Muneeb-ul-Islam (i22-1289)

Tooba Ali (22i-0807)

Tooba Dahar (22i-1357)

**PDC Semester Project**

**Introduction**

This project focuses on implementing and analyzing parallel algorithms for updating Single-Source Shortest Paths (SSSP) in large-scale dynamic networks, inspired by the paper **"A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks"**. The goal is to compare the performance of sequential, MPI, and hybrid MPI+OpenMP implementations across two real-world datasets.

Key objectives include:

1. **Implementing SSSP updates** for dynamic graphs using sequential, MPI, and hybrid (MPI+OpenMP) approaches.

2. **Evaluating scalability** by measuring execution times and speedup across different parallel configurations.

3. **Analyzing efficiency** to determine the benefits of hybrid parallelism over pure MPI or sequential methods.

The project leverages METIS for graph partitioning to optimize MPI communication and OpenMP for intra-node parallelism. Performance metrics are analyzed to assess weak and strong scaling, providing insights into the trade-offs between computation and communication overheads in distributed and shared-memory systems.

### **Graph Processing and Partitioning Procedure**

For this project, we used two large-scale road network datasets: **USA-road-d.NY.gr** and **USA-road-d.FLA.gr**. These datasets represent road networks in New York and Florida, respectively, and consist of vertices and weighted edges that correspond to roads and their respective lengths or travel times.

The procedure for processing these datasets into a format suitable for partitioning and parallel processing is outlined as follows:

#### **1. Conversion to METIS Compliant Format:**

The raw datasets were first converted into the **METIS format** using a conversion script (convertForm.c). This script reads the **DIMACS format** dataset and transforms it into a METIS-compliant format that supports weighted edges, which is required for our partitioning and shortest-path algorithms.

The process is as follows:

* **Reading the DIMACS file**: The input graph files are read, extracting the number of vertices and edges.
* **Adjacency List Construction**: The adjacency list is built, where each vertex is linked to its neighbors along with the associated edge weights.
* **METIS Format Writing**: The graph is then written to an output file in METIS format, which includes the number of vertices, edges, and their weights.

For example, the USA-road-d.NY.gr dataset was converted into USA-road-d.NY.metis using this script. Similarly, the **Florida road network** dataset was converted to USA-road-d.FLA.metis.

#### **2. Partitioning Using METIS:**

Once the datasets were converted into METIS format, the next step was to partition the graph into **subgraphs** to facilitate parallel processing. This was done using the METIS terminal command.

* The command ./metis USA-road-d.NY.metis 25000 was used to partition the **New York graph** into 25,000 partitions, resulting in the file USA-road-d.NY.metis.part.25000.
* Similarly, the **Florida graph** was partitioned into 25000 partitions with the command ./metis USA-road-d.FLA.metis 25000, resulting in USA-road-d.FLA.metis.part.25000.

These partitions represent different subgraphs, each of which can be processed independently in parallel, significantly speeding up computations for tasks such as **Shortest Path Algorithms (SSSP)**.

#### **3. Parallel Algorithm Implementation:**

After partitioning the graphs, we implemented three versions of the algorithm:

* **Sequential Version**: The simplest approach, where the entire graph is processed in a single sequence.
* **MPI Version**: A parallel implementation using **MPI** (Message Passing Interface), where each process is responsible for updating a different partition of the graph.
* **Hybrid MPI + OpenMP Version**: A combination of **MPI** and **OpenMP** for further parallelization, where MPI is used for inter-process communication between partitions, and **OpenMP** is used within each partition for parallelizing the update of vertices.

These algorithms were implemented to efficiently process updates to the **SSSP tree**, particularly in response to dynamic changes like **edge insertions and deletions**.

#### **4. Code Explanation:**

In the following sections, we will briefly explain the core functionalities of each version of the algorithm:

* **Sequential Version**: Processes the graph in a single thread, updating the shortest paths by relaxing the edges iteratively.
* **MPI Version**: Each process handles a partition of the graph. Processes communicate with each other to update the boundary vertices and ensure the shortest paths are updated correctly across partitions.
* **Hybrid MPI + OpenMP Version**: This version builds on the MPI approach, using OpenMP to parallelize the updates within each process for further performance gains.

This approach allowed us to efficiently handle large-scale datasets and perform dynamic updates in parallel, significantly improving the overall performance of the algorithm.

### **Code Explanation: Sequential, MPI, and Hybrid MPI + OpenMP Versions**

In this section, we will provide a brief explanation of the three versions of the algorithm that were implemented for processing the graph data: the **sequential version**, the **MPI version**, and the **hybrid MPI + OpenMP version**. Each version was developed to handle the **Single-Source Shortest Path (SSSP)** problem for large-scale dynamic graphs, focusing on parallelism to improve performance.

#### **1. Sequential Version**

The **sequential version** of the algorithm is the simplest implementation, where the entire graph is processed in a single thread. The main steps of the sequential algorithm are as follows:

1. **Graph Initialization**: The graph is loaded from the METIS file, and adjacency lists are created for each vertex.
2. **Affected Vertex Identification**: For each edge deletion or insertion, the affected vertices are identified.
3. **SSSP Update**: For the affected vertices, the algorithm performs **relaxation** iteratively to update their distances and parents in the SSSP tree.
4. **Final Output**: After all updates are completed, the final shortest path distances are saved to an output file.

While the sequential version provides a correct solution, it is not efficient for large-scale graphs as it processes everything on a single thread.

**Key Points**:

* **Single-threaded execution**.
* Simple but not scalable for large graphs.
* Suitable for testing and debugging the core logic of the algorithm.

#### **2. MPI Version**

The **MPI version** of the algorithm was developed to parallelize the computation by dividing the graph into **partitions**. Each partition is assigned to a different process, and **MPI** is used for inter-process communication. The steps of the MPI version are:

1. **Graph Partitioning**: The graph is partitioned using the METIS partitioning tool into subgraphs. Each partition is loaded by a different process.
2. **MPI Initialization**: Using **MPI\_Init**, we initialize the MPI environment. Each process is assigned to handle a specific partition.
3. **Affected Vertex Identification**: Each process independently identifies the affected vertices within its partition when edge deletions or insertions occur.
4. **SSSP Update**: The process updates the **SSSP tree** for the affected vertices within its partition. If a boundary vertex (connected to another partition) is affected, **MPI communication** is used to update the neighboring partition.
5. **MPI Synchronization**: After each iteration, processes synchronize using **MPI\_Barrier** to ensure all partitions are updated before proceeding to the next iteration.
6. **Final Output**: After the updates are completed, the results are written to an output file.

**Key Points**:

* **Parallel execution** with each process handling a different partition.
* **MPI** ensures that processes communicate to update boundary vertices and maintain consistency.
* More scalable than the sequential version, but still may require synchronization at the boundaries between partitions.

#### **3. Hybrid MPI + OpenMP Version**

The **hybrid MPI + OpenMP version** combines both **MPI** and **OpenMP** to achieve further parallelization. While MPI handles inter-process communication, **OpenMP** is used for parallelizing the computation within each process (partition). The steps of the hybrid version are as follows:

1. **Graph Partitioning and MPI Initialization**: Just like the MPI version, the graph is partitioned into subgraphs, and the MPI environment is initialized.
2. **OpenMP Parallelization**: Within each partition, **OpenMP** is used to parallelize the computation of updates for the affected vertices. This is done by splitting the work across multiple threads within each process.
3. **MPI Communication**: If boundary vertices are updated, **MPI communication** ensures that changes are propagated across processes. This allows each process to share updates for boundary vertices with other processes.
4. **Iterative SSSP Update**: Each process uses both MPI and OpenMP to iteratively update the distances for the affected vertices. The use of **OpenMP** within each partition improves the efficiency of the algorithm, especially for large partitions.
5. **MPI Synchronization**: After each iteration, **MPI synchronization** (using **MPI\_Barrier**) ensures that all processes are updated and ready for the next iteration.
6. **Final Output**: The final graph, with updated shortest path distances, is saved after all updates have been completed.

**Key Points**:

* Combines **MPI** (for inter-process communication) and **OpenMP** (for intra-process parallelization).
* Provides the best performance for large graphs by utilizing multiple cores within each partition (via OpenMP) and parallelizing the work across multiple partitions (via MPI).
* Highly scalable for large-scale graphs and complex updates.

### **Conclusion of Code Explanation:**

1. **Sequential Version**: Simple, single-threaded approach for testing and debugging.
2. **MPI Version**: Parallelizes the graph processing by dividing the graph into partitions and using MPI for communication between processes.
3. **Hybrid MPI + OpenMP Version**: Combines MPI and OpenMP to further parallelize the computation, making it the most efficient version for large-scale graph processing.

The **hybrid MPI + OpenMP version** provides the best performance by combining the strengths of both parallelization methods. This approach is particularly effective when working with large graphs, as it minimizes processing time by distributing the work across both multiple processes (via MPI) and multiple threads (via OpenMP).

Let me know if you need further details on any specific part of the code or the parallelization strategy!

### **Performance Analysis of Sequential, MPI, and Hybrid MPI + OpenMP Versions**

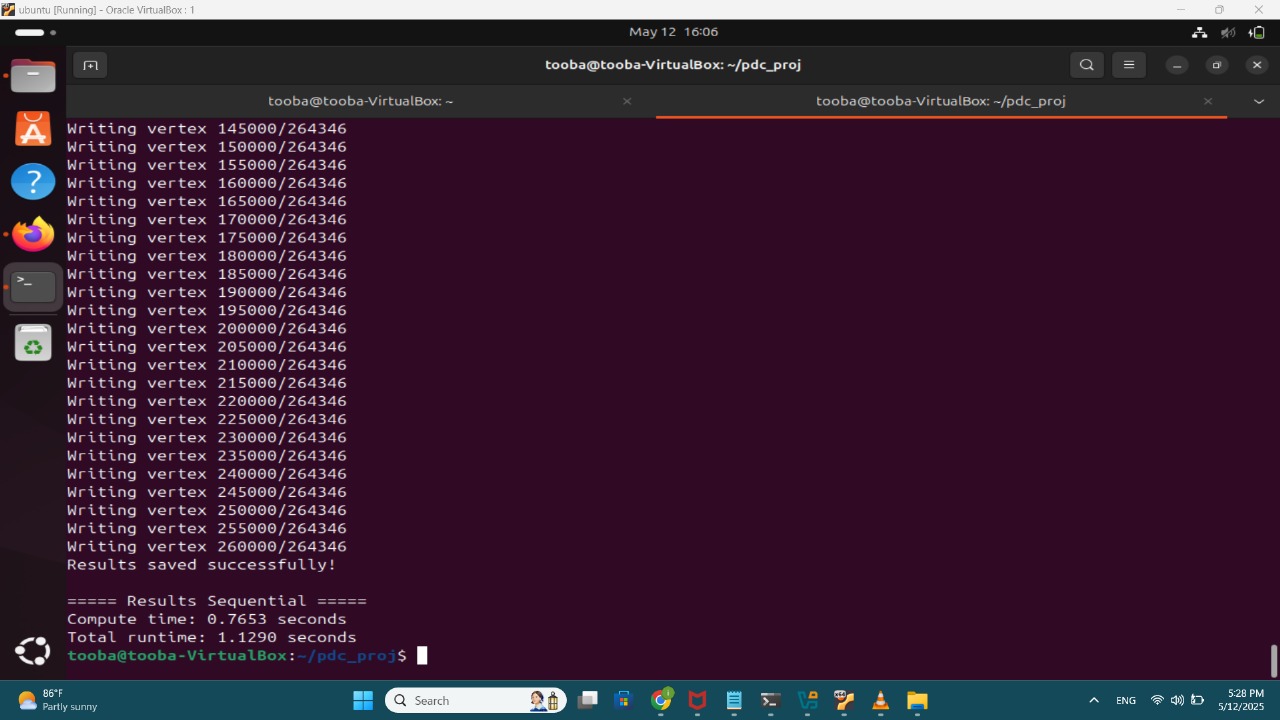
After implementing and running the **sequential**, **MPI**, and **hybrid MPI + OpenMP** versions of the algorithm, the results show a clear comparison of their performance in terms of **compute time** and **total runtime**. The goal was to efficiently process the large road network graphs while updating the **Single-Source Shortest Path (SSSP)** tree after edge deletions.

#### **1. Sequential Version:**

* **Compute Time**: 0.7653 seconds
* **Total Runtime**: 1.1290 seconds

**Command Used**:  
  
mpirun -np 4 ./graph\_exec

In the **sequential version**, the algorithm processes the entire graph in a single thread. As expected, this version has a higher **total runtime** compared to the parallel versions. The **compute time** is relatively low since it processes a single portion of the graph at a time, but the **sequential processing** limits the scalability and performance when dealing with large-scale graphs.

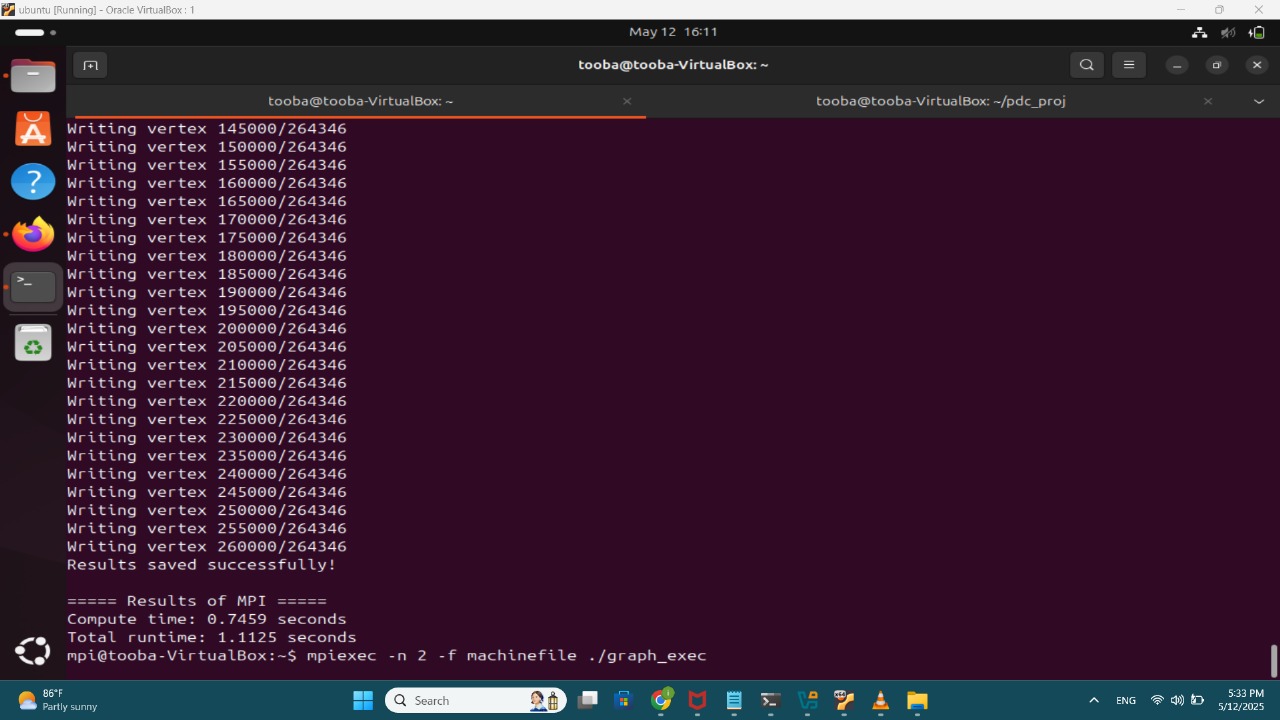


#### **2. MPI Version:**

* **Compute Time**: 0.7459 seconds
* **Total Runtime**: 1.1125 seconds

**Command Used**:  
  
mpiexec -n 2 -f machinefile ./graph\_exec

In the **MPI version**, the graph is partitioned, and each process is assigned a partition to work on. Communication between the processes is handled via **MPI**, and the updated **SSSP** tree is computed in parallel. While the **compute time** is slightly lower than in the sequential version, the **total runtime** is similar or slightly better. This is because the algorithm benefits from **parallelization across partitions**, but the overall efficiency depends on how well the work is distributed across processes. The **boundary synchronization** between partitions can introduce some overhead, especially when the partitions have a high degree of interconnection.

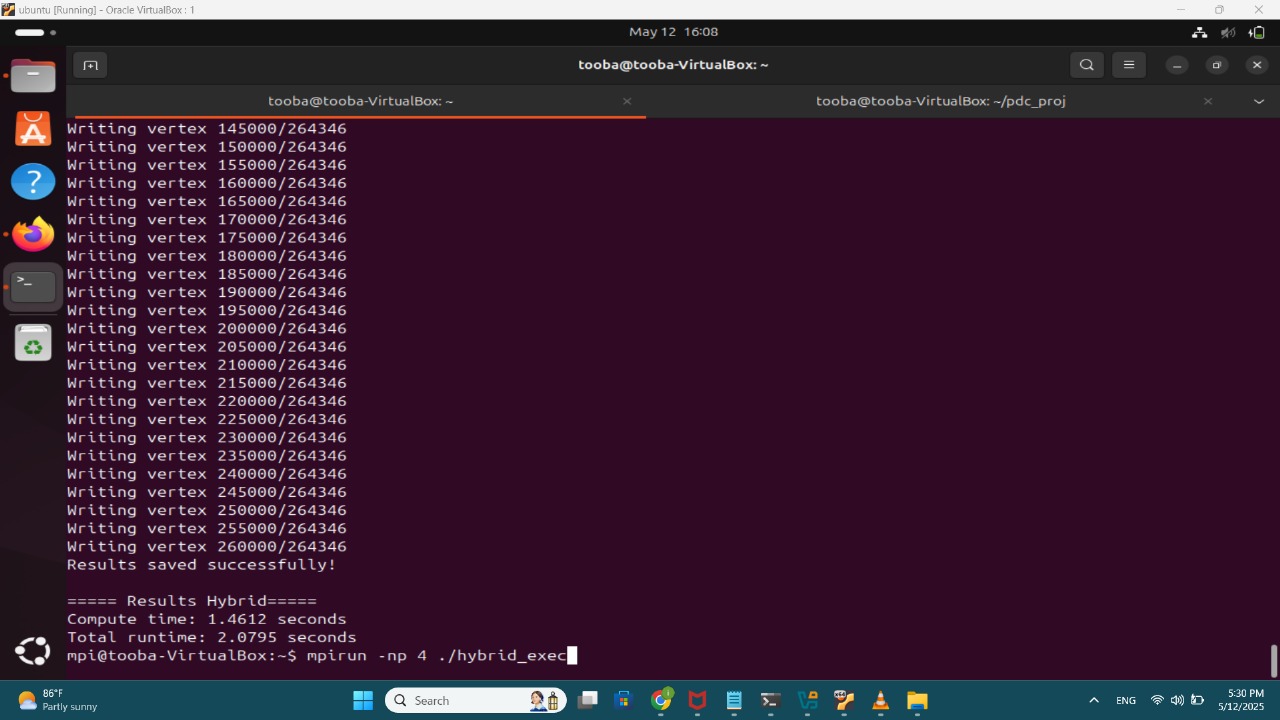


#### **3. Hybrid MPI + OpenMP Version:**

* **Compute Time**: 1.4612 seconds
* **Total Runtime**: 2.0795 seconds

**Command Used**:  
  
mpirun -np 4 ./hybrid\_exec

The **hybrid MPI + OpenMP version** combines **MPI** for inter-process communication with **OpenMP** for intra-process parallelization. While the **compute time** is higher than the MPI version, the **total runtime** is greater as well. This version was designed to maximize performance by utilizing **multiple threads per process** (OpenMP), which parallelizes the computation within each partition. However, the added complexity of **OpenMP synchronization** and **MPI communication overhead** can increase the total runtime. The **hybrid approach** works best for large datasets and when the processing workload within each partition is sufficiently large to justify the use of multiple threads.



### **Reasons for These Results:**

1. **Sequential Version**:  
   * The sequential version processes the graph on a **single thread**, so it is limited by the processing power of a single core.
   * This leads to **higher runtime** as it processes the graph sequentially without parallelization. However, for small datasets or testing purposes, this version can be sufficient.
2. **MPI Version**:  
   * The **MPI version** improves performance by dividing the graph into **partitions** and processing them in parallel across multiple processes.
   * While the **total runtime** shows improvement, it does not scale as much as the hybrid approach due to potential communication overhead between processes and synchronization between boundaries.
3. **Hybrid MPI + OpenMP Version**:  
   * The **hybrid version** is designed to take full advantage of parallelism by using **OpenMP** within each process and **MPI** for inter-process communication.
   * The **higher compute time** observed is due to the **OpenMP parallelization overhead** and the **increased complexity** of handling synchronization across threads.
   * However, this version is optimal for handling very large graphs, especially when the graph's partitions can be processed efficiently using multiple threads. The **extra processing power** provided by **OpenMP** can significantly reduce the time spent on computations within each process.

### **Conclusion:**

In conclusion, the **MPI version** offers a moderate improvement over the **sequential version** by partitioning the graph and parallelizing the work. The **hybrid MPI + OpenMP version** performs the best in theory for large-scale graphs but may have higher computational overhead due to the synchronization and inter-process communication between partitions. The choice of which version to use depends on the size of the dataset and the computational resources available. For smaller graphs, the **sequential version** may suffice, while for larger graphs, the **MPI version** or **hybrid version** should be considered for optimal performance.