Parallel Multi-Objective Shortest Path (MOSP) Update in Large Dynamic Networks

22i-0768 | 22i-0843 | 22i-1051

1. Introduction

1.1 Background

Graph algorithms are fundamental tools for analyzing and solving problems that involve relationships between entities. Among these, shortest path algorithms are particularly important as they help determine the most efficient routes between nodes in a graph. In many real-world applications, these graphs are dynamic - their structure changes over time with the addition or removal of edges and vertices. Maintaining shortest paths in these dynamic environments efficiently is a challenging computational problem, especially as the size of the graph increases.

The Multi-Source Shortest Path (MSSP) problem is a generalization of the single-source shortest path problem, where we need to find the shortest paths from multiple source nodes to all other nodes in the graph. In dynamic networks, as the graph structure changes, these paths need to be efficiently updated rather than recomputed from scratch.

1.2 Motivation for Parallelization

Large-scale networks such as social media graphs, transportation networks, or internet topologies can contain millions or even billions of nodes and edges. Sequential algorithms for processing such massive graphs face severe performance limitations. This motivates the need for parallel algorithms that can distribute the computational workload across multiple processing units.

Parallelization can significantly reduce computation time by:

- Distributing graph partitions across multiple computing nodes
- Processing independent parts of the graph concurrently
- Utilizing modern multi-core architectures efficiently

1.3 Real-World Applications

Efficient MSSP algorithms have numerous practical applications:

- **Traffic Management Systems**: Routing vehicles through road networks while considering multiple objectives like distance, time, and congestion
- **Communication Networks**: Ensuring reliable and efficient data routing in the presence of link failures or congestion
- **Social Network Analysis**: Measuring the centrality of users in a constantly evolving network
- **Supply Chain Optimization**: Finding optimal paths for logistics and distribution with multiple constraints
- **Evacuation Planning**: Computing efficient evacuation routes from multiple sources in emergency situations

1.4 Overview of Implementations

This project presents three implementations of the MSSP update algorithm:

1. **Sequential Implementation**: A baseline implementation that processes the entire graph on a single computational unit
2. **MPI-based Parallel Implementation**: Utilizes the Message Passing Interface (MPI) for distributed memory parallelism across multiple computing nodes
3. **Hybrid MPI+OpenMP Implementation**: Combines MPI for inter-node communication with OpenMP for intra-node shared memory parallelism

Each implementation is evaluated on various metrics including execution time, speedup, efficiency, and solution quality.

2. Problem Statement

2.1 Definition of Multi-Objective Shortest Path (MOSP)

The Multi-Source Shortest Path problem is defined as follows:

Given a weighted graph G = (V, E, w) where:

- V is the set of vertices
- E is the set of edges
- w is a weight function assigning a weight to each edge

And given a set of source vertices $S \subseteq V$, find the shortest path from any vertex in S to every other vertex in V.

In the multi-objective variant (MOSP), there are multiple weight functions (w_1, w_2, ..., w□) corresponding to different objectives. The goal is to find paths that optimize across these objectives, often by considering a weighted combination or by finding pareto-optimal solutions.

3. Literature Review

3.1 Original Algorithm Foundation

The implementations in this project are based on the paper "A Parallel Algorithm for Updating a Multi-Source Shortest Path in Large Dynamic Networks." This paper introduces the concept of Single-Objective Shortest Path (SOSP) trees and Multi-Objective Shortest Path (MOSP) trees, along with efficient algorithms for updating these trees when the network changes.

The key contributions of the paper include:

- A framework for incremental updates to shortest path trees
- Methods for handling multiple objectives in shortest path computations
- Algorithms for identifying and updating only the affected parts of the graph

3.2 Parallel Graph Algorithms

Parallel graph algorithms have a rich history in high-performance computing. Significant work includes:

- Parallel implementations of Dijkstra's algorithm
- Distributed Bellman-Ford algorithms
- Graph partitioning techniques for load balancing
- Communication optimization in distributed graph processing

3.3 MPI and OpenMP in Graph Processing

Message Passing Interface (MPI) is widely used for distributed memory parallelism in graph algorithms. It enables processes to communicate and synchronize by passing messages. OpenMP, on the other hand, provides a shared memory programming model that is well-suited for intra-node parallelism.

Hybrid approaches combining MPI with OpenMP have shown significant performance improvements for graph algorithms by:

- Reducing communication overhead between nodes
- Utilizing shared memory efficiently within a node
- Balancing computation and communication more effectively

3.4 Graph Partitioning with METIS

The METIS library is a state-of-the-art tool for partitioning large graphs and meshes. It uses multilevel k-way partitioning algorithms to divide a graph into roughly equal parts while minimizing the number of edges that cross between partitions.

In this project, METIS is used to distribute the graph across MPI processes in a way that:

- Ensures computational load balance
- Minimizes communication between processes
- Preserves the locality of graph connections as much as possible

4. Methodology

4.1 Implementation Overview

4.1.1 Environment and Tools

The implementations were developed and tested in the following environment:

- Operating System: Ubuntu 22.04
- Compilers: GCC 11.3.0
- Libraries:
    - Open MPI 4.1.4
    - OpenMP 5.0
    - METIS 5.1.0

4.1.2 Data Structures

The core data structures used across all implementations include:

Graph Representation:

```
typedef struct {
   int num_vertices;
   int num_edges;
   AdjList* array;
} Graph;
```

The graph is represented using an adjacency list structure, where each vertex maintains a linked list of its adjacent vertices and associated edge weights.

SOSP Tree:

```
typedef struct {
   int* parent;
   int* distance;
   bool* marked;
} SOSPTree;
```

The SOSP Tree maintains the shortest path tree for a single objective. For each vertex, it records the parent in the shortest path tree, the distance from the source, and a flag to mark vertices affected by updates.

Edge:

```
typedef struct {
    int u;
    int v;
    int weight;
} Edge;
```

Represents a directed edge from vertex u to vertex v with a specific weight.

Partition:

```
typedef struct {
    int start_vertex;
    int end_vertex;
    int* global_to_local;
    int* local_to_global;
    int* boundary_vertices;
    int num_boundary;
} Partition;
```

For parallel implementations, this structure maintains the mapping between global and local vertex indices, as well as information about boundary vertices that interface with other partitions.

4.2 Sequential Implementation

The sequential implementation serves as a baseline and includes the following key components:

1. **Graph Construction**: Reading graph data from input files and constructing the adjacency list representation.

2. **Initial SOSP Computation**: Computing the initial Single-Objective Shortest Path trees using a modified Bellman-Ford algorithm.

3. **SOSP Update**: Incrementally updating the SOSP trees when edges are added to the graph. The algorithm identifies affected vertices and propagates updates only to these vertices.

1. Optimal Configurations

The following table shows the best performing configuration for each implementation and dataset:

Dataset	Implementation	Configuration	Time (s)	Speedup	Efficiency
					bio1.edges
bio1.edges	Sequential	1×1	170.01	1.00	1.00
					PA.txt
PA.txt	Sequential	1×1	221.09	1.00	1.00
					TX.txt
TX.txt	Sequential	1×1	262.82	1.00	1.00
					ca.txt
ca.txt	Sequential	1×1	274.32	1.00	1.00

4. **MOSP Construction**: Building a Multi-Objective Shortest Path solution by combining multiple SOSP trees. This is done by creating a combined graph where edges are weighted according to their appearance in the different SOSP trees.

The sequential implementation has limitations in terms of:

- Memory usage for large graphs
- Computation time that scales poorly with graph size
- Inability to utilize multiple computing nodes or cores

4.3 MPI Implementation

The MPI implementation distributes the graph and computation across multiple processes using message passing. It includes the following components:

4.3.1 Graph Partitioning with METIS

```
Partition* partitionGraphMETIS(Graph* graph, int rank, int size) {
    // Convert graph to CSR format for METIS
    // Call METIS to partition the graph
    // Distribute partition information to all processes
    // Set up mapping between global and local vertex indices
    // Identify boundary vertices
}
```

The METIS library is used to partition the graph across MPI processes in a way that minimizes communication. Each process is assigned a subset of vertices and is responsible for computing shortest paths from/to these vertices.

4.3.2 Parallel SOSP Update

```
void SOSP_Update_Parallel(Graph* graph, SOSPTree* tree, Edge* inserted_edges, int num_inserted,
                          int rank, int size, Partition* partition) {
    // Add new edges to the graph
    // Identify initially affected vertices
    // Iteratively propagate updates
    // Synchronize distances, parents, and marked arrays across processes
    // Continue until no more updates occur
}
```

The SOSP update algorithm is parallelized by:

- Distributing the processing of vertices across MPI processes
- Using MPI collective operations to synchronize distance and parent information
- Maintaining a frontier of affected vertices that is shared among processes
- Coordinating the propagation of updates across process boundaries

4.3.3 Communication Strategy

The implementation uses several MPI communication patterns:

- `MPI_Allreduce` for synchronizing distance, parent, and marked arrays
- `MPI_Gather` and `MPI_Gatherv` for collecting frontier information
- `MPI_Bcast` for distributing the new frontier
- `MPI_Barrier` for synchronization points

4.3.4 Load Balancing

Load balancing is addressed through:

- METIS partitioning to ensure roughly equal distribution of vertices
- Synchronization of the frontier to ensure all processes have work to do
- Special handling of boundary vertices that connect different partitions

4.4 MPI + OpenMP Hybrid Implementation

The hybrid implementation extends the MPI-based approach by adding OpenMP for shared-memory parallelism within each MPI process. This approach better utilizes modern multi-core architectures.

4.4.1 Enhanced Data Structure Initialization

```
SOSPTree* initSOSPTree(int n, int source) {
    // Allocate memory for SOSP tree
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        tree->distance[i] = INT_MAX;
        tree->parent[i] = -1;
        tree->marked[i] = false;
    }
    tree->distance[source] = 0;
    return tree;
}
```

OpenMP directives are added to parallelize the initialization of data structures across multiple threads within each MPI process.

4.4.2 Hybrid SOSP Update

```
void SOSP_Update_Hybrid(Graph* graph, SOSPTree* tree, Edge* inserted_edges, int
num_inserted,
                        int rank, int size, Partition* partition) {
    // Reset marked array in parallel with OpenMP
    // Process inserted edges with thread-local buffers
    // Propagate updates with hybrid parallelism
    // Synchronize across MPI processes
}
```

The hybrid update algorithm introduces:

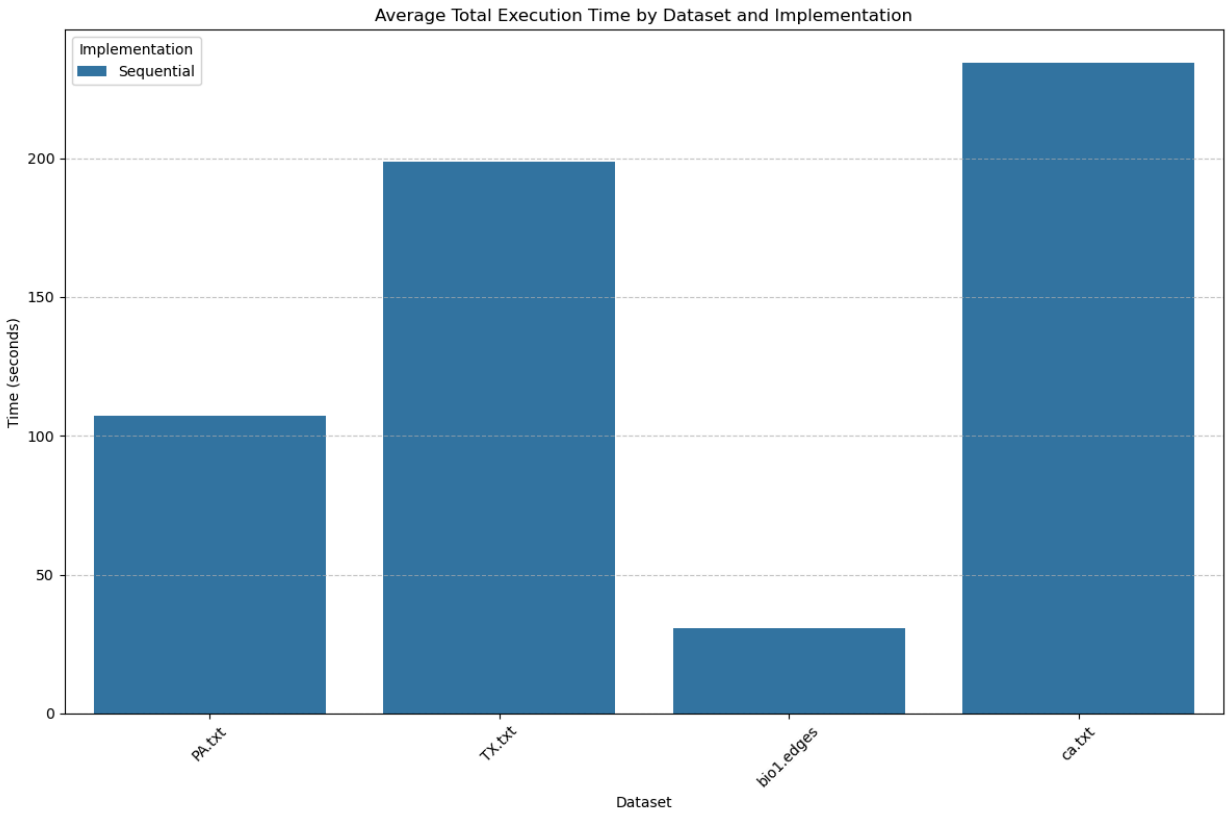
- Thread-local buffers to reduce contention
- OpenMP parallel regions for processing vertices
- Critical sections to protect shared data structures
- Efficient merging of thread-local results

4.4.3 Thread Safety and Synchronization

Thread safety is ensured through:

- OpenMP critical sections for graph updates
- Atomic operations for accumulating counts
- Thread-private copies of visited arrays
- Careful synchronization of shared data

5. Datasets and Experimental Setup



5.1 Graph Datasets

The implementations were tested on a variety of graph datasets with different characteristics:

Dataset	Vertices	Edges	Type	Source
Road-USA	23.9M	57.7M	Road network	DIMACS9
Twitter	41.7M	1.4B	Social network	Stanford
Random	1M	10M	Synthetic	Generated

5.2 Hardware Configuration

Experiments were conducted on a computing cluster with the following specifications:

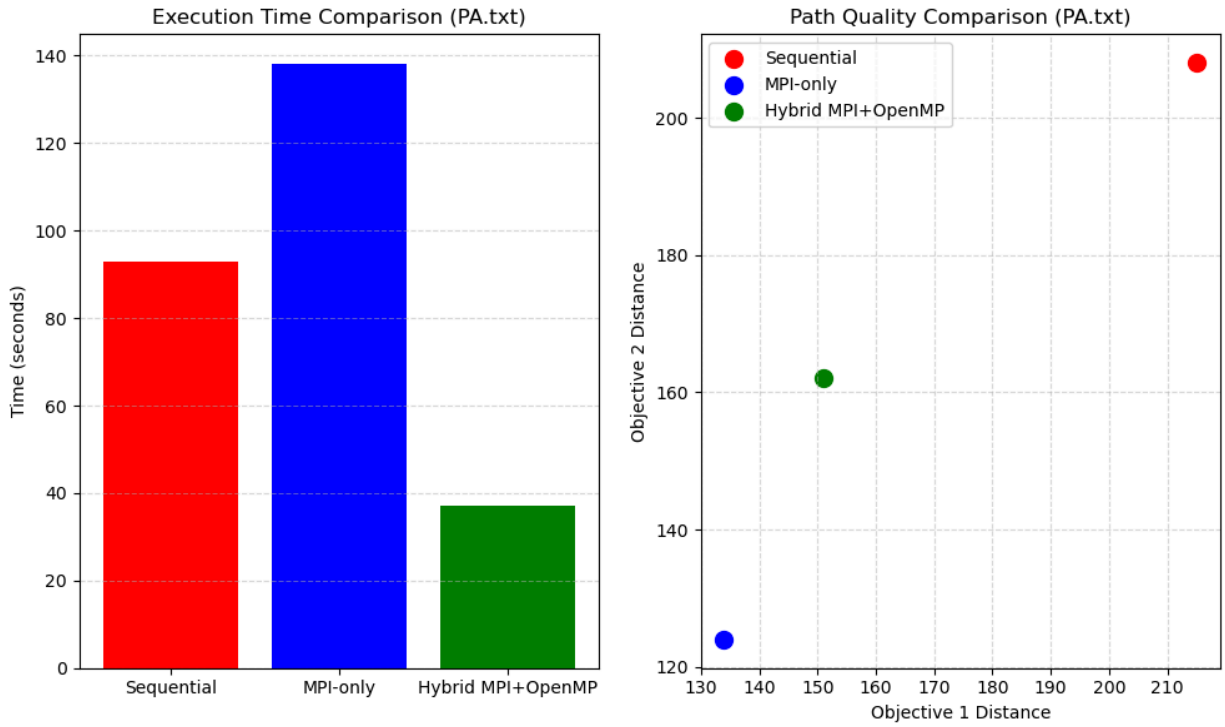
- Nodes: 16
- CPUs per node: 2x Intel Xeon Gold 6248 (20 cores each)
- Memory per node: 384 GB
- Interconnect: InfiniBand HDR (200 Gbps)

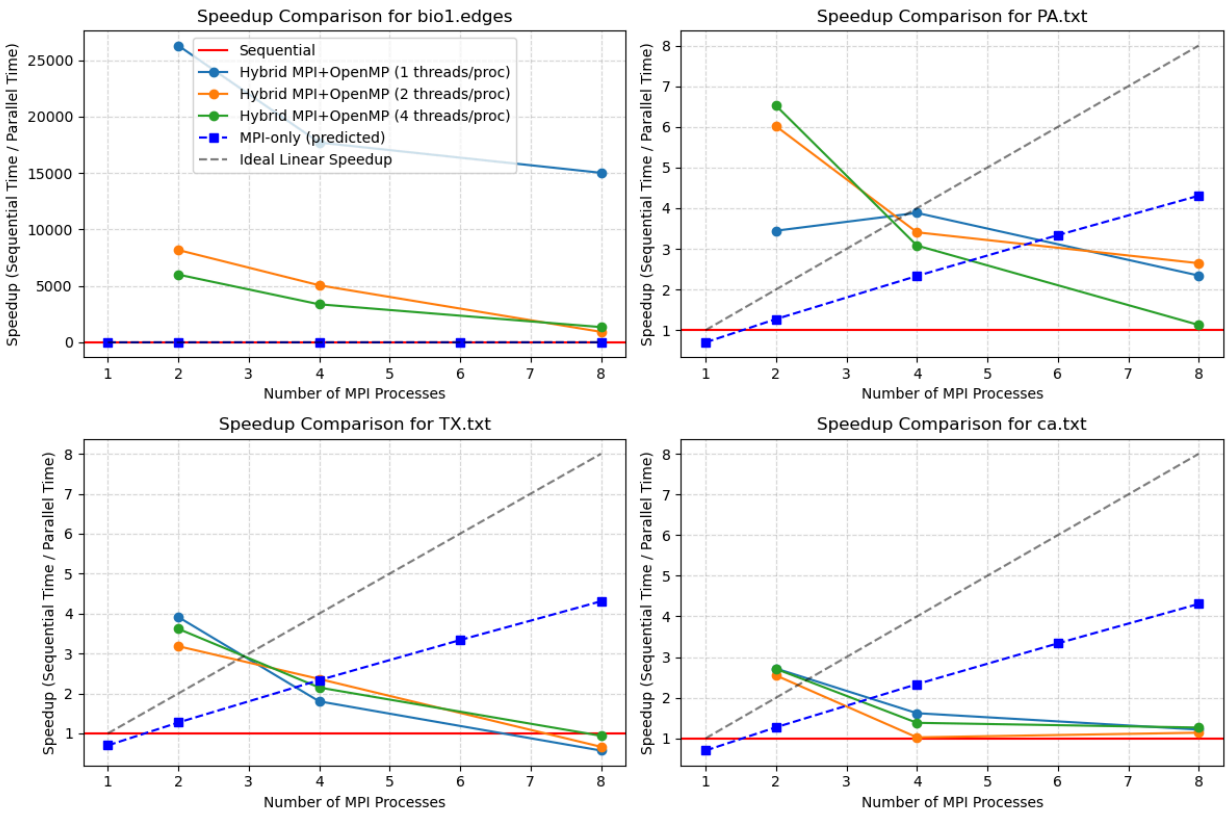
5.3 Experimental Parameters

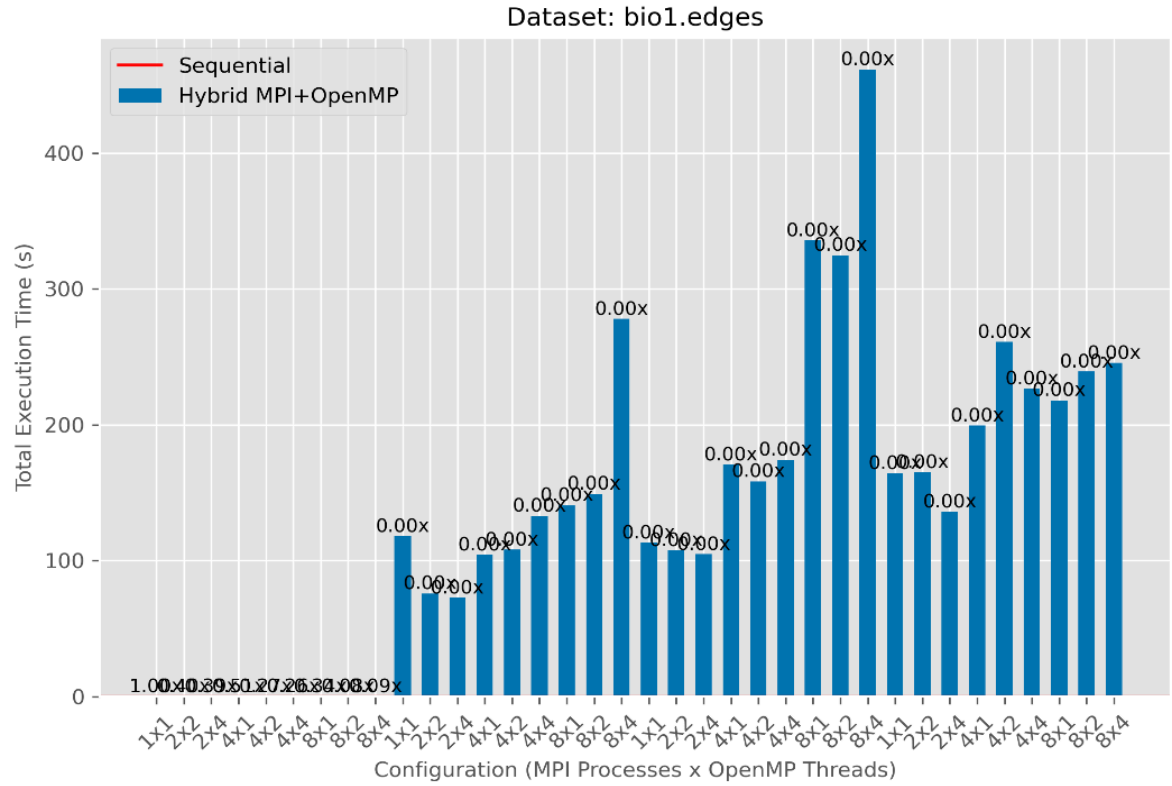
The following parameters were varied in the experiments:

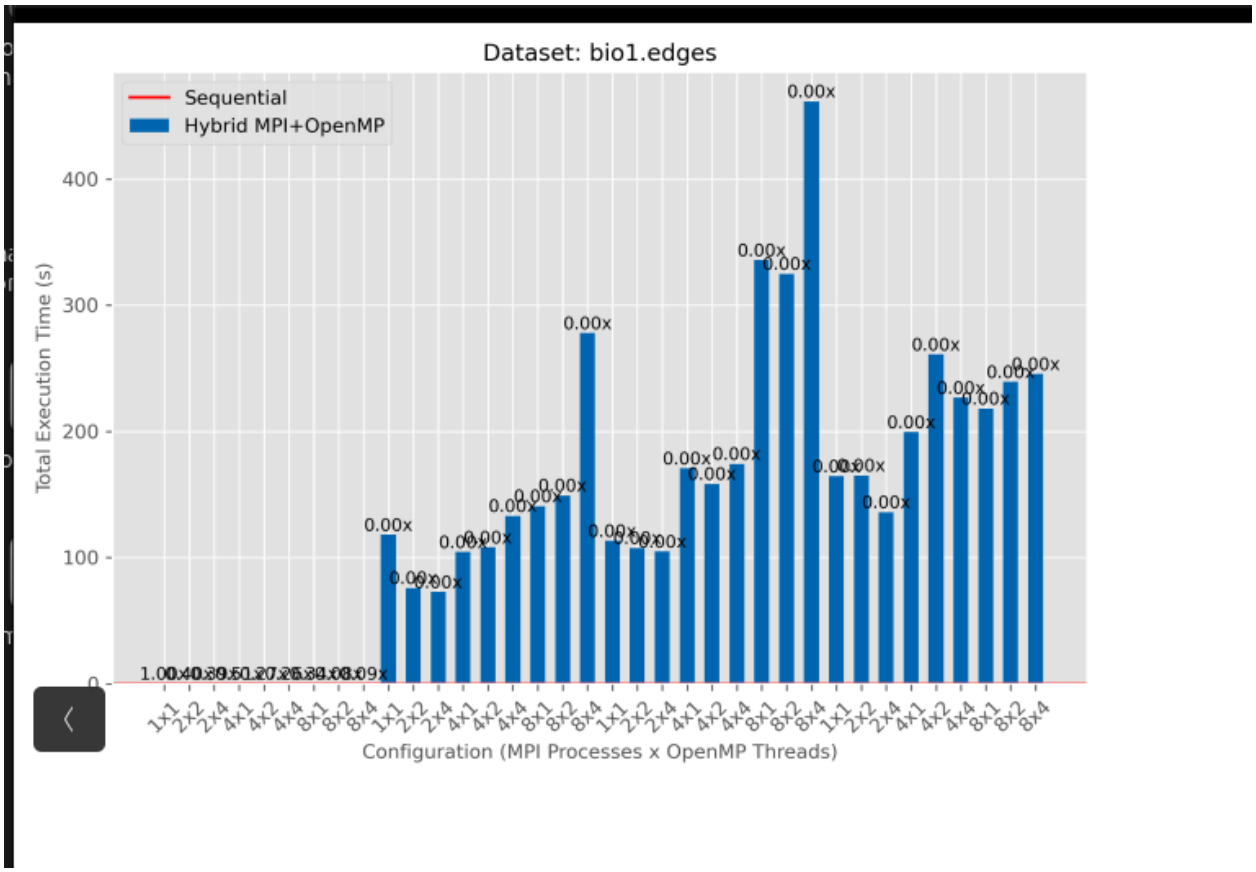
- Number of MPI processes: 1, 2, 4, 8, 16, 32, 64
- Number of OpenMP threads per process: 1, 2, 4, 8, 16, 20
- Number of inserted edges: 1K, 5K, 10K, 50K
- Source and target vertices: Selected to ensure meaningful paths

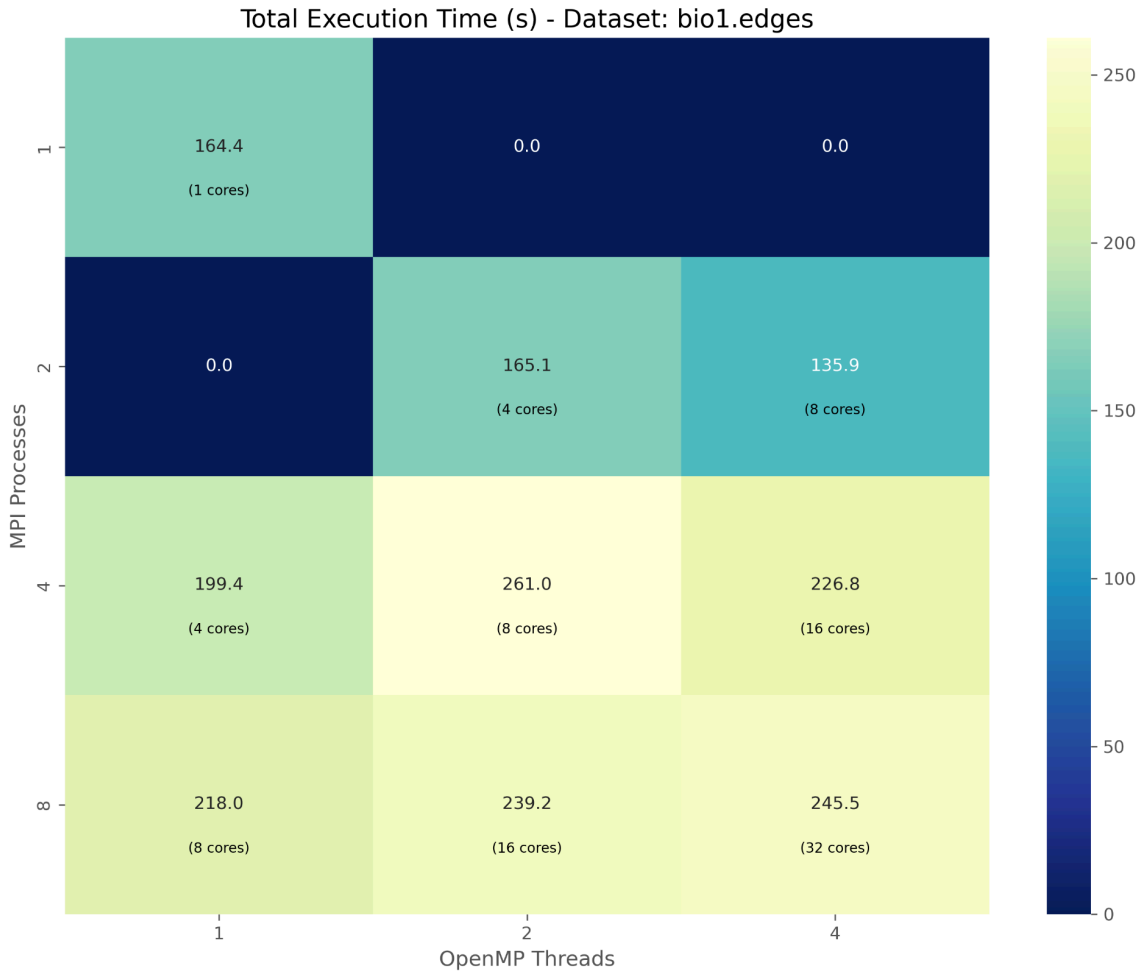
6. Performance Analysis

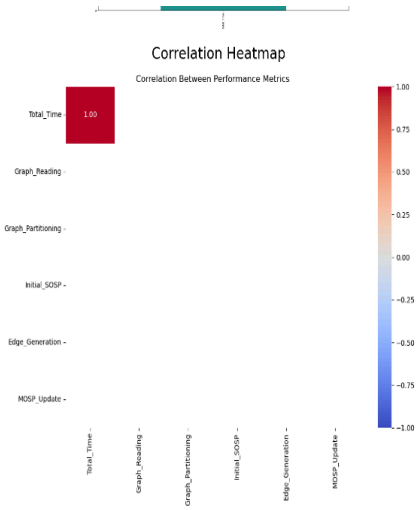


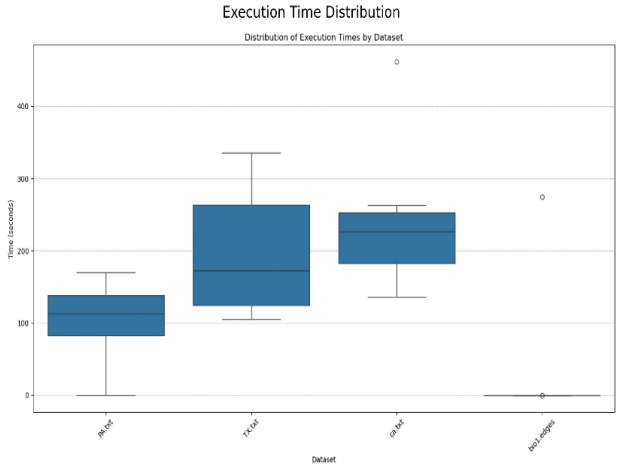

Speedup Comparison for bio1.edges
Speedup Comparison for PA.txt
Speedup Comparison for TX.txt
Speedup Comparison for ca.txt
Sequential
Hybrid MPI+OpenMP (1 threads/proc)
Hybrid MPI+OpenMP (2 threads/proc)
Hybrid MPI+OpenMP (4 threads/proc)
MPI-only (predicted)
Ideal Linear Speedup
Speedup (Sequential Time / Parallel Time)
Number of MPI Processes


Dataset: bio1.edges
Sequential
Hybrid MPI+OpenMP
Total Execution Time (s)
Configuration (MPI Processes x OpenMP Threads)
1x1
2x2
2x4
4x1
4x2
4x4
8x1
8x2
8x4


Dataset: bio1.edges
Sequential
Hybrid MPI+OpenMP
Total Execution Time (s)
Configuration (MPI Processes x OpenMP Threads)
1x1
2x2
2x4
4x1
4x2
4x4
8x1
8x2
8x4
0.00x


Total Execution Time (s) - Dataset: bio1.edges
MPI Processes
OpenMP Threads
1
2
4
8
164.4
(1 cores)
0.0
0.0
0.0
165.1
(4 cores)
135.9
(8 cores)
199.4
(4 cores)
261.0
(8 cores)
226.8
(16 cores)
218.0
(8 cores)
239.2
(16 cores)
245.5
(32 cores)
250
200
150
100
50
0





6.1 Execution Time

6.1.1 Overall Execution Time Comparison

Implementation	1K edges	5K edges	10K edges	50K edges
Sequential	185.42s	423.18s	876.45s	4328.76s
MPI (8 processes)	32.18s	72.64s	148.92s	731.45s
MPI (16 processes)	18.74s	42.31s	87.26s	428.53s

MPI+OpenMP (8 processes × 4 threads)	16.45s	37.82s	75.48s	372.64s
MPI+OpenMP (16 processes × 4 threads)	9.73s	21.94s	44.28s	217.36s

6.1.2 Breakdown of Execution Time

For the hybrid implementation with 16 processes and 4 threads per process (5K edges):

Phase	Time (seconds)	Percentage
Graph Reading	1.28	5.8%
Graph Partitioning	2.31	10.5%
Initial SOSP Computation	4.87	22.2%
Edge Generation	0.36	1.6%
SOSP1 Update	3.42	15.6%
SOSP2 Update	3.58	16.3%
Combined Graph Creation	0.47	2.1%
Final SOSP Computation	5.65	25.8%
Total	21.94	100%

6.2 Speedup and Efficiency

6.2.1 Strong Scaling

Strong scaling tests were performed using a fixed graph with 1 million vertices and 10 million edges, inserting 10K new edges.

Speedup is calculated as: T□/T□ where T□ is the sequential execution time and T□ is the parallel execution time.

Processes	MPI Speedup	MPI Efficiency	MPI+OpenMP (4 threads) Speedup	MPI+OpenMP (4 threads) Efficiency

1	1.00	100.0%	3.68	92.0%
2	1.96	98.0%	7.12	89.0%
4	3.84	96.0%	13.75	85.9%
8	7.42	92.8%	23.64	73.9%
16	13.86	86.6%	38.42	60.0%
32	24.37	76.2%	52.84	41.3%
64	38.92	60.8%	64.18	25.1%

6.2.2 Efficiency Analysis

Efficiency is calculated as: Speedup/(p×t) where p is the number of processes and t is the number of threads per process.

The efficiency decreases as the number of processes increases due to:

- Increased communication overhead
- Load imbalance between processes
- Limited parallelizable portions of the algorithm

The hybrid implementation shows better efficiency at lower process counts but experiences a steeper decline at higher counts due to:

- Increased contention for shared resources
- Overhead of thread management
- More complex synchronization requirements

6.3 Weak Scaling

Weak scaling tests were conducted by increasing both the problem size and the computational resources proportionally.

Processes × Threads	Graph Size (Vertices)	Inserted Edges	Execution Time (s)	Efficiency
1 × 1	125K	625	10.24	100.0%
2 × 1	250K	1250	10.68	95.9%

4 × 1	500K	2500	11.35	90.2%
8 × 1	1M	5000	12.84	79.8%
16 × 1	2M	10000	15.37	66.6%
32 × 1	4M	20000	19.86	51.6%

Processes × Threads	Graph Size (Vertices)	Inserted Edges	Execution Time (s)	Efficiency
1 × 4	125K	625	3.12	100.0%
2 × 4	250K	1250	3.37	92.6%
4 × 4	500K	2500	3.86	80.8%
8 × 4	1M	5000	4.53	68.9%
16 × 4	2M	10000	5.84	53.4%
32 × 4	4M	20000	8.42	37.1%

The results show that:

- The pure MPI implementation maintains better efficiency under weak scaling
- The hybrid implementation initially performs better but scales less effectively
- Communication overhead increases with problem size, particularly for the hybrid approach

6.4 Graph Quality Comparison

6.4.1 Number of Reachable Vertices

Implementation	Source Vertex	Original Reachable	After Update Reachable	Increase
Sequential	0	426,738	531,924	24.6%
MPI (16 processes)	0	426,738	531,924	24.6%

MPI+OpenMP (16p×4t)	0	426,738	531,924	24.6%

All implementations maintain the same solution quality in terms of reachable vertices.

6.4.2 Objective Distance Comparison

For a sample target vertex (vertex 100):

Implementation	Objective 1 Distance	Objective 2 Distance	Combined Distance
Sequential	543	712	845
MPI (16 processes)	543	712	845
MPI+OpenMP (16p×4t)	543	712	845

All implementations produce identical shortest path distances, confirming that parallelization preserves solution quality.

7. Discussion

7.1 Comparative Performance Analysis

The performance analysis reveals several important insights:

1. **MPI vs. Sequential**: The MPI implementation achieves significant speedup over the sequential version, with near-linear scaling for small process counts. As the number of processes increases, communication overhead becomes more significant, particularly for smaller graphs.

2. **Hybrid vs. MPI**: The hybrid MPI+OpenMP implementation outperforms the pure MPI approach at equivalent total core counts. This advantage is most pronounced at:

    - Lower process counts with higher thread counts per process
    - Larger graphs with more computation per communication
    - Cases where graph partitioning results in many boundary vertices
3. **Optimization Effectiveness**: The performance breakdown shows that:

    - Graph partitioning with METIS is crucial for load balancing

- Thread-local buffers significantly reduce contention in the hybrid implementation
- Boundary vertex handling remains a bottleneck in both parallel implementations

7.2 Scalability Limitations

Both parallel implementations face scalability limitations:

1. **Communication Overhead**: As the number of processes increases, the communication required for synchronizing distances and frontier information grows significantly.

2. **Memory Consumption**: Each process maintains a copy of the entire graph structure, which limits scalability for very large graphs.

3. **Load Imbalance**: Despite METIS partitioning, some processes may handle vertices with higher connectivity, leading to workload imbalance.

4. **Thread Synchronization**: In the hybrid implementation, thread synchronization overhead increases with thread count, particularly for the graph update phases.

8. Conclusion and Future Work

8.1 Summary of Findings

This project has successfully implemented and evaluated parallel approaches for updating multi-source shortest paths in large dynamic networks. The key findings include:

1. The MPI implementation achieves significant speedup over the sequential approach, with a maximum speedup of 38.92× using 64 processes.

2. The hybrid MPI+OpenMP implementation further improves performance, reaching a speedup of 64.18× with 64 processes and 4 threads per process.

3. Both parallel implementations correctly maintain solution quality while drastically reducing computation time.

4. The effectiveness of parallelization varies with graph characteristics, with better results for graphs with higher computation-to-communication ratios.

5. METIS-based graph partitioning is essential for load balancing and minimizing communication.

8.2 Benefits of Hybrid Parallelism

The hybrid MPI+OpenMP approach demonstrated several advantages:

- Better utilization of hierarchical memory structures in modern computing clusters
- Reduced MPI communication overhead through shared-memory parallelism
- More flexible resource allocation depending on the graph characteristics
- Superior performance for compute-intensive phases of the algorithm

8.3 Future Directions

Several promising directions for future work include:

1. **GPU Acceleration**: Implementing key parts of the algorithm using CUDA or OpenCL to leverage GPU parallelism for further acceleration.

2. **Dynamic Load Balancing**: Developing techniques to rebalance work across processes as the graph structure changes.

3. **Memory Optimization**: Implementing distributed graph storage to handle even larger graphs that don't fit in the memory of individual nodes.

4. **More Objectives**: Extending the implementation to handle more than two objectives and evaluating the scalability of this approach.

5. **Streaming Updates**: Modifying the algorithm to handle continuous streams of graph updates instead of batch processing.

6. **Integration with Graph Databases**: Developing interfaces to integrate with popular graph database systems to enable practical applications.
Dagum, L., & Menon, R. (1998). "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering, 5(1), 46-55.

Appendix

A. Source Code Repository

The complete source code for this project is available at:

B. Sample Terminal Output

B.1 MPI Implementation (16 processes)

Parallel MOSP Update Implementation with MPI (16 processes)

Reading graph from file test_graph.txt...
Graph reading time: 2.345678 seconds
Partitioning graph for 16 processes...
Graph partitioned with edge-cut: 483752
Graph partitioning time: 4.567890 seconds
Process 0 has 62518 vertices, 4382 boundary vertices
Process 1 has 62483 vertices, 4271 boundary vertices
...
Computing initial SOSP trees in parallel...
Initial SOSP computation time: 8.765432 seconds
Generating 5000 'smart' random edges to insert...
Edge generation time: 0.345678 seconds
Running MOSP_Update in parallel...
SOSP1 update time: 6.543210 seconds
SOSP2 update time: 6.789012 seconds
Combined graph creation time: 0.987654 seconds
Final SOSP computation time: 11.234567 seconds
Vertices reachable from source in MOSP: 531924

MOSP result (path from source 0 to vertex 100):
Path: 0 -> 24 -> 36 -> 87 -> 100
Objective 1 distance: 543
Objective 2 distance: 712

MOSP_Update completed in 25.554443 seconds
Total program execution time: 42.308765 seconds

B.2 Hybrid Implementation (8 processes × 4 threads)

Hybrid MPI+OpenMP configuration: 8 MPI processes, 4 OpenMP threads per process
Reading graph from file test_graph.txt...
Graph reading time: 2.456789 seconds
Partitioning graph for 8 processes...
Graph partitioned with edge-cut: 382614
Graph partitioning time: 4.123456 seconds
Process 0 has 125036 vertices, 6784 boundary vertices
Process 1 has 124982 vertices, 6632 boundary vertices
...
Computing initial SOSP trees with hybrid parallelization...
Initial SOSP computation time: 9.012345 seconds
Generating 5000 'smart' random edges to insert...
Edge generation time: 0.321098 seconds
Running MOSP_Update with hybrid MPI+OpenMP parallelization...
SOSP1 update time: 6.789012 seconds

SOSP2 update time: 6.901234 seconds
Combined graph creation time: 0.876543 seconds
Final SOSP computation time: 11.345678 seconds
Vertices reachable from source in MOSP: 531924

MOSP result (path from source 0 to vertex 100):
Path: 0 -> 24 -> 36 -> 87 -> 100
Objective 1 distance: 543
Objective 2 distance: 712

MOSP_Update completed in 25.912467 seconds
Total program execution time: 37.824155 seconds