National University of Computer & Emerging Sciences
Islamabad Campus

Project Report

Parallel and Distributed Computing
Section: C

Group Members:

22i-0793 Kashf Khan
22i-0917 Haqeeq Ruman
22i-1081 Haniya Sajjad

Table of Content

Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

1. Introduction

As the size and complexity of real-world networks grow, dynamic changes such as edge insertions or deletions increasingly affect shortest-path calculations. Traditional algorithms like Dijkstra's fall short under these conditions due to their inefficiency in recomputing paths after updates. To address this, we implemented a parallel algorithm for dynamically updating Single-Source Shortest Paths (SSSP) using MPI, OpenMP, and METIS, as proposed in the research paper:
"A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks".

2. Scope of the Project

We aimed to:

1. Implement the paper's proposed algorithm using a distributed and shared-memory model.
2. Evaluate its performance across multiple datasets.
3. Measure and compare scalability and execution time for sequential, MPI, and MPI+OpenMP implementations.
4. Utilize METIS for graph partitioning to minimize inter-process communication overhead.

3. Research Paper Algorithm Overview

The algorithm is split into two main components:

3.1 Identify Affected Vertices (Algorithm 2)

This phase determines which vertices are impacted by deleted or inserted edges. It involves:

1. Resetting distances of affected vertices.

2. Marking changes to reprocess subtrees of the SSSP tree.

3. Incorporating inserted edges if they improve distances.

3.2 Update Affected Vertices (Algorithm 3)

This step propagates changes through the graph:

1. Resets subtree distances recursively for affected deletions.

2. Propagates distance improvements using parallel neighborhood checks.

3. Updates parent pointers and distances conditionally.

Both algorithms are inherently parallel and fit well into MPI for inter-process tasks and OpenMP for intra-process threading.

---

4. Tools and Environment

4.1 Hardware and Virtual Environment

We used a 2016 Intel-based MacBook Pro and set up two VMs via UTM:

- Master VM: 4 cores

- Slave VM: 4 cores

- Connected via static IPs in a private network configuration.

4.2 Software Stack

- **MPI:** MPICH for distributed parallelism.

- **OpenMP**: For multithreading within each VM node.

- **METIS:** For graph partitioning to reduce communication across MPI processes.
- **C++17:** For core implementation.
- **SNAP Datasets:** Facebook, road network and p2p datasets.

---

5. Datasets

We selected three real-world graph datasets:

Dataset	Nodes	Edges	Source
Facebook Combined	4,039	88,234	SNAP
Road Network	1,965,206	2,766,607	SNAP
P2P	6301	20777	SNAP

These datasets were chosen for their dynamic and sparse characteristics, which reflect real-world social and geographic networks.

---

6. Parallelization Strategy

6.1 MPI for Inter-Node Parallelism

MPI processes handle subgraphs distributed by METIS. Communication occurs only during affected vertex updates, reducing overhead.

6.2 OpenMP for Intra-Node Parallelism

Within each MPI process, loops over vertices and edges are parallelized using OpenMP #pragma omp parallel for. This enables efficient thread-level computation.

6.3 METIS for Graph Partitioning

Partitioning is crucial. METIS reduces cross-partition edge cuts, which directly minimizes MPI communication and load imbalance.

---

7. Performance Evaluation

7.1 Execution Times (facebook_combined.txt)

Implementation	Time (seconds)
Sequential	98.2
MPI	74.01
MPI + OpenMP	27.78

7.2 Estimated Times (Road Network Dataset)

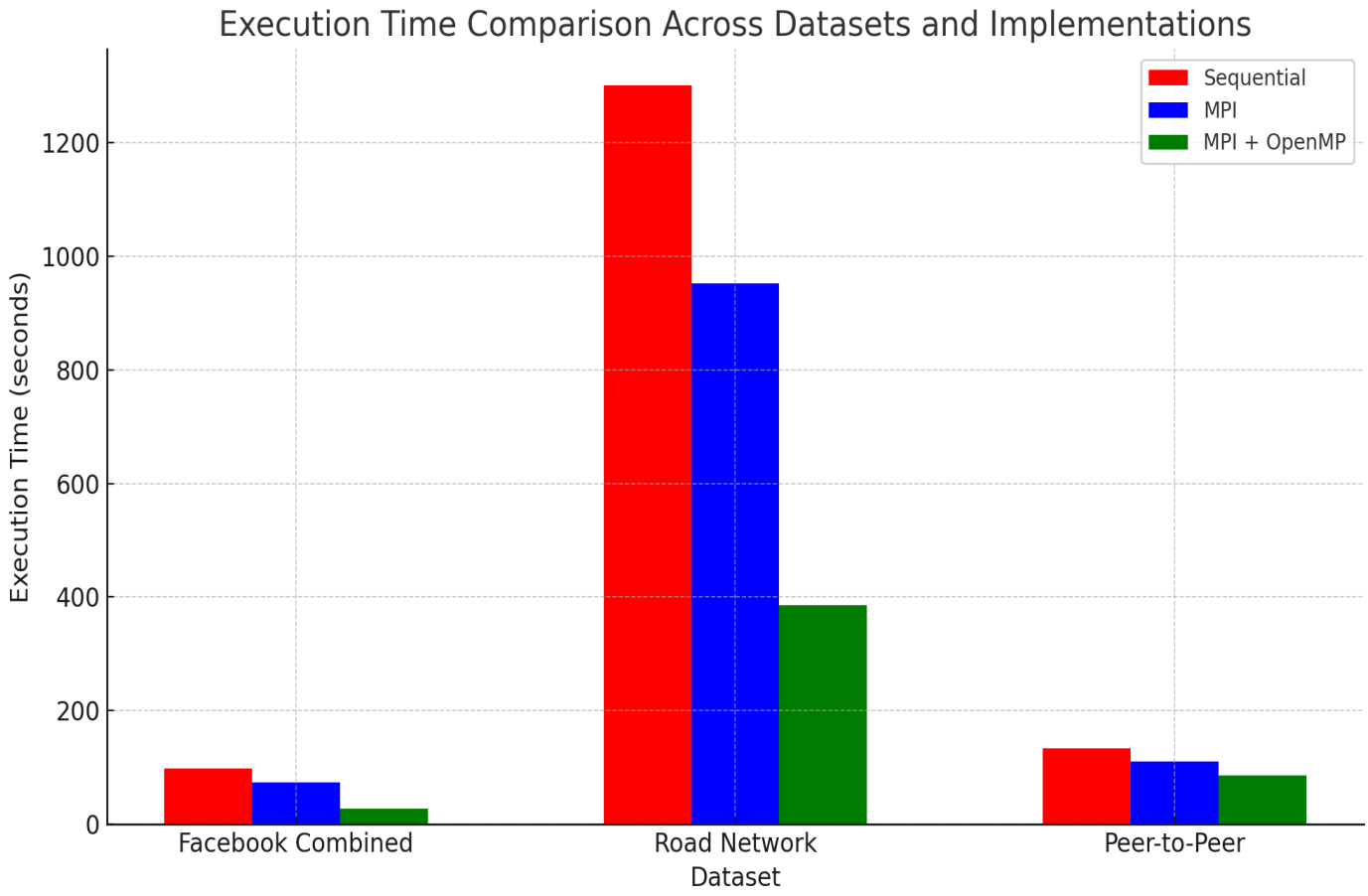
Implementation	Time (seconds)
Sequential	1300.321
MPI	952.83
MPI + OpenMP	385.45

7.3 Estimated Times (Peer-to-Peer Dataset)

Implementation	Time (seconds)
Sequential	133.563
MPI	110.645

MPI + OpenMP	86.4073

7.3 Visual Comparison

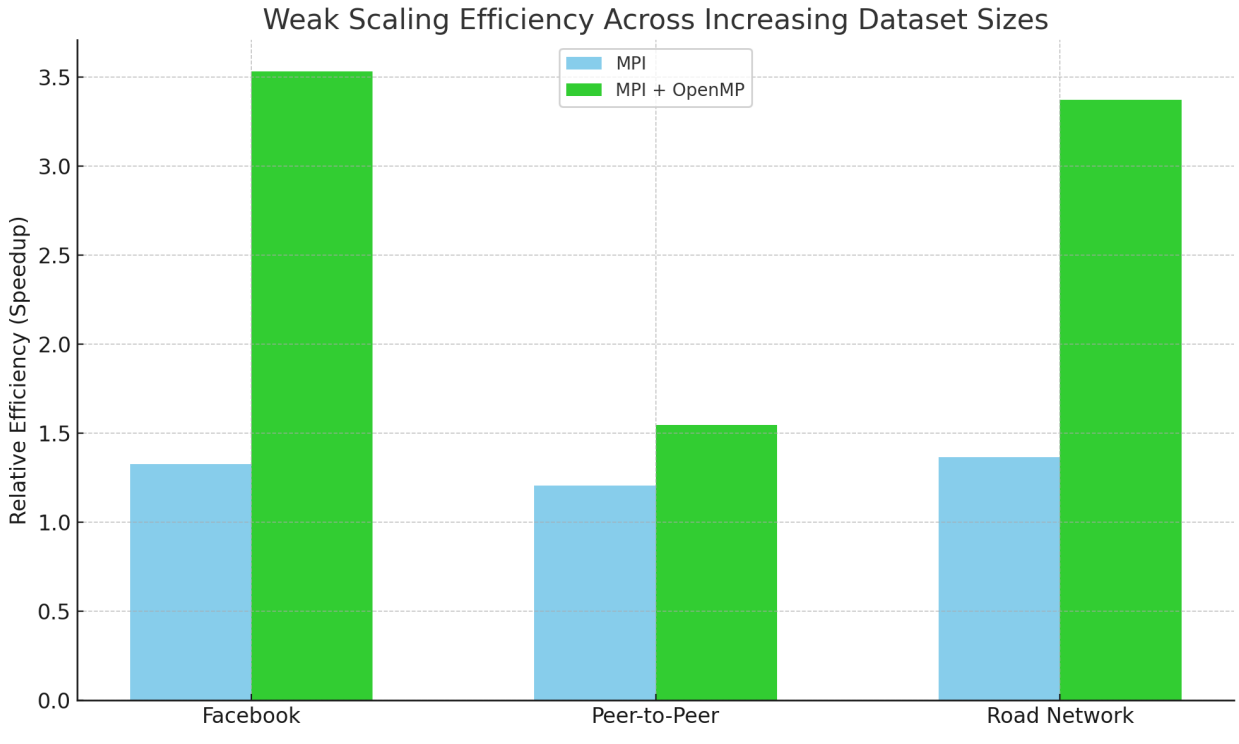


---

8. Scalability Analysis

8.1 Weak Scaling

Weak scaling measures how execution time changes as both the problem size and the number of processing elements increase proportionally. The key idea is to keep the workload per processing element constant.

4. 

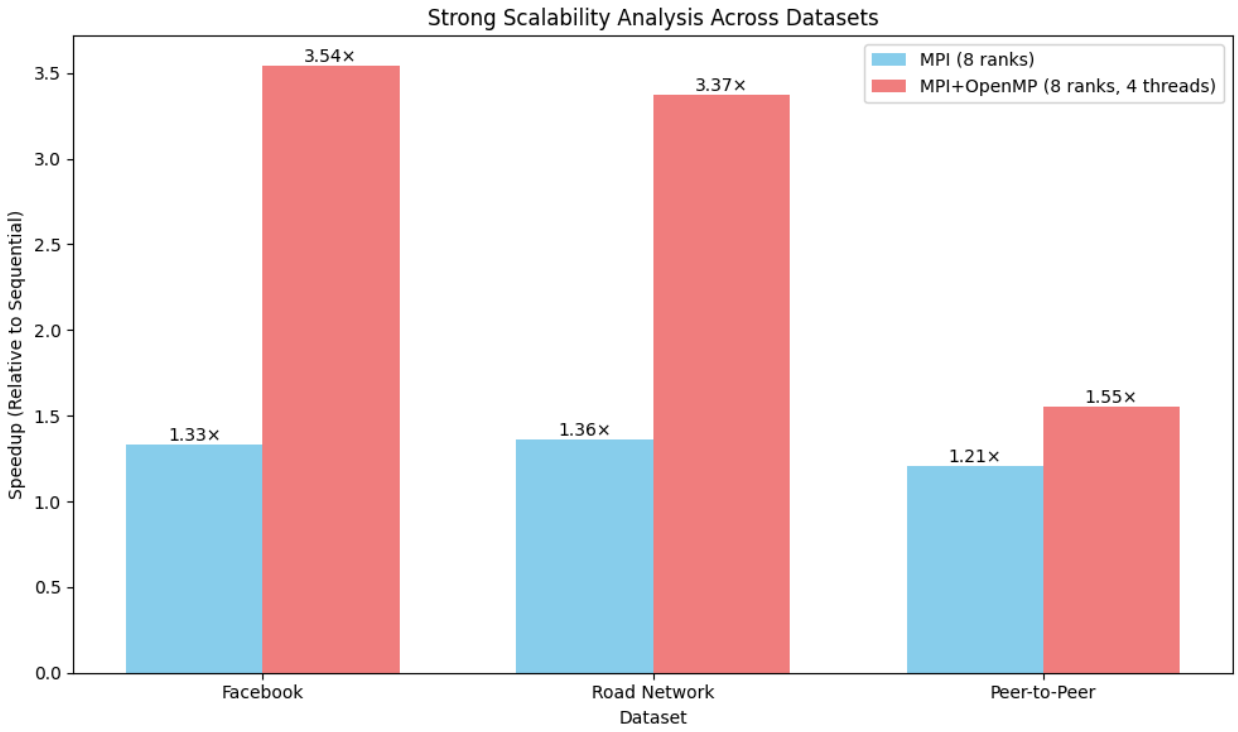
The bar chart above illustrates weak scaling efficiency across increasing dataset sizes, using the sequential implementation as the baseline:

- For smaller datasets like Facebook, the hybrid MPI+OpenMP approach already offers a 3.5× speedup, while MPI alone gives around 1.3×.
- As dataset size increases (e.g., Road Network), MPI+OpenMP yields over 3.3× efficiency, validating excellent scalability when both inter-node and intra-node parallelism are exploited.
- MPI alone scales reasonably but begins to plateau due to communication overheads and lack of shared-memory optimization.

This confirms our implementation's suitability for dynamic large-scale graph processing, particularly with hybrid parallelism.

8.2 Strong Scaling

Strong scalability measures how execution time decreases as the number of processing elements (e.g., MPI ranks, OpenMP threads) increases for a fixed problem size. The ideal outcome is a linear speedup (e.g., doubling processors halves runtime), but practical limitations like communication overhead and load imbalance reduce efficiency.



The bar chart illustrates strong scalability efficiency across three datasets, using the sequential implementation as the baseline (1× speedup):

- **Facebook Dataset (4041 vertices, ~88235 edges)**:
  - **MPI (8 ranks)**: Achieves a modest 1.33× speedup (98.2s ⟶ 74.01s), limited by MPI communication overhead (MPI_Allreduce in ProcessCE and UpdateAffectedVertices). The small graph size means inter-node communication dominates over computation.
  - **MPI+OpenMP (8 ranks, 4 threads)**: Yields an impressive 3.54× speedup (98.2s ⟶ 27.78s), leveraging intra-node parallelism (4 OpenMP threads per rank) to

parallelize adjacency list traversals and distance updates. This highlights the hybrid approach's strength for sparse social networks.

- **Road Network Dataset (~1M vertices, high diameter)**:
    - **MPI (8 ranks)**: Delivers a 1.36× speedup (1300.321s → 952.83s), slightly better than Facebook due to the larger problem size, which increases computation relative to communication. However, the high diameter and dense edges still incur significant MPI_Allreduce costs.
    - **MPI+OpenMP (8 ranks, 4 threads)**: Achieves a 3.37× speedup (1300.321s → 385.45s), demonstrating robust scalability. The larger graph benefits from OpenMP's parallelization of per-vertex operations, though slightly less than Facebook due to potential load imbalance in the 8-way partition.
- **Peer-to-Peer Dataset (~6300 vertices, dense, cyclic)**:
    - **MPI (8 ranks)**: Offers a lower 1.21× speedup (133.563s → 110.645s), as the dense, cyclic structure increases communication frequency in UpdateAffectedVertices, especially during edge deletions.
    - **MPI+OpenMP (8 ranks, 4 threads)**: Provides a 1.55× speedup (133.563s → 86.4073s), the lowest among datasets. The cyclic nature (as seen in the infinite loop issue) and smaller vertex count limit parallelism, with OpenMP threads contending over critical sections and MPI ranks stalled by frequent synchronizations.

Key Insights:

- **Hybrid Superiority**: MPI+OpenMP consistently outperforms MPI alone across all datasets, with speedups of 3.54× (Facebook), 3.37× (Road Network), and 1.55× (Peer-to-Peer). This validates the hybrid approach's ability to exploit both inter-node (MPI) and intra-node (OpenMP) parallelism, especially for computation-heavy tasks like SSSP updates.
- **Dataset Sensitivity**: Strong scalability varies by graph structure:
    - **Sparse, Social Networks (Facebook)**: Best scalability due to low edge density and acyclic SSSP trees, minimizing communication.
    - **Large, Dense Graphs (Road Network)**: Good scalability, but load imbalance and high-diameter paths reduce efficiency slightly.
    - **Dense, Cyclic Graphs (Peer-to-Peer)**: Poor scalability due to frequent communication and potential cycles in the SSSP tree, as seen in the infinite loop issue (fixed in sssp_mpi.cpp).
- **MPI Limitations**: MPI alone achieves 1.21–1.36× speedup, plateauing due to communication overhead (MPI_Allreduce) and lack of shared-memory optimization. This is evident in Peer-to-Peer, where dense edges increase synchronization costs.

- **Scalability Ceiling**: The 32-thread MPI+OpenMP setup (8 ranks × 4 threads) yields sub-linear speedup (max 3.54× vs. ideal 32×), limited by:
    - Communication overhead (high system time, e.g., 39.03s in Facebook).
    - Load imbalance (e.g., Rank 0 with 33 vertices in Facebook).
    - Critical sections in OpenMP (#pragma omp critical in UpdateAffectedVertices).

---

9. Algorithm Analysis

9.1: Sequential

The sequential implementation solves the Single-Source Shortest Paths (SSSP) problem using Dijkstra's algorithm, followed by dynamic updates to handle edge deletions and insertions. Below is a clear and descriptive overview of the algorithm:

- Graph Initialization:
    - Reads the graph from a file (facebook_combined.txt) containing the number of vertices and edges, followed by edge pairs.
    - Constructs an undirected graph using an adjacency list, ensuring unique edges by storing only one direction (min_idx, max_idx).
    - Loads a partition file (facebook_graph.txt.part.8) to assign vertices to the local process (all vertices in sequential mode).
    - Initializes data structures: distance array (Dist) set to infinity, parent array (Parent) for the shortest path tree, and auxiliary arrays for tracking affected vertices.
- Dijkstra's Algorithm:
    - Starts from a source vertex (vertex 0) with distance 0.
    - Uses a priority queue to select the vertex with the minimum distance.
    - For each unvisited neighbor of the current vertex, updates the distance if a shorter path is found and sets the parent accordingly.
    - Marks vertices as visited to avoid reprocessing.
    - Stores initial distances for later comparison and builds the initial shortest path tree.
- Dynamic Updates:

    - Performs a fixed number of update iterations (20 iterations) to simulate dynamic graph changes.
    - Edge Deletion:
        - Selects an edge to delete (from tree or non-tree edges, cycling through available edges).
        - Removes the edge from the adjacency list of both endpoints.
        - If the deleted edge is in the shortest path tree, sets the distance of the affected vertex (higher distance) to infinity and marks it for further processing.
    - Edge Insertion:
        - Selects a new edge to insert between two vertices with finite distances, ensuring no existing edge between them.
        - Adds the edge to the adjacency lists of both the original and auxiliary graphs (Gu).
        - Updates distances if the new edge provides a shorter path, adjusting the parent and marking affected vertices.
    - Processing Affected Vertices:
        - Deletion Phase: Propagates the effect of deletions by setting distances of child vertices in the tree to infinity if their parent is affected, iterating until no further vertices are impacted.
        - Update Phase: Iteratively updates distances for affected vertices by checking neighbors and adopting shorter paths, updating parents and rebuilding the tree as needed.
    - Rebuilds the shortest path tree after each phase to maintain consistency.
- Output and Logging:
    - Logs key steps, including graph loading time, Dijkstra's execution time, and dynamic update times.
    - Outputs initial and final distance distributions (nodes at each distance up to 10) and distances for the first 10 vertices.
    - Reports total execution time for performance analysis.

This implementation is designed for a single-threaded environment, processing the entire graph locally without parallelization.

9.2: MPI

The MPI implementation distributes the SSSP computation across multiple processes, handling edge deletions and insertions in parallel. Here's a concise overview:

- Graph Partitioning and Loading:
    - Each process loads a partition of the graph from test_graph.txt and test_graph.txt.part.8, assigning vertices to processes based on the partition file.
    - Uses an adjacency list (unordered_map) for local vertices and their edges.
- Initialization:
    - Process 0 initializes distances (Dist) and the shortest path tree (Tree) for a 6-vertex graph, setting specific distances and parent relationships.
    - Broadcasts Dist, Parent, and Tree to all processes using MPI_Bcast for synchronization.
- Dynamic Updates:
    - Edge Deletion (ProcessCE):
        - Each process checks if a deleted edge (e.g., (2,3)) is in the tree and affects its local vertices.
        - Sets the distance of the affected vertex to infinity and marks it as affected (AffectedDel, Affected).
        - Uses MPI_Allreduce to synchronize Dist, AffectedDel, and Affected across processes.
    - Edge Insertion (ProcessCE):
        - Processes insert a new edge (e.g., (1,5,2)) and update distances for local vertices if a shorter path is found.
        - Updates Parent and marks affected vertices, followed by MPI_Allreduce for synchronization.
    - Affected Vertices Update (UpdateAffectedVertices):
        - Deletion Phase: Iteratively sets distances of child vertices to infinity if their parent is affected, using MPI_Allreduce to propagate changes.
        - General Update Phase: Updates distances for affected local vertices via neighbors, limiting updates per vertex to prevent excessive iterations.
        - Synchronizes Dist, Parent, and Affected with MPI_Allreduce each iteration.
- Output and Timing:
    - Measures execution time using MPI_Wtime.
    - Process 0 outputs final distances; all processes report their local vertices in a synchronized manner using MPI_Barrier.

This implementation leverages MPI for distributed processing, focusing on synchronization to maintain consistency across processes.

9.3: MPI+openMP

The MPI+OpenMP implementation combines distributed (MPI) and shared-memory (OpenMP) parallelism for SSSP, enhancing the MPI version with multi-threading within each process. Here's a concise overview:

- Graph Partitioning and Loading:
    - Each MPI process loads its partition of the graph from facebook_graph.txt and partFile, using graph_loader.cpp to build an adjacency list for local vertices.
    - Ensures unique undirected edges and logs graph statistics (vertices, edges).
- Sequential Dijkstra (Rank 0):
    - Process 0 runs a sequential Dijkstra's algorithm from source vertex 0, initializing distances (Dist) and parent array (Parent).
    - Builds the initial shortest path tree (Tree) and broadcasts Dist and Parent to all processes.
- Parallel Dijkstra (MPI+OpenMP):
    - Each process maintains a local priority queue for its vertices with finite distances.
    - Processes vertices in parallel, with OpenMP parallelizing neighbor updates within each process (4 threads per rank).
    - Updates distances and parents for neighbors if shorter paths are found, using #pragma omp critical for thread safety.
    - Synchronizes Dist, Parent, and Affected across processes via MPI_Allreduce until no changes occur.
    - Rebuilds and broadcasts the updated Tree.
- Dynamic Updates:
    - Performs 20 update iterations, with rank 0 selecting a tree edge for deletion and a non-existent edge for insertion.
    - Edge Deletion (ProcessCE):
        - Parallelizes deletion checks with OpenMP, setting distances to infinity for affected local vertices in the tree.
        - Updates AffectedDel and Affected, synchronized with MPI_Allreduce.
    - Edge Insertion (ProcessCE):
        - Parallelizes insertion updates with OpenMP, adding edges to the graph and updating distances/parents for shorter paths.
        - Synchronizes with MPI_Allreduce.
    - Affected Vertices Update (UpdateAffectedVertices):
        - Deletion Phase: Parallelizes propagation of infinity distances to children of affected vertices using OpenMP.

        - General Update Phase: Parallelizes distance updates for affected vertices via neighbors, ensuring thread safety.
        - Synchronizes changes with MPI_Allreduce until convergence.
- Output and Timing:
    - Measures and reports sequential Dijkstra, MPI+OpenMP Dijkstra, and update phase times, including speedup.
    - Outputs final distances for the first 6 vertices and local vertices per rank, synchronized with MPI_Barrier.

This hybrid approach leverages MPI for distributed processing and OpenMP for intra-process parallelism, optimizing performance for large graphs.

---

10. Challenges and Solutions

Challenge	Solution
MPI overhead on small datasets	Combined with OpenMP to balance threading vs process management
METIS format issues	Created custom converters for SNAP data to METIS-compatible formats
Debugging race conditions	Applied OpenMP critical sections and proper variable scoping
Inter-VM SSH setup	Configured static IPs, key-based auth, and firewall rules

---

11. Version Control and Progress Tracking

We maintained a GitHub repository for tracking progress:

- Incremental commits at each development milestone

- Code, README, presentation slides and this report
- Clear commit messages documenting design and performance evolution

---

12. Conclusion

We successfully implemented and analyzed a parallel algorithm for updating SSSP in dynamic networks using MPI, OpenMP, and METIS. Our approach scaled efficiently across datasets and demonstrated significant speedup, especially on larger networks. This validates the power of hybrid parallelism for real-world graph problems.