西北工業大學

Graduate Course Programming Report

得分：

Student ID 2024280038

Name Jason Rich Darmawan

Course Name High Performance Computing

Date 2025/01/07

西北工业大学研究生院

1 Parallel Matrix-Vector Multiplication

The algorithm source code is provided in APPENDIX B. The implementation can be accessed from the GitHub repository [1]: the source code is available in the folder 07-matrix_vector_multiplication.

The purpose of the experiment is to evaluate the reduction in running time for matrix-vector multiplication through parallelization. Specifically, the goal is to demonstrate that distributing the matrix rows in a block-row fashion across multiple processes can effectively reduce the computational time required for matrix-vector multiplication. By partitioning the matrix and leveraging parallelism, the experiment aims to show that this distribution strategy leads to improved performance in terms of running time.

The experiment was conducted using an input matrix of size 8×10^7 and an input vector of size $10^7 \times 1$. The performance of the matrix-vector multiplication was evaluated under different process configurations, specifically utilizing 1 to 8 processes. The results were then analyzed to assess the impact of penalization on the running time.

The program was compiled and executed on a MacBook Pro 13" (M2, 2022), equipped with an Apple M2 chip, featuring an 8-core CPU and 16 GB of unified memory. The software environment consisted of macOS Sequoia 15.0 as the operating system. The program was compiled using Apple's Clang compiler, version 15.0.0 with support for the C++17 standard. For parallel computation, the program utilized the OpenMPI library, version 5.0.6, to enable distributed processing across multiple cores.

The parallel algorithm consists of two main components: matrix row distribution via the MPI_Mat_vect_scatter_row function and matrix-vector multiplication via the MPI_Mat_vect_mult function.

1.1 MPI_Mat_vect_scatter_row Function

The MPI_Mat_vect_scatter_row function is responsible for distributing the rows of an input matrix $A_{m,n}$, where m represents the number of rows and n represents the number of columns) from the root process (Process 0) to multiple processes within the custom Message Passing Interface (MPI) communicator. The distribution mechanism depends on the number of processes available and the number of rows in the matrix.

Initially, the matrix $A_{m,n}$ is stored entirely on the root process (Process 0). This matrix is then distributed across the available processes according to two key constraints:

Constraint 1: $P \geq m$ (Optimal Scenario)

When the number of available processes is greater than or equal to the number of rows in the matrix, the matrix rows are evenly distributed across the processes. Each process is assigned one row, ensuring a balanced workload across the processes. This scenario is achieved using the MPI_Comm_split function, which divides the communicator into smaller groups, assigning each process its respective row of the matrix.

In this optimal scenario, where $P \geq m$, each process holds exactly one row of the matrix. The distribution is illustrated in Figure 1, where the matrix is split among m processes, with Process 0, Process 1, ..., Process $m-1$ each receiving one row.

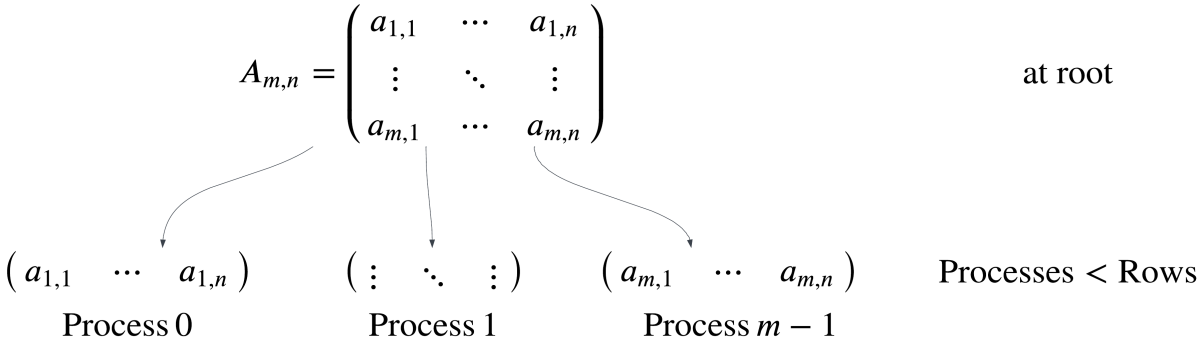

Figure 1 Matrix Rows Distribution in Optimal Scenario

Constraint 2: $P < m$ (Worst-case Scenario)

When the number of available processes P is smaller than the number of rows in the matrix, some processes must be assigned more than one row to balance the computational load. This uneven distribution is achieved using the MPI_Scatterv function, which allows for distributing different numbers of rows to each process based on the total number of rows and processes available.

In this worst-case scenario, the workload distribution becomes uneven, and some processes are assigned more rows than others. This results in certain processes performing more computations, leading to potential load imbalance. The distribution is illustrated in Figure 2, where the matrix rows are distributed unevenly across the processes.

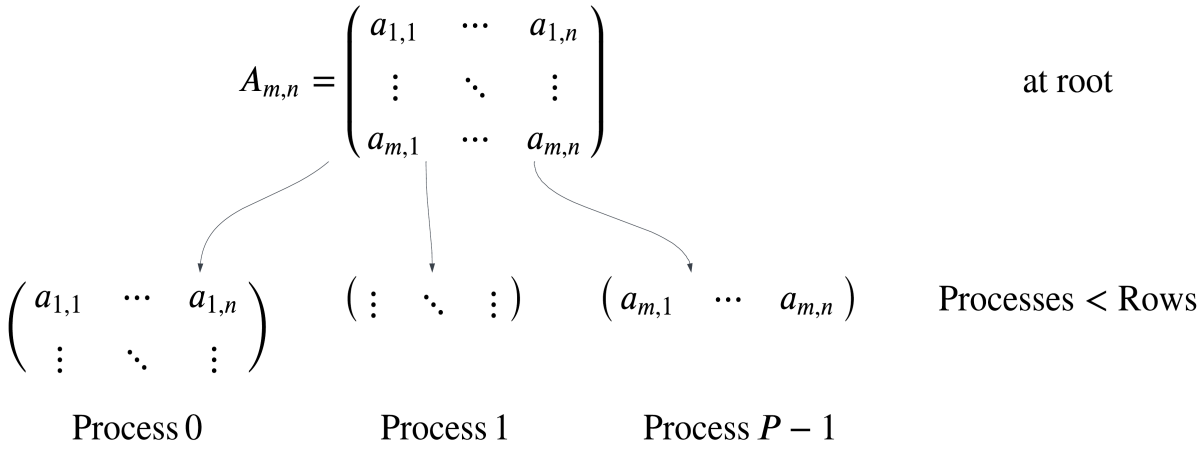

Figure 2 Matrix Rows Distribution in Worst-Case Scenario

1.2 MPI_Mat_vect_mult Function

The MPI_Mat_vect_mult function is responsible for performing parallel matrix-vector multiplication, gathering partial results computed by each process, and broadcasting the final computed results to all processes within the custom communicator. This function utilizes the MPI_Allgatherv function to collect partial computation results of varying sizes from each process and broadcast them to all processes in the communicator.

In matrix-vector multiplication, the goal is to compute the result vector $\vec{y} = A_{m,n} \cdot \vec{x}$, where $A_{m,n}$ is the matrix of size $m \times n$ and $\vec{x}$ is the input vector of size $n \times 1$. The computed result vector $\vec{y}$ is distributed across all processes in the communicator. The MPI_Allgatherv operation is illustrated in Figure 3 and Figure 4.

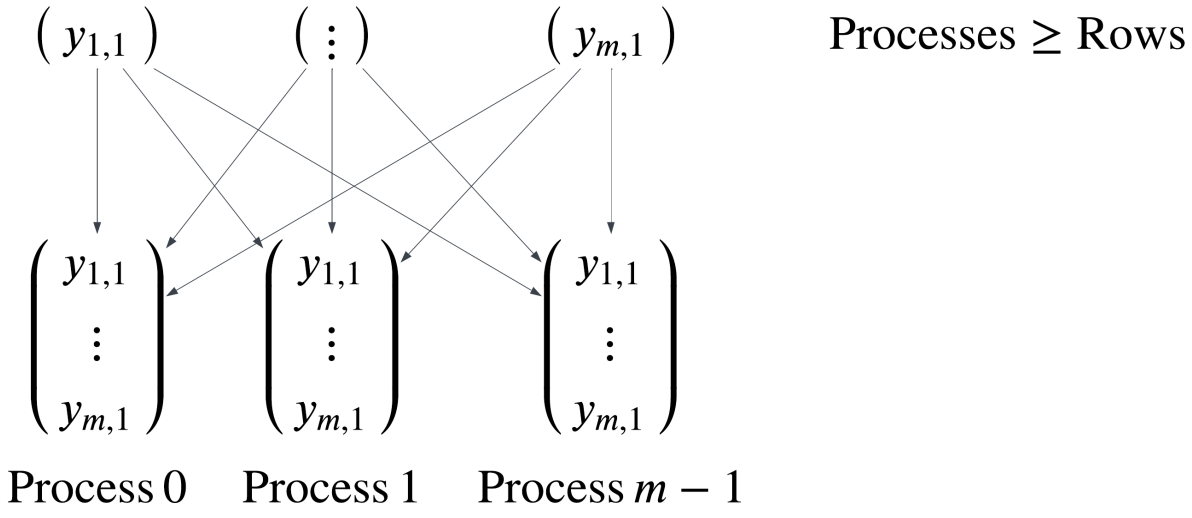

Figure 3 Parallel Matrix-Vector Multiplication in Optimal Scenario

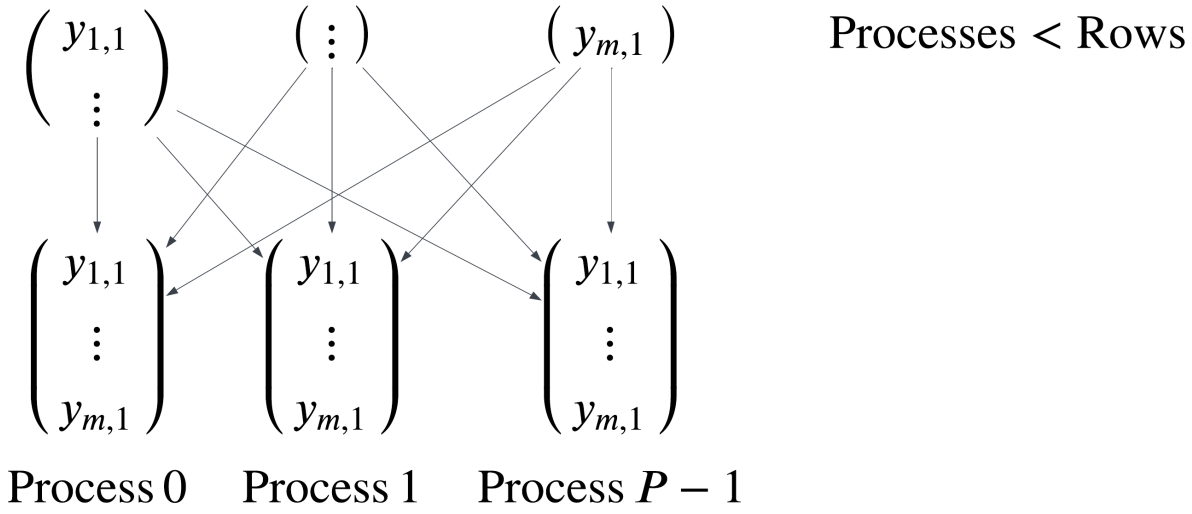

Figure 4 Parallel Matrix-Vector Multiplication in Worst-Case Scenario

1.3 Compile and Running Command

To compile the source code, execute the following commands:

```
chmod +x build.sh
./build.sh
```

To run the program, use the following command:

```
mpirun –np <number of processes> ./build/main
```

1.4 Experimental Results

The screenshots of the program's running results are provided in APPENDIX C for reference.

Consider a matrix of dimensions 8×10^7 being multiplied by a vector of size $10^7 \times 1$. Table 1 demonstrates the relationship between the running time and the number of processes for two scenarios:

$$\Delta\text{Running time} = \begin{cases} \text{Negative (Decrease)}, & \text{Process Count} \leq \text{Rows} \\ 0, & \text{Process Count} > \text{Rows} \end{cases}$$

The running time decreases ($\Delta\text{Running time} < 0$) as the number of processes increases when the process count is less than or equal to the number of rows ($\text{Process Count} \leq \text{Rows}$). This improvement is attributed to the distribution of workload among processes. However, uneven workload distribution results in the system waiting for the process with the largest workload to finish. Additionally, due to communication overhead, setting the process count

equal to the number of rows ($\text{Process Count} = \text{Rows}$) does not yield the best performance, although it performs better compared to a single process ($\text{Process Count} = 1$).

When the process count exceeds the number of rows ($\text{Process Count} > \text{Rows}$), no further reduction in running time ($\Delta\text{Running time} = 0$) is observed. This is because the algorithm employs the MPI_Comm_split function to create a new communicator, effectively capping the number of active processes at $\text{Process Count} \leq \text{Rows}$.

Table 1: Relationship between Running Time and Number of Processes

Process Count	Trial 1 (μs)	Trial 2 (μs)	Trial 3 (μs)	Trial 4 (μs)	Trial 5 (μs)	Average Time (μs)
1	414711	515880	528227	396741	375141	446140
2	547340	305170	248223	383517	267574	350364,8
3	410318	308783	213828	312796	228827	294910,4
4	198239	161877	185414	198384	173927	183568,2
5	341011	257647	335624	294165	261457	297980,8
6	638028	671693	374777	338029	389217	482348,8
7	871031	297949	287442	308211	309911	414908,8
8	440638	380230	359616	283111	293561	351431,2

2 Parallel Shortest Path Calculation Between Any Two Vertices in a Graph

The algorithm source code is provided in APPENDIX D. The implementation can be accessed from the GitHub repository [1]: the source code is available in the folder 08-floyd_warshall.

The purpose of the experiment is to evaluate the reduction in running time for shortest path calculation between any two vertices through parallelization. Specifically, the goal is to demonstrate that distributing the graph rows in a block-row fashion across multiple processes can effectively reduce the computational time required for shortest path calculation between any two vertices. By partitioning the distance matrix and leveraging parallelism, the experiment aims to show that this distribution strategy leads to improved performance in terms of running time.

The experiment was conducted using an input graph under various vertex and process configurations. Specifically, tests were performed with configuration of 5 vertices, 50 vertices and 500 vertices, utilizing 1, 5 and 7 processes. The edges for each configuration were determined using the formula for the maximum number of edges in a directed graph:

$$\text{edges} = \text{vertices} \times (\text{vertices} - 1)$$

The program was compiled and executed on a MacBook Pro 13" (M2, 2022), equipped with an Apple M2 chip, featuring an 8-core CPU and 16 GB of unified memory. The software environment consisted of macOS Sequoia 15.0 as the operating system. The program was compiled using Apple's Clang compiler, version 15.0.0 with support for the C++17 standard. For parallel computation, the program utilized the OpenMPI library, version 5.0.6, to enable distributed processing across multiple cores.

The parallel algorithm, implemented in the MPI_GRAPH_shortest_paths_floyd_warshall function, compromises three primary components: (1) the distribution of the graph using the MPI_Scatterv function, (2) the computation of the shortest paths utilizing the Floyd-Warshall Algorithm, and (3) the aggregation of partials results computed by each process using the MPI_Allgatherv function.

2.1 MPI_GRAPH_shortest_paths_floyd_warshall

The MPI_GRAPH_shortest_paths_floyd_warshall function is responsible for distributing the rows of an input graph $V_{n,n}$, where n represents the vertices from the root process (Process 0) to multiple processes within a custom Message Passing Interface (MPI) communicator. The distribution mechanism aligns with the approach used in the Parallel Matrix-Vector Multiplication algorithm, factoring in both the number of available processes and the total number of vertices.

The Floyd-Warshall Algorithm computes the shortest paths between any pair of vertices by iteratively considering intermediate vertices. This process is mathematically represented as follows:

$$\text{distance}[i][j] = \min(\text{distance}[i][j], \text{distance}[i][k] + \text{distance}[k][j])$$

Here, i represents the source vertex, j represents the destination vertex, k denotes the intermediate vertex. In essence, the formula calculates the shortest distance between the source and destination vertices, factoring in the possibility of passing through an intermediate vertex.

In addition to distributing graph rows among available processes, the algorithm broadcasts the intermediate row to all processes to ensure consistent and accurate computation. Initially, the intermediate row is broadcasted by Process 0 since k starts at 0. With each iteration, as k increments, the process responsible for broadcasting the updated intermediate row changes dynamically. This assignment is determined by the following algorithm.

$$\text{if remainder} = 0$$

$$k\_\text{owner} = \frac{k}{\text{max\_rows\_per\_process}}$$

else

$$k\_\text{owner} = \begin{cases} \dfrac{k}{\text{max\_rows\_per\_process}}, & \text{if } k < \text{remainder} * \text{max\_rows\_per\_process} \\ \dfrac{k - \text{remainder}}{\text{max\_rows\_per\_process} - 1}, & \text{otherwise} \end{cases}$$

Where:

- $\text{remainder} = \text{vertices} \,\%\, \text{size}$
- $\text{max\_row\_per\_process} = (\text{vertices} + size - 1) \,/\, \text{size}$

Alternatively, to enhance readability and maintainability, the computation of k_owner can be implemented using the algorithm outlined in APPENDIX F. However, this approach may exhibit scalability limitations when the number of processes increases to the thousands. It is important to note that the experimental results presented do not utilize the alternative algorithm.

This process ensure that intermediate rows are broadcasted efficiently by the process owning the row, as illustrated in Figure 5, which depicts the local data at each process during computation.

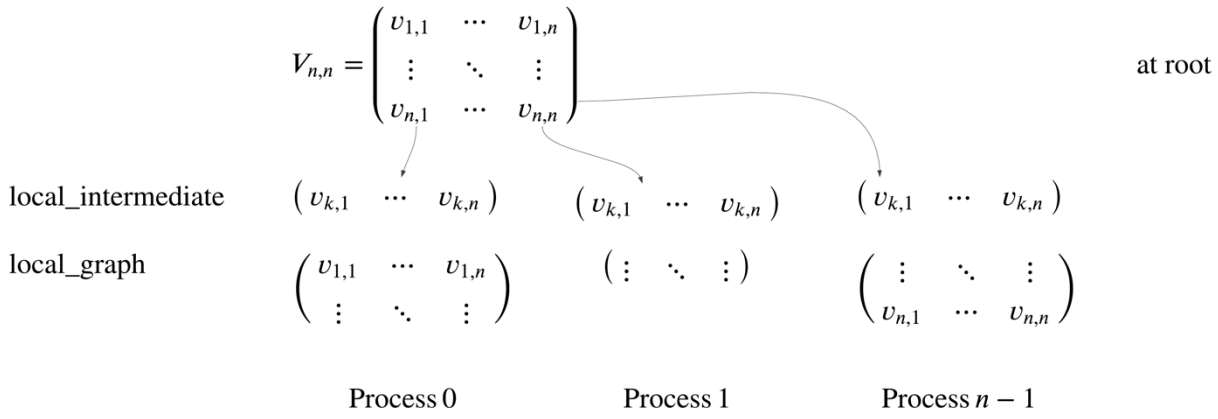

Figure 5 Parallel Shortest Paths Calculation Graph Distribution

2.2 Compile and Running Command

To compile the source code, execute the following commands:

```
chmod +x build.sh
./build.sh
```

To run the program, use the following command:

```
mpirun -np <number of processes> ./build/main
```

2.3 Experimental Results

Consider an input graph under various vertex and process configurations. Table 2 highlights the impact of the number of processes and vertices on the running time of shortest path calculations. Detailed results are provided in APPENDIX E.

The table demonstrates that parallelization using the Floyd-Warshall algorithm to find shortest paths between any two vertices introduces overhead for small graphs (under 50 vertices). However, for large graphs (over 500 vertices), parallelization proves to be significantly beneficial, reducing the overall running time.

The overhead for smaller graphs arises primarily due to communication between processes: including:

- The distribution of the graph by the root process using the MPI_Scatterv function.
- Broadcasting the intermediate row during computation using the MPI_Bcast function.
- Gathering computed partial results from each process using the MPI_Allgatherv function.

These communication steps introduce latency, which outweighs the computational benefits for small graphs but becomes negligible compared to the computational gains for larger graphs.

Table 2 Average Running Time for Shortest Paths Calculation

Process Count	5 vertices (μs)	50 vertices (μs)	500 vertices (μs)
1	7,4	1074,8	782701,4
5	33,6	2417,4	290261,6
7	49	2275,2	246217,4

3 Conclusion

Parallelization can be achieved by distributing data across available processes using the MPI_Scatterv function. Subsequently, partial results from each process are collected through the MPI_Allgatherv function. A notable limitation of this approach is the communication overhead, which may outweigh its advantages for small-scale computations. However, this overhead becomes negligible when processing large data.

Experimental results highlight the importance of equipping developers with the skills to design an abstraction layer that effectively addresses these limitations. Such a layer would manage two scenarios: 1) employing a single process for small-scale computations, and 2) utilizing multiple processes for large-scale computation.

4 References

[1] J. Darmawan, “Examples of OpenMPI and CUDA,” Jan. 2025, *GitHub*. Accessed: Jan. 07, 2025. [Online]. Available: https://github.com/jasonrichdarmawan/learn-openmpi

APPENDIX A

ALGORITHM OVERVIEW

Figure 6 illustrates the input data for the parallel matrix-vector multiplication process: a 3x4 input matrix and a 4x1 input vector. Table 3: Matrix-Vector Multiplication Process Distribution provides a detailed breakdown of how the matrix rows are distributed and computed across 1, 2, or 3 processes.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Figure 6: Example of a 3x4 Input Matrix and a 4x1 Input Vector Used for Parallel Distribution

Table 3: Matrix-Vector Multiplication Process Distribution

Process Count	1 Process	2 Processes	3 Processes
Process 0	$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 30 \\ 70 \\ 110 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 30 \\ 110 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = [30]$
Process 1	N/A	$\begin{bmatrix} 9 & 10 & 11 & 12 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = [70]$	$\begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = [70]$
Process 2	N/A	N/A	$\begin{bmatrix} 9 & 10 & 11 & 12 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = [110]$

APPENDIX B

MATRIX-VECTOR MULTIPLICATION BLOCK-ROW SOURCE CODE

Figure 7 demonstrates how to use the program. Figure 8 describes the implementation of the function for generating random matrix and vector with input rows and columns. Figure 9 outlines the file structure of the input matrix and vector. Figure 10 presents the implementation of the function for reading input matrix and vector from a file. Figure 11 describes the parallel matrix-vector distribution logic. Figure 12 describes the parallel matrix-vector multiplication logic.

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // assume both the matrix and the vector are input from a data file.
    double *mat, *vect;
    int rows, cols;
    if (rank == 0)
    {
        // Mat_vect_read_file("data.txt", &mat, &vect, &rows, &cols);
        rows = 8;
        cols = 1e6;
        srand(0);
        Mat_vect_create(&mat, &vect, rows, cols);
    }

    // Print the matrix
    // if (rank == 0)
    // {
    //     printf("%d | Matrix:\n", rank);
    //     for (int i = 0; i < rows; i++)
    //     {
    //         printf("%d | ", rank);
    //         for (int j = 0; j < cols; j++)
    //         {
    //             printf("%.1f ", mat[i * cols + j]);
    //         }
    //         printf("\n");
    //     }
    // }

    // use optimal number of processes
```

```c
    MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Comm newcomm = MPI_COMM_NULL;
    int color = (rank < rows) ? 0 : MPI_UNDEFINED;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &newcomm);
    if (newcomm == MPI_COMM_NULL)
    {
        MPI_Finalize();
        return 0;
    }

    // Start timing
    MPI_Barrier(newcomm);
    double start_time = MPI_Wtime();

    double *local_mat;
    int local_rows;
    MPI_Mat_vect_scatter_row(mat,
                             &local_mat,
                             &vect,
                             &local_rows,
                             &rows,
                             &cols,
                             newcomm);

    double *result_vect;
    MPI_Mat_vect_mult_row(local_mat,
                          vect,
                          &result_vect,
                          local_rows,
                          rows,
                          cols,
                          newcomm);

    // End timing
    MPI_Barrier(newcomm);
    double end_time = MPI_Wtime();
    if (rank == 0)
    {
        double elapsed_time = end_time - start_time;
        printf("Elapsed time: %.0f µs\n", elapsed_time * 1e6);
    }

    // Print the result vector
    // printf("%d | Result vector: ", rank);
    // for (int i = 0; i < rows; i++)
    // {
    //     printf("%.1f ", result_vect[i]);
    // }
```

```
    // printf("\n");

    // Clean up
    if (rank == 0)
    {
        free(mat);
    }
    free(local_mat);
    free(vect);
    free(result_vect);

    MPI_Comm_free(&newcomm);

    MPI_Finalize();
    return 0;
}
```

Figure 7 Parallel Matrix-Vector Multiplication Main Program

```
void Mat_vect_create(double **mat, double **vect, int rows, int cols)
{
    *mat = (double *)malloc(rows * cols * sizeof(double));
    *vect = (double *)malloc(cols * sizeof(double));
    for (int I = 0; I < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            (*mat)[I * cols + j] = rand() % 100 + 1;
        }
    }
    for (int I = 0; I < cols; i++)
    {
        (*vect)[i] = rand() % 100 + 1;
    }
}
```

Figure 8 Matrix-Vector Create Function

```
rows cols
matrix_00 matrix_01 matrix_02 matrix_03
matrix_10 matrix_11 matrix_12 matrix_13
matrix_20 matrix_21 matrix_22 matrix_23
vector_0 vector_1 vector_2 vector_3
```

Figure 9 File Structure of Input Matrix and Vector

```
void MPI_Mat_vect_read_file(const char *filename,
                            double **local_mat,
                            double **vect,
                            int *local_rows,
                            int *rows,
                            int *cols,
                            MPI_Comm comm)
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // assume both the matrix and the vector are input from a data file.
    Double *mat;
    if (rank == 0)
    {
        Mat_vect_read_file("data.txt", &mat, vect, rows, cols);
    }

    MPI_Mat_vect_scatter_row(mat, local_mat, vect, local_rows, rows, cols, comm);

    // Clean up
    if (rank == 0)
    {
        free(mat);
    }
}
```

Figure 10 Matrix-Vector File Reader Function

```
void MPI_Mat_vect_scatter_row(const double *mat,
                              double **local_mat,
                              double **vect,
                              int *local_rows,
                              const int *rows,
                              const int *cols,
                              MPI_Comm comm)
{
    int rank;
    MPI_Comm_rank(comm, &rank);

    MPI_Bcast((void *)rows, 1, MPI_INT, 0, comm);
    int size;
    MPI_Comm_size(comm, &size);
    *local_rows = (*rows) / size;
    int remainder = (*rows) % size;
    if (rank < remainder) {
        (*local_rows)++;
```

```
    }

    MPI_Bcast((void *)cols, 1, MPI_INT, 0, comm);
    int *sendcounts = (int *)malloc(size * sizeof(int));
    int *displs = (int *)malloc(size * sizeof(int));
    int offset = 0;
    // think of I as the rank
    for (int I = 0; I < size; i++) {
        sendcounts[i] = ((*rows) / size) * (*cols);
        if (I < remainder) {
            sendcounts[i] += (*cols);
        }
        displs[i] = offset;
        offset += sendcounts[i];
    }

    // matrices are distributed among the processes in block-row fashion
    *local_mat = (double *)malloc((*local_rows) * (*cols) * sizeof(double));
    MPI_Scatterv(mat, sendcounts, displs, MPI_DOUBLE,
                 *local_mat, (*local_rows) * (*cols), MPI_DOUBLE,
                 0, comm);

    // vectors are distributed among the processes as blocks
    if (rank != 0)
    {
        *vect = (double *)malloc((*cols) * sizeof(double));
    }
    MPI_Bcast(*vect, *cols, MPI_DOUBLE, 0, comm);

    // Clean up
    free(sendcounts);
    free(displs);
}
```

Figure 11 Parallel Matrix-Vector Distribution Function

```
void MPI_Mat_vect_mult_row(const double *local_mat,
                           const double *vect,
                           double **result_vect,
                           const int local_rows,
                           const int rows,
                           const int cols,
                           MPI_Comm comm)
{
    // Perform the matrix-vector multiplication
    double *local_result = (double *)malloc(local_rows * sizeof(double));
    for (int I = 0; I < local_rows; i++) {
```

```
        local_result[i] = 0.0;
        for (int j = 0; j < cols; j++) {
            local_result[i] += local_mat[I * cols + j] * vect[j];
        }
    }

    // Cases
    // Case 1: rows < size
    // Suppose the matrix is 3x4 and the size is 4
    // recvcounts = {1, 1, 1, 0}
    // displs_result = {0, 1, 2, 3}
    //
    // Case 2: rows = size
    // Suppose the matrix is 3x4 and the size is 3
    // recvcounts = {1, 1, 1}
    // displs_result = {0, 1, 2}
    //
    // Case 3: rows > size
    // Suppose the matrix is 3x4 and the size is 2
    // recvcounts = {2, 1}
    // displs_result = {0, 2}
    int size;
    MPI_Comm_size(comm, &size);
    int *recvcounts = (int *)malloc(size * sizeof(int));
    int *displs_result = (int *)malloc(size * sizeof(int));
    int remainder = rows % size;
    int offset = 0;
    // think of I as the rank
    for (int I = 0; I < size; i++) {
        recvcounts[i] = rows / size;
        if (I < remainder) {
            recvcounts[i]++;
        }
        displs_result[i] = offset;
        offset += recvcounts[i];
    }

    // the result vector c should be distributed among the processes as blocks.
    *result_vect = (double *)malloc(rows * sizeof(double));
    MPI_Allgatherv(local_result, local_rows, MPI_DOUBLE,
                   *result_vect, recvcounts, displs_result, MPI_DOUBLE,
                   comm);

    // Clean up
    free(local_result);
    free(recvcounts);
    free(displs_result);
}
```

Figure 12 Parallel Matrix-Vector Multiplication Block-Row Function

APPENDIX C

SCREENSHOTS OF THE PARALLEL MATRIX-VECTOR MULTIPLICATION PROGRAM RUNNING RESULTS

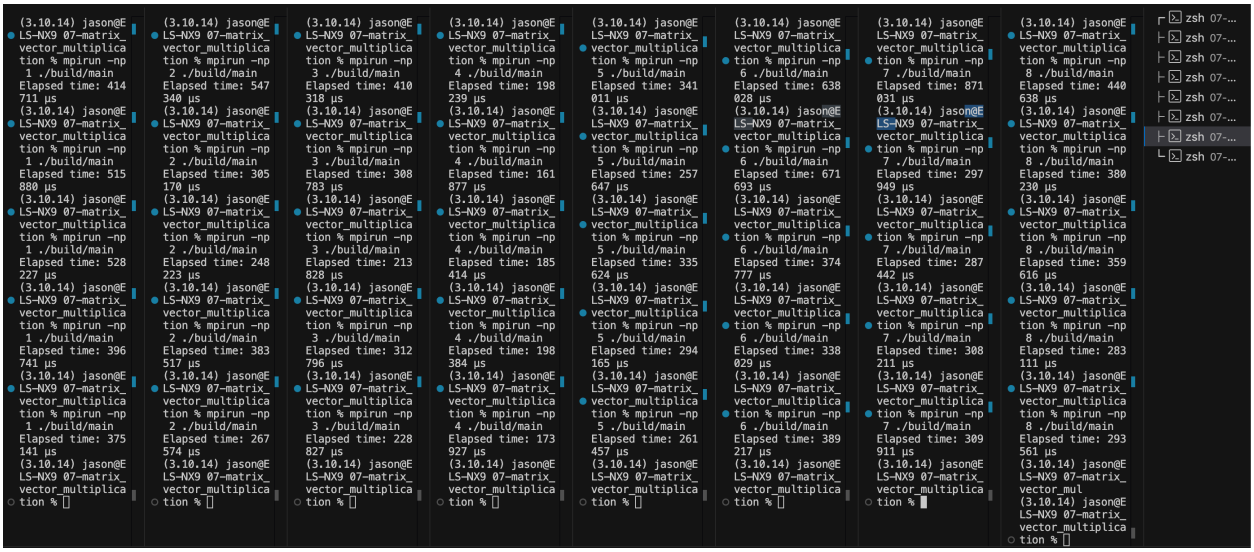

Figure 13 Screenshots of the Program's Running Results

APPENDIX D

PARALLEL FLOYD-WARSHALL ALGORITHM SOURCE CODE

Figures Figure 14 through Figure 17 provide a comprehensive description of various aspects of the program and its implementation. Specifically, Figure 14 illustrates how to use the program, while Figure 15 details the implementation of the function designed for generating a random graph based on input vertices and edges. Figure 16 highlights the implementation of the function responsible for flattening a 2D double structure. Finally, Figure 17 illustrates the workload distribution and the computation logic for determining the shortest paths between any two vertices.

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int vertices;
    double **graph;
    double *flattened_graph;
    srand(0);
    if (rank == 0)
    {
        vertices = 100;
        int edges = vertices * (vertices - 1);
        int isDirected = 1;
        if (GRAPH_random_graph(&graph, vertices, edges, isDirected) != 0)
        {
            printf("Error creating the graph\n");
            return -1;
        }
        flatten_2D_double((const double * const *)graph,
                          &flattened_graph,
                          vertices);
    }

    // use optimal number of processes
    MPI_Bcast((void *)&vertices, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Comm newcomm = MPI_COMM_NULL;
    int color = (rank < vertices) ? 0 : MPI_UNDEFINED;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &newcomm);
    if (newcomm == MPI_COMM_NULL)
    {
        MPI_Finalize();
```

```
        return 0;
    }

    // Start timing
    MPI_Barrier(newcomm);
    double start_time = MPI_Wtime();

    double *distance;
    if (MPI_GRAPH_shortest_paths_floyd_warshall(flattened_graph,
                                                vertices,
                                                &distance,
                                                newcomm) != 0)
    {
        printf("Error computing the shortest paths\n");
        return -1;
    }

    // End timing
    MPI_Barrier(newcomm);
    double end_time = MPI_Wtime();
    if (rank == 0)
    {
        double elapsed_time = end_time - start_time;
        printf("Elapsed time: %.0f µs\n", elapsed_time * 1e6);
    }

    // Free the graph
    if (rank == 0)
    {
        for (int I = 0; I < vertices; i++)
        {
            free(graph[i]);
        }
        free(graph);
        free(flattened_graph);
    }

    // Free the distance matrix
    free(distance);

    MPI_Comm_free(newcomm);

    MPI_Finalize();

    return 0;
}
```

Figure 14 Parallel Floyd-Warshall Algorithm Main Program

```c
// the shortcoming of this function is that it doesn't create multigraphs
// the experiment goal is to do parallel computing, so we don't focus on the graph creation
int GRAPH_random_graph(double ***graph, int vertices, int edges, int isDirected)
{
    // Check if the number of edges is valid
    if (edges > GRAPH_max_edges(vertices, isDirected))
    {
        printf("Too many edges for the number of vertices\n");
        return -1;
    }

    *graph = (double **)malloc(vertices * sizeof(double *));

    // Initialize the graph with infinity
    for (int i = 0; i < vertices; i++)
    {
        (*graph)[i] = (double *)malloc(vertices * sizeof(double));
        for (int j = 0; j < vertices; j++)
        {
            if (i == j)
            {
                (*graph)[i][j] = 0;
            }
            else
            {
                (*graph)[i][j] = DBL_MAX; // to represent the abscene of an edge between two vertices
            }
        }
    }

    // Create random edges
    for (int i = 0; i < edges; i++)
    {
        int source = rand() % vertices; // % vertices to make sure the edge is within the graph
        int destination = rand() % vertices;
        double weight = (double)(rand() % 100) + 1; // to avoid 0 weight, and this generates a random integer between 1 and 100

        // Avoid self loops and no duplicate edges / to guarantee the amount of edges created is the same as the input parameter
        if (source == destination || (*graph)[source][destination] != DBL_MAX)
        {
            i--; // Repeat this iteration
```

```
            continue;
        }

        // For directed graphs, you only set the weight for the edge from the source
to the destination
        (*graph)[source][destination] = weight;
        if (isDirected)
        {
            continue;
        }

        // For undirected graphs, you set the weight for both edges
        (*graph)[destination][source] = weight;
    }

    return 0;
}
```

Figure 15 Graph Create Function

```
void flatten_2D_double(const double * const *input, double **output, const int size)
{
    *output = (double *)malloc(size * size * sizeof(double));
    for (int row = 0; row < size; row++)
    {
        for (int col = 0; col < size; col++)
        {
            (*output)[row * size + col] = input[row][col];
        }
    }
}
```

Figure 16 2D Flatten Function

```
int MPI_GRAPH_shortest_paths_floyd_warshall(const double *graph,
                                            const int vertices,
                                            double **distance,
                                            MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    int local_rows = vertices / size;
    int remainder = vertices % size;
    if (rank < remainder)
    {
        local_rows++;
```

```
    }

    int *sendcounts = (int *)malloc(size * sizeof(int));
    int *displs = (int *)malloc(size * sizeof(int));

    int offset = 0;
    for (int i = 0; i < size; i++)
    {
        sendcounts[i] = vertices / size * vertices;
        if (i < remainder)
        {
            sendcounts[i] += vertices;
        }
        displs[i] = offset;
        offset += sendcounts[i];
    }

    // graph are distributed among the process in block-row fashion
    double *local_graph = (double *)malloc(local_rows * vertices * sizeof(double));
    MPI_Scatterv(graph, sendcounts, displs, MPI_DOUBLE,
                 local_graph, local_rows * vertices, MPI_DOUBLE,
                 0, comm);

    // Initialize the dist matrix with the same values as the graph
    double *local_distance = (double *)malloc(local_rows * vertices *
sizeof(double));
    memcpy(local_distance, local_graph, local_rows * vertices * sizeof(double));

    // k is the intermediate vertex
    // i is the source vertex
    // j is the destination vertex
    int max_rows_per_process = (vertices + size - 1) / size; // equivalent to
ceil(vertices / size), but ceil is for double
    for (int k = 0; k < vertices; k++)
    {
        // Broadcast the k-th row of the distance vector
        double *k_row = (double *)malloc(vertices * sizeof(double));
        int k_owner;
        if (remainder == 0) {
            k_owner = k / max_rows_per_process;
        } else
        {
            k_owner = (k < remainder * max_rows_per_process) ?
                      k / max_rows_per_process :
                      (k - remainder) / (max_rows_per_process - 1);
        }

        if (rank == k_owner)
```

```
        {
            int k_relative_row;
            if (remainder == 0)
            {
                k_relative_row = max_rows_per_process == 1 ?
                                 0 :
                                 k % max_rows_per_process;
            }
            else
            {
                k_relative_row = (k < remainder * max_rows_per_process) ?
                                 k % max_rows_per_process :
                                 (k - remainder) % (max_rows_per_process - 1);
            }
            memcpy(k_row, &local_distance[k_relative_row * vertices], vertices *
sizeof(double));
        }
        MPI_Bcast(k_row, vertices, MPI_DOUBLE, k_owner, comm);

        for (int i = 0; i < local_rows; i++)
        {
            for (int j = 0; j < vertices; j++)
            {
                // performance optimization
                // Skip if there is no edge between i and k or between k and j
                if (local_distance[i * vertices + k] == DBL_MAX || k_row[j] ==
DBL_MAX)
                {
                    continue;
                }

                // Skip if the distance between i and j is already shorter than the
distance between i and k + k and j
                int b = local_distance[i * vertices + k] + k_row[j];
                if (local_distance[i * vertices + j] < b)
                {
                    continue;
                }

                local_distance[i * vertices + j] = b;
            }
        }

        // Clean up
        free(k_row);
    }

    // the result vector c should be distributed among the processes as blocks.
```

```
    *distance = (double *)malloc(vertices * vertices * sizeof(double));
    MPI_Allgatherv(local_distance, local_rows * vertices, MPI_DOUBLE,
                   *distance, sendcounts, displs, MPI_DOUBLE,
                   comm);

    // Clean up
    free(sendcounts);
    free(displs);
    free(local_graph);
    free(local_distance);

    return 0;
}
```

Figure 17 Parallel Floyd-Warshall Algorithm Distribution and Computation Function

APPENDIX E

SCREENSHOTS OF THE SHORTEST PATHS CALCULATION BETWEEN ANY TWO VERTICES PROGRAM RUNNING RESULTS

Table 4 Experiments with 5 Vertices and 20 Edges

Process Count	Trial 1 (μs)	Trial 2 (μs)	Trial 3 (μs)	Trial 4 (μs)	Trial 5 (μs)	Average Time (μs)
1	7	7	11	6	6	7,4
5	41	27	35	31	34	33,6
7	46	61	53	38	47	49

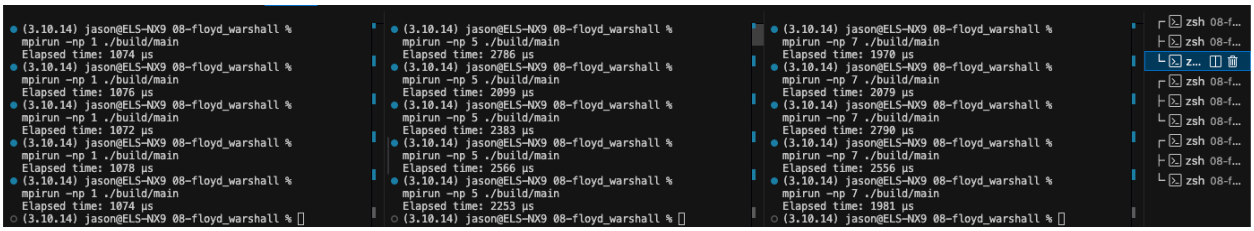

Figure 18 Screenshots Experiments with 5 Vertices and 20 Edges

Table 5 Experiments with 50 vertices and 2450 edges

Process Count	Trial 1 (μs)	Trial 2 (μs)	Trial 3 (μs)	Trial 4 (μs)	Trial 5 (μs)	Average Time (μs)
1	1074	1076	1072	1078	1074	1074,8
5	2786	2099	2383	2566	2253	2417,4
7	1970	2079	2790	2556	1981	2275,2

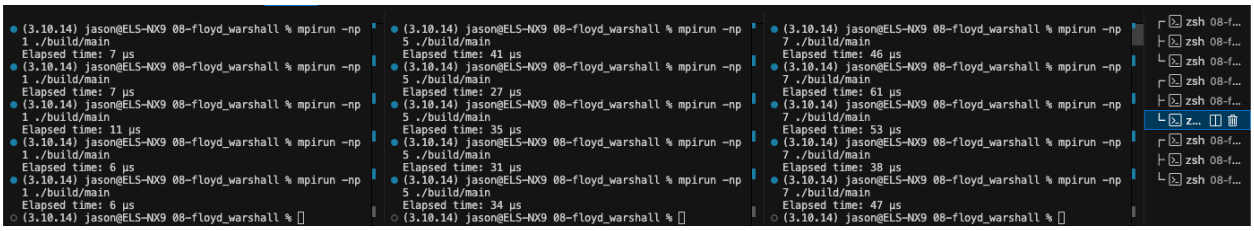

Figure 19 Screenshots Experiments with 50 Vertices and 2450 Edges

Table 6 Experiments with 500 vertices and 249500 edges

Process Count	Trial 1 (μs)	Trial 2 (μs)	Trial 3 (μs)	Trial 4 (μs)	Trial 5 (μs)	Average Time (μs)
1	784415	781567	780564	781210	785751	782701,4
5	292469	284934	292750	281938	299217	290261,6
7	263285	250846	234056	243322	239578	246217,4

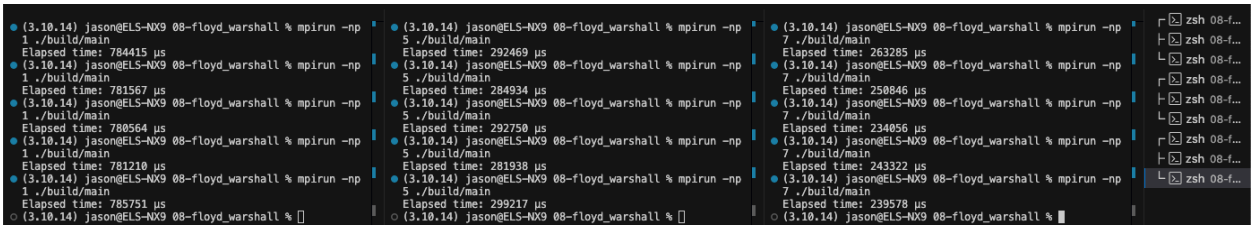

Figure 20 Experiments with 500 Vertices and 249500 Edges

APPENDIX F

ROWS OWNER MAPPER FUNCTION

```
    // Determine the owner process and the relative row for each vertex
    int (*rows_owner)[2] = (int (*)[2])malloc(vertices * sizeof(int[2]));
    for (int i = 0; i < vertices; i++) {
        int owner_process = 0;
        // displs[owner_process] / vertices is the first row of the owner process
        // sendcounts[owner_process] / vertices is the number of rows of the owner
process
        while (i >= displs[owner_process] / vertices + sendcounts[owner_process] /
vertices) {
            owner_process++;
        }
        rows_owner[i][0] = owner_process; // process rank
        rows_owner[i][1] = i - displs[owner_process] / vertices; // relative row
within the process
    }
```

Figure 21 Rows Owner Mapper Function