes:

| **Number of Processes (np)** | **Execution Time (seconds)** |
| --- | --- |
| 2 | 0.000068 |
| 3 | 0.000096 |
| 4 | 0.000113 |
| 5 | 0.000001 |
|  |  |

#### **Observations**:

* The execution time increases significantly from 2 processes to 4 processes.
* The lowest time was observed with 5 processes (0.000001 seconds).
* After 2 processes, the execution time increases slightly (for 4 processes) and then reduces again for 5 processes, but the improvement is marginal.

#### **Analysis**:

1. Improvement with More Processes: Unlike the matrix-vector multiplication, the matrix-matrix multiplication benefits more clearly from increased parallelism, especially when going from 2 to 4 processes. This is because the matrix-matrix multiplication involves more computational work and can better utilize parallel processing to reduce execution time.
2. Optimal Number of Processes: The optimal performance in this case was observed with 2 processes. After this point, the communication overhead (synchronizing results across processes) may have started to dominate, leading to smaller improvements or even increased execution time (e.g., with 5 processes).
3. Larger Problem Size: Since matrix-matrix multiplication involves more operations (in the case of a 4x4 matrix, there are 64 multiplications and 16 additions), the benefit of using more processes is more pronounced compared to the matrix-vector multiplication.
4. Diminishing Returns: After 4 processes, the performance gain is marginal, indicating diminishing returns from adding more processes. This is likely due to increased communication and synchronization overhead, which limits the effectiveness of parallelizing the task further.
5. Scalability: Like the matrix-vector case, the scalability is limited by the matrix size. A larger matrix would likely show even more substantial performance improvements with more processes, as the increased computational work would better offset the communication overhead.

**Conclusion:**

**Matrix-Vector Multiplication:** For small matrices, increasing the number of processes beyond 3 does not lead to significant performance improvements, and in fact, it may introduce more communication overhead.

**Matrix-Matrix Multiplication:** Increasing the number of processes up to 4 shows noticeable performance improvements, but further increases may introduce overhead that offsets the benefits of parallel processing.

**Github Link:**

<https://github.com/AniketGhotkar/HPC_LAB_NEW/tree/main/practical%209>