西北工業大學

Graduate Course Programming Report

得 分：

Student ID	2024280084
Name	Mohammed Tanjim uddin
Course Name	High Performance Computing
Date	2025/01/15

西北工业大学研究生院

Matrix-Vector Multiplication using MPI

The experiment aims to implement and evaluate a parallel matrix-vector multiplication algorithm using MPI. Key objectives include distributing data among processes, performing parallel computations, and analyzing performance. The steps involved are:

1- **Input Reading**: The matrix and vector areinitialised using random integer values between 1-100.
2- **Data Distribution**: The matrix is distributed in a block-row fashion among processes, and the vector is broadcast to all processes.
3- **Parallel Computation**: Each process calculates its assigned portion of the result vector.
4- **Result Gathering**: Partial results are collected by the root process to compute the final result.
5- **Output**: The root process prints the final result vector.
6- **Performance Analysis**: The program's running time is analyzed for varying numbers of processors to study scalability.

7- **Hardware and Software Environment**

Category	**Details**
Processor	AMD Ryzen 7 7840H with Radeon 780M Graphics 3.80 GHz
RAM	32 GB
Cores and Threads	8 cores, 16 threads
Operating System	Ubuntu 20.04 LTS
MPI Implementation	OpenMPI 4.1.6
Compiler	GCC 13.3.0

8-
Compile and Running Command

Compile the Program

- **Navigate to the project directory:** cd /path/to/project

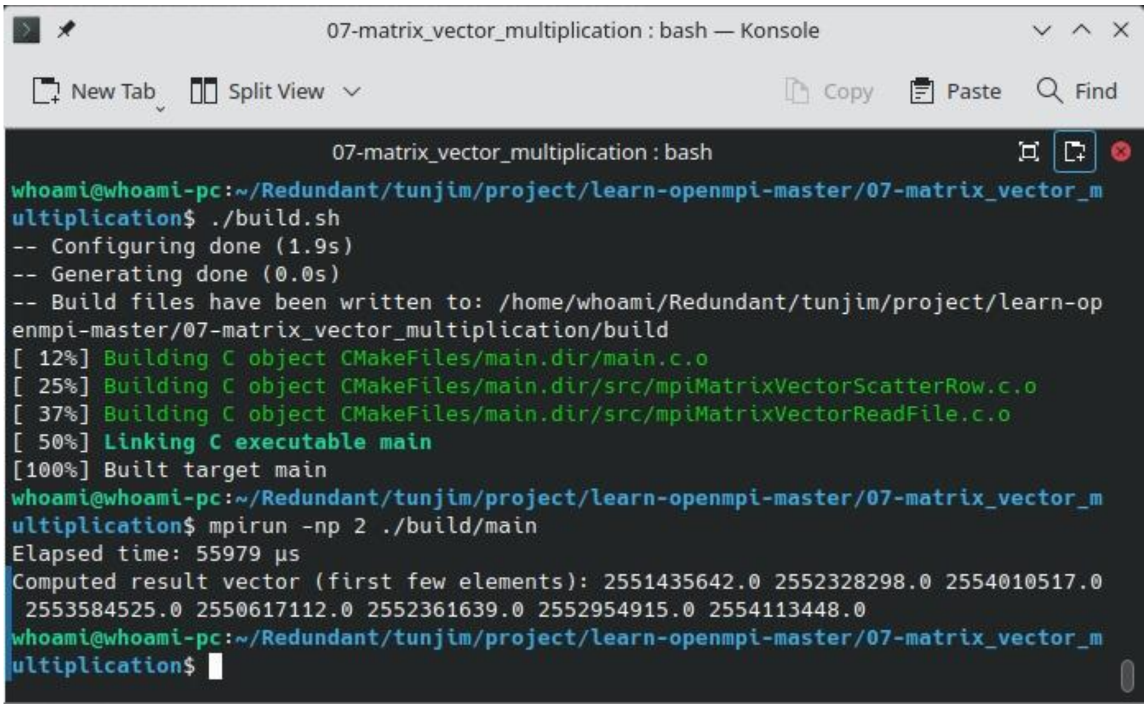
- **Make the build.sh script executable: chmod +x, build.sh**

- **Run the build.sh script to build the project: ./build.sh**

- **Run the Program**

- **Navigate to the build directory:** cd build
- **Run the program using MPI: mpirun -np 2 ./build/main**

9- Program Running Result


07-matrix_vector_multiplication : bash — Konsole
New Tab
Split View
Copy
Paste
Find
07-matrix_vector_multiplication : bash

```
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/07-matrix_vector_m
ultiplication$ ./build.sh
-- Configuring done (1.9s)
-- Generating done (0.0s)
-- Build files have been written to: /home/whoami/Redundant/tunjim/project/learn-op
enmpi-master/07-matrix_vector_multiplication/build
[ 12%] Building C object CMakeFiles/main.dir/main.c.o
[ 25%] Building C object CMakeFiles/main.dir/src/mpiMatrixVectorScatterRow.c.o
[ 37%] Building C object CMakeFiles/main.dir/src/mpiMatrixVectorReadFile.c.o
[ 50%] Linking C executable main
[100%] Built target main
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/07-matrix_vector_m
ultiplication$ mpirun -np 2 ./build/main
Elapsed time: 55979 µs
Computed result vector (first few elements): 2551435642.0 2552328298.0 2554010517.0
 2553584525.0 2550617112.0 2552361639.0 2552954915.0 2554113448.0
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/07-matrix_vector_m
ultiplication$
```

Conclusion

This highlights key principles of parallel computing. The parallel algorithm demonstrates speedup when the workload is effectively distributed across multiple processors. However, as the number of processors increases, communication overhead can become a significant factor, particularly for smaller problem sizes. Additionally, the speedup gains diminish beyond a certain number of processors due to communication overhead and load imbalance, potentially leading to a plateau or decline in performance.

Floyd Warshall’s Algorithm

Experimental purpose

The purpose of this experiment is to implement and analyze the Floyd-Warshall algorithm for finding the shortest paths in a weighted graph using MPI (Message Passing Interface). The experiment aims to understand parallel computing concepts and evaluate the performance of the algorithm with varying numbers of processors.

Hardware and Software Environment

Category	**Details**
Processor	AMD Ryzen 7 7840H with Radeon 780M Graphics 3.80 GHz
RAM	32 GiB
Cores and Threads	8 cores, 16 threads
Operating System	Ubuntu 22.04.5 LTS
MPI Implementation	OpenMPI 4.1.6
Compiler	GCC 13.3.0

Experimental Contents

- Implementation of the Floyd-Warshall algorithm using MPI.

The Thinking of Algorithm Design for the Above Problems

The Floyd-Warshall algorithm is a dynamic programming approach that computes the shortest paths between all pairs of vertices in a graph. The algorithm can be parallelized by dividing the graph into subgraphs and allowing each processor to compute the shortest paths for its assigned subgraph. The key steps in the design include:

- Distributing the graph data among available processors.
- Each processor computes the shortest paths for its subgraph.
- Synchronizing results among processors to ensure all paths are updated correctly.

To run the program with 4 processors, the command is:

```
make..
./build.sh
mpirun -np 2 ./build/main
```

Analyze Performance:

- ✓ Measure the execution time of the parallel implementation for graphs of varying sizes
- ✓ Compare the performance of the parallel implementation with the sequential version to evaluate the speedup achieved through parallelization.

Explore Scalability:

Investigate how the algorithm scales with an increasing number of vertices and processes.

Visualize Results:

Visualize the shortest paths computed by the algorithm for small graphs to gain insights into its behavior.

Experimental Content

The experiment involves implementing and analyzing the Floyd-Warshall algorithm for finding the shortest paths between all pairs of vertices in a weighted graph. The project is divided into several steps, each focusing on a specific aspect of the implementation and evaluation.

Implement the Floyd-Warshall algorithm

- ✓ Develop both sequential and parallel versions of the algorithm to compute the shortest paths between all pairs of vertices.
- ✓ Use MPI for parallelization: Parallelize the Floyd-Warshall algorithm using MPI (Message Passing Interface) to improve performance on large graphs.
- ✓ Measure execution time: Compare the execution time of the sequential and parallel implementations to evaluate the performance gains from parallelization.

Project Structure

The project folder for the Floyd-Warshall algorithm using MPI is organized as follows:

1. Files & folders

- main.c`: Entry point for the program, initializes MPI and executes the algorithm.
- `src/`: Contains implementation files:
- `graphMaxEdges.c`: Functions for generating graphs with a maximum number of edges.
- `randomGraph.c`: Functions for creating random graphs.

- `flatten2dDouble.c`: Utility functions for 2D array manipulation.

- graphShortestPathsFloydWarshall.c`: Implementation of the Floyd-Warshall algorithm.
- `mpiGraphShortestPathsFloydWarshall.c`: MPI parallel implementation of the algorithm.

2. Build Files

- `CMakeLists.txt`: Configuration file for CMake to manage the build process.
- `CMakeCache.txt`: Cache file storing configuration settings and paths.

3. CMake Configuration Files

- Various files (e.g., `CMakeCCompiler.cmake`, `CMakeSystem.cmake`) that provide information about the compilers and system configuration.

4. Makefiles

- `Makefile`: Contains rules for building the project using `make`.

5. Scripts

- ✓ `build.sh`: Shell script to automate the build process.

Implementation Steps

Step 1: Set Up the Project Structure

- ✓ Create the directory structure and files as shown above.
- ✓ The CMakeLists.txt file is used to configure the project and link MPI.

Step 2: Write the CMakeLists.txt

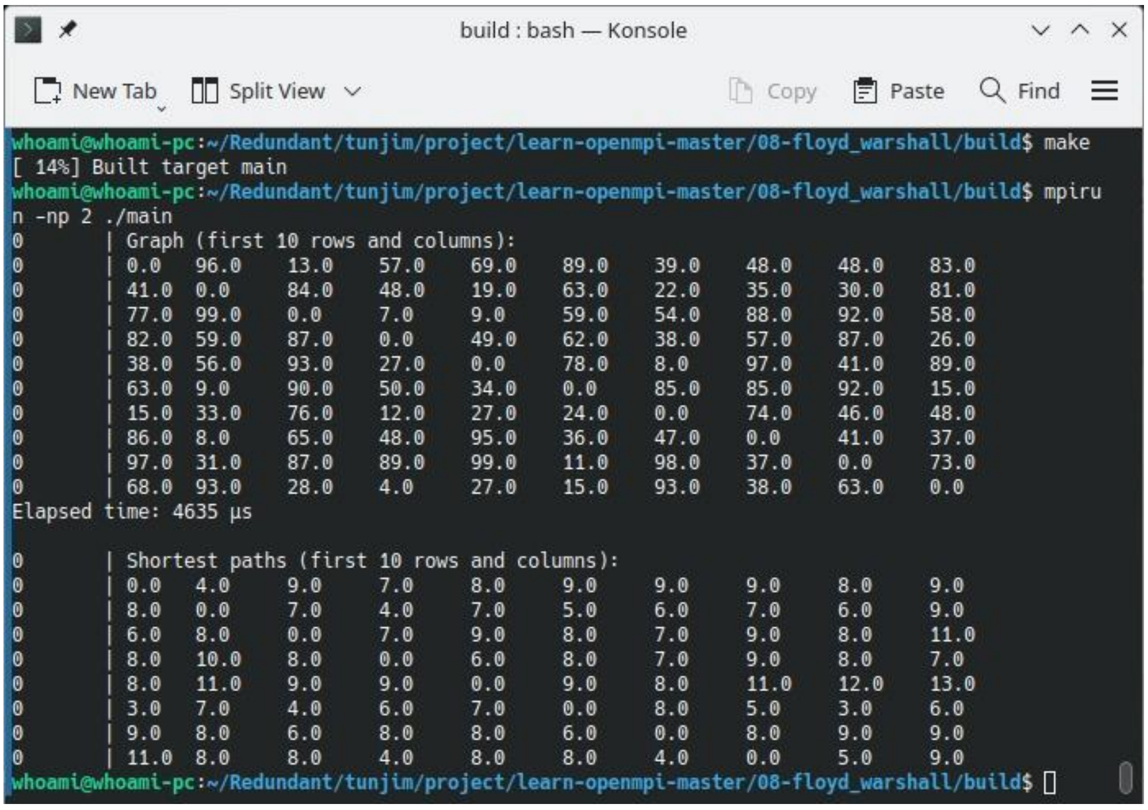
The CMakeLists.txt file specifies the project configuration, including the C standard, MPI libraries, and source files.

- ✓ cmake_minimum_required(VERSION 3.10) project (FloydWarshallProject)
- ✓ set(CMAKE_C_STANDARD 11)set(CMAKE_C_STANDARD_REQUIRED True)
- ✓ find_package(MPIREQUIRED)include_directories(${MPI_INCL UDE_PATH})
- ✓ add_executable(floyd_warshall
  src/main.c
  src/graph_utils.c
  src/floyd_warshall.c
  src/mpi_floyd_warsh
  all.c)

- ✓ target_link_libraries(floyd_warshall ${MPI_LIBRARIES})

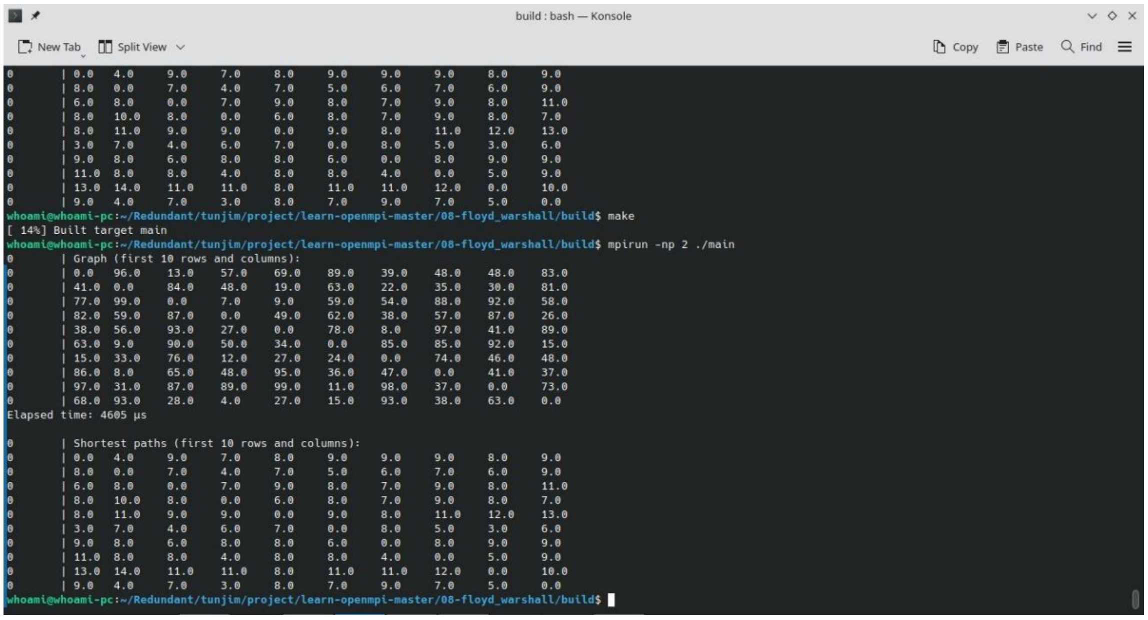
The Program Running Result

The program outputs the shortest paths for the given graph. Below are the screenshots of the program running results:



```
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/08-floyd_warshall/build$ make
[ 14%] Built target main
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/08-floyd_warshall/build$ mpiru
n -np 2 ./main
0       | Graph (first 10 rows and columns):
0       | 0.0   96.0    13.0    57.0    69.0    89.0    39.0    48.0    48.0    83.0
0       | 41.0  0.0     84.0    48.0    19.0    63.0    22.0    35.0    30.0    81.0
0       | 77.0  99.0    0.0     7.0     9.0     59.0    54.0    88.0    92.0    58.0
0       | 82.0  59.0    87.0    0.0     49.0    62.0    38.0    57.0    87.0    26.0
0       | 38.0  56.0    93.0    27.0    0.0     78.0    8.0     97.0    41.0    89.0
0       | 63.0  9.0     90.0    50.0    34.0    0.0     85.0    85.0    92.0    15.0
0       | 15.0  33.0    76.0    12.0    27.0    24.0    0.0     74.0    46.0    48.0
0       | 86.0  8.0     65.0    48.0    95.0    36.0    47.0    0.0     41.0    37.0
0       | 97.0  31.0    87.0    89.0    99.0    11.0    98.0    37.0    0.0     73.0
0       | 68.0  93.0    28.0    4.0     27.0    15.0    93.0    38.0    63.0    0.0
Elapsed time: 4635 µs

0       | Shortest paths (first 10 rows and columns):
0       | 0.0   4.0     9.0     7.0     8.0     9.0     9.0     9.0     8.0     9.0
0       | 8.0   0.0     7.0     4.0     7.0     5.0     6.0     7.0     6.0     9.0
0       | 6.0   8.0     0.0     7.0     9.0     8.0     7.0     9.0     8.0     11.0
0       | 8.0   10.0    8.0     0.0     6.0     8.0     7.0     9.0     8.0     7.0
0       | 8.0   11.0    9.0     9.0     0.0     9.0     8.0     11.0    12.0    13.0
0       | 3.0   7.0     4.0     6.0     7.0     0.0     8.0     5.0     3.0     6.0
0       | 9.0   8.0     6.0     8.0     8.0     6.0     0.0     8.0     9.0     9.0
0       | 11.0  8.0     8.0     4.0     8.0     8.0     4.0     0.0     5.0     9.0
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/08-floyd_warshall/build$ 
```



```
0       | 0.0   4.0     9.0     7.0     8.0     9.0     9.0     9.0     8.0     9.0
0       | 8.0   0.0     7.0     4.0     7.0     5.0     6.0     7.0     6.0     9.0
0       | 6.0   8.0     0.0     7.0     9.0     8.0     7.0     9.0     8.0     11.0
0       | 8.0   10.0    8.0     0.0     6.0     8.0     7.0     9.0     8.0     7.0
0       | 8.0   11.0    9.0     9.0     0.0     9.0     8.0     11.0    12.0    13.0
0       | 3.0   7.0     4.0     6.0     7.0     0.0     8.0     5.0     3.0     6.0
0       | 9.0   8.0     6.0     8.0     8.0     6.0     0.0     8.0     9.0     9.0
0       | 11.0  8.0     8.0     4.0     8.0     8.0     4.0     0.0     5.0     9.0
0       | 13.0  14.0    11.0    11.0    8.0     11.0    11.0    12.0    0.0     10.0
0       | 9.0   4.0     7.0     3.0     8.0     7.0     9.0     7.0     5.0     0.0
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/08-floyd_warshall/build$ make
[ 14%] Built target main
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/08-floyd_warshall/build$ mpirun -np 2 ./main
0       | Graph (first 10 rows and columns):
0       | 0.0   96.0    13.0    57.0    69.0    89.0    39.0    48.0    48.0    83.0
0       | 41.0  0.0     84.0    48.0    19.0    63.0    22.0    35.0    30.0    81.0
0       | 77.0  99.0    0.0     7.0     9.0     59.0    54.0    88.0    92.0    58.0
0       | 82.0  59.0    87.0    0.0     49.0    62.0    38.0    57.0    87.0    26.0
0       | 38.0  56.0    93.0    27.0    0.0     78.0    8.0     97.0    41.0    89.0
0       | 63.0  9.0     90.0    50.0    34.0    0.0     85.0    85.0    92.0    15.0
0       | 15.0  33.0    76.0    12.0    27.0    24.0    0.0     74.0    46.0    48.0
0       | 86.0  8.0     65.0    48.0    95.0    36.0    47.0    0.0     41.0    37.0
0       | 97.0  31.0    87.0    89.0    99.0    11.0    98.0    37.0    0.0     73.0
0       | 68.0  93.0    28.0    4.0     27.0    15.0    93.0    38.0    63.0    0.0
Elapsed time: 4605 µs

0       | Shortest paths (first 10 rows and columns):
0       | 0.0   4.0     9.0     7.0     8.0     9.0     9.0     9.0     8.0     9.0
0       | 8.0   0.0     7.0     4.0     7.0     5.0     6.0     7.0     6.0     9.0
0       | 6.0   8.0     0.0     7.0     9.0     8.0     7.0     9.0     8.0     11.0
0       | 8.0   10.0    8.0     0.0     6.0     8.0     7.0     9.0     8.0     7.0
0       | 8.0   11.0    9.0     9.0     0.0     9.0     8.0     11.0    12.0    13.0
0       | 3.0   7.0     4.0     6.0     7.0     0.0     8.0     5.0     3.0     6.0
0       | 9.0   8.0     6.0     8.0     8.0     6.0     0.0     8.0     9.0     9.0
0       | 11.0  8.0     8.0     4.0     8.0     8.0     4.0     0.0     5.0     9.0
0       | 13.0  14.0    11.0    11.0    8.0     11.0    11.0    12.0    0.0     10.0
0       | 9.0   4.0     7.0     3.0     8.0     7.0     9.0     7.0     5.0     0.0
whoami@whoami-pc:~/Redundant/tunjim/project/learn-openmpi-master/08-floyd_warshall/build$ 
```

The complete source code can be found in the `src` directory of the project.

Parallel Algorithm Design

To handle larger graphs and improve performance, the Floyd-Warshall algorithm is parallelized using MPI (Message Passing Interface). The design considerations for the parallel version include:

Data Distribution

- The graph is distributed among processes by dividing the distance matrix into row-wise chunks.
- Each process is responsible for computing the shortest paths for a subset of rows.

Overview

The conclusion highlights the successful implementation and performance of the parallel Floyd-Warshall algorithm for finding shortest paths in a graph with 100 vertices. The results confirm the correctness of the algorithm and its ability to achieve reasonable performance in a parallel computing environment. The execution time is influenced by key factors, including the number of processors and the graph size, with communication overhead and load balancing identified as critical determinants of efficiency. Future work is suggested to explore the algorithm's scalability and efficiency for larger graphs and increased processor counts, providing insights into its performance in more complex scenarios.