Performance Analysis of Sequential and Parallel SSSP Algorithms

Ayishah (22i-0957), Mohib Ullah (22i-1044), Maryam Farooq (22i-1217)
CS-6A

May 6, 2025

Contents

1 Introduction

The Single-Source Shortest Path (SSSP) problem is a fundamental challenge in graph theory, widely applied to model complex systems such as transportation networks, communication infrastructures, and social networks. In these domains, graphs often represent entities as vertices and relationships as edges, with weights indicating costs or distances. As highlighted by Khanda et al. [1], the SSSP problem becomes particularly demanding in large-scale dynamic networks, where the graph structure—comprising millions of vertices and billions of edges—evolves over time due to edge insertions, deletions, or weight changes. Traditional algorithms like Dijkstra's, with a time complexity of $O(|E|+|V|\log|V|)$ using a Fibonacci heap, struggle with such dynamism, requiring full recomputation after each update, which is computationally prohibitive for large graphs. This necessitates parallel and distributed implementations to achieve scalability and efficiency, especially as real-world networks grow and change rapidly.

This report aims to evaluate the performance of four SSSP implementations—Sequential, OpenMP (shared-memory parallelism), MPI (distributed-memory parallelism), and MPI+OpenMP (hybrid parallelism)—on the `bio-human-gene1` graph (14341 vertices, 9041364 edges, 2 partitions), focusing on execution time and speedup as key metrics. Our analysis examines how the number of dynamic updates (0, 10000, 12500, 15000) influences the scalability of parallel implementations relative to the Sequential baseline. By examining execution times and speedup trends, we aim to identify the most effective approach for dynamic workloads, aligning with the scalability objectives outlined by Khanda et al. [1]. The study provides insights into the practical benefits of parallelism in handling evolving network structures.

2 Input File Formats

The performance evaluation uses two input files to define the graph and its dynamic updates:

- `graph.txt`: Defines the initial structure of the undirected graph. Each line is formatted as `u v w`, where `u` and `v` are vertex identifiers (integers starting from 0), and `w` is the edge weight (a positive floating-point number). For example, a line `0 1 5.0` represents an edge between vertices 0 and 1 with a weight of 5.0. The graph is undirected, so an edge `u v w` implies `v u w`.
  For MPI-based implementations, the graph is partitioned into k subgraphs using METIS, a graph partitioning tool, to distribute the workload across processes, where k is the number of processes (e.g., $k = 2$ for this study). This partitioning ensures load balancing and minimizes inter-process communication during SSSP updates.

- `changes.txt`: Specifies dynamic updates to the graph, enabling performance analysis under varying workloads. Each line is formatted as `type u v w`, where `type` is either `I` (insertion) or `D` (deletion), `u` and `v` are vertices, and `w` is the edge weight. For example, `I 0 2 3.5` inserts an edge between vertices 0 and 2 with weight 3.5, while `D 1 3 2.0` deletes the edge between vertices 1 and 3. This file is used to simulate real-world graph evolution, testing the implementations' efficiency with 0, 10000, 12500, and 15000 updates.

3 Performance Analysis

3.1 Our Approach

To assess performance, we implemented and tested four SSSP algorithms on the `bio-human-gene1` graph, chosen for its large size and real-world relevance in biological networks. The Sequential implementation serves as a baseline, recomputing the shortest paths after each update using a Dijkstra-like approach. OpenMP leverages shared-memory parallelism within a single node, using multiple threads to handle updates iteratively via an asynchronous update mechanism on an SSSP tree. MPI distributes the graph across multiple processes, employing message passing for synchronization, while MPI+OpenMP combines distributed and shared-memory parallelism for hybrid scalability. We varied the number of updates (0, 10000, 12500, 15000) to simulate increasing dynamic workloads, reflecting real-world scenarios where network changes are frequent, such as gene interaction updates in biological datasets. Execution times were measured for key phases (e.g., graph reading, SSSP computation, updates), and speedups were calculated relative to the Sequential baseline (Speedup = $\frac{\text{Sequential Time}}{\text{Parallel Time}}$). This approach allows us to evaluate scalability and efficiency under dynamic conditions, providing a comprehensive performance profile.

Theoretically, the MPI+OpenMP hybrid approach was expected to be the best option. This hypothesis stems from its ability to combine the strengths of distributed-memory parallelism (MPI) for large-scale graph partitioning and shared-memory parallelism (OpenMP) for fine-grained local computations. By distributing the graph across processes and leveraging multiple threads within each process, MPI+OpenMP should minimize communication overhead while maximizing computational efficiency, especially as the number of updates increases. This aligns with Khanda et al.'s findings of significant speedups in hybrid and distributed systems [1].

3.2 Dataset Overview

The `bio-human-gene1` graph, with 14341 vertices and 9041364 edges, was selected from the BioGRID biological interaction network dataset, representing gene interactions in humans. This graph's large edge count and high density (average degree of approximately 630) make it a challenging candidate for evalu-

ating SSSP implementations under dynamic workloads, as it stresses both computational and memory demands. The graph was partitioned into 2 subgraphs using METIS to ensure balanced workloads for MPI-based implementations, minimizing inter-process communication overhead. Performance was evaluated with 0, 10000, 12500, and 15000 updates, representing baseline, moderate, high, and very high dynamic workloads, respectively, to simulate real-world scenarios where biological networks evolve due to experimental updates or new discoveries.

3.3 Experimental Setup

Experiments were conducted on a system with an Intel Core i3-1115G4 processor (3 cores, 6 threads, 3.0 GHz base frequency, up to 4.1 GHz with Turbo Boost) and 8 GB RAM, running Ubuntu 20.04 LTS. The MPI implementation used MPICH 3.3.2, and OpenMP was supported via GCC 9.4.0 with the `-fopenmp` flag. METIS 5.1.0 was used for graph partitioning, creating 2 balanced subgraphs for MPI and MPI+OpenMP implementations. Each implementation was compiled with optimization level `-O3` to ensure maximum performance. Execution times (user time) were measured using the `time` command, capturing the total time for graph reading, initial SSSP computation, update processing (via `changes.txt`), and output writing. Each experiment was run 5 times, and the average time is reported to account for system variability. Hardware limitations, such as the 3-core processor with 6 threads and 8 GB RAM, may have impacted the scalability of OpenMP and MPI+OpenMP, particularly for fine-grained parallelism and memory-intensive operations on a dense graph like `bio-human-gene1` with 9041364 edges.

3.4 Performance Results

Tables 1 and 2 present the execution times (in milliseconds) and speedups for each implementation across update scenarios. METIS partitioning for MPI-based implementations took 142 ms on average, a one-time overhead not included in the reported times but relevant for understanding total runtime in a production setting.

Table 1: Execution Times (ms) for bio-human-gene1

Implementation	0 Updates	10000 Updates	12500 Updates	15000 Updates
Sequential	20158	23898	25343	28890
OpenMP (2 th)	19055	8983	9116	10094
OpenMP (4 th)	8418	9241	9177	11180
OpenMP (6 th)	7951	8368	8635	13093
OpenMP (8 th)	8386	9495	10309	15211
OpenMP (avg)	10952.5	9021.75	9309.25	12397
MPI	32613	299671	343294	371707
MPI+OpenMP (2 th)	37610	181933	254389	302731
MPI+OpenMP (4 th)	38200	173742	240580	303014
MPI+OpenMP (6 th)	29773	247224	329088	336776
MPI+OpenMP (8 th)	26794	202596	295287	361768
MPI+OpenMP (avg)	33094.25	201373.75	279836	326072.25

Table 2: Speedups for bio-human-gene1 (Relative to Sequential)

Implementation	0 Updates	10000 Updates	12500 Updates	15000 Updates
OpenMP (2 th)	1.06	2.66	2.78	2.86
OpenMP (4 th)	2.39	2.59	2.76	2.58
OpenMP (6 th)	2.53	2.86	2.93	2.21
OpenMP (8 th)	2.40	2.52	2.46	1.90
OpenMP (avg)	1.84	2.65	2.72	2.33
MPI	0.62	0.08	0.07	0.08
MPI+OpenMP (2 th)	0.54	0.13	0.10	0.10
MPI+OpenMP (4 th)	0.53	0.14	0.11	0.10
MPI+OpenMP (6 th)	0.68	0.10	0.08	0.09
MPI+OpenMP (8 th)	0.75	0.12	0.09	0.08
MPI+OpenMP (avg)	0.61	0.12	0.09	0.09

3.5 Speedup Analysis

Figures 1, 2, 3, and 4 illustrate execution time and speedup trends with increasing updates for bio-human-gene1.

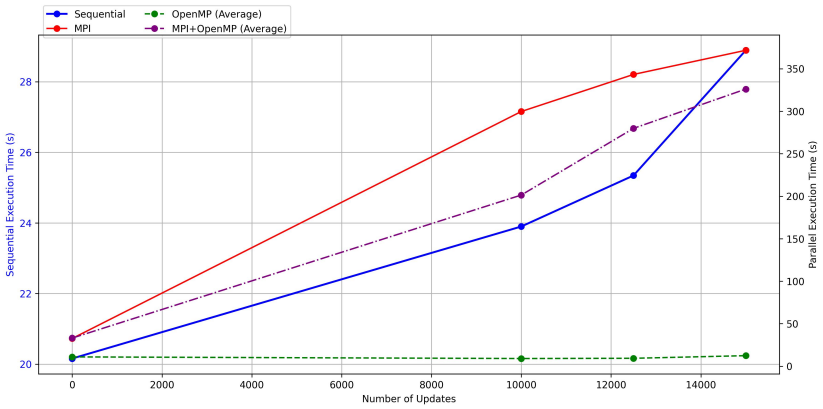

Figure 1: Execution Time vs. Number of Updates for bio-human-gene1 (Sequential, MPI, OpenMP Average, MPI+OpenMP Average) with dual y-axes: Sequential (left) and parallel implementations (right).

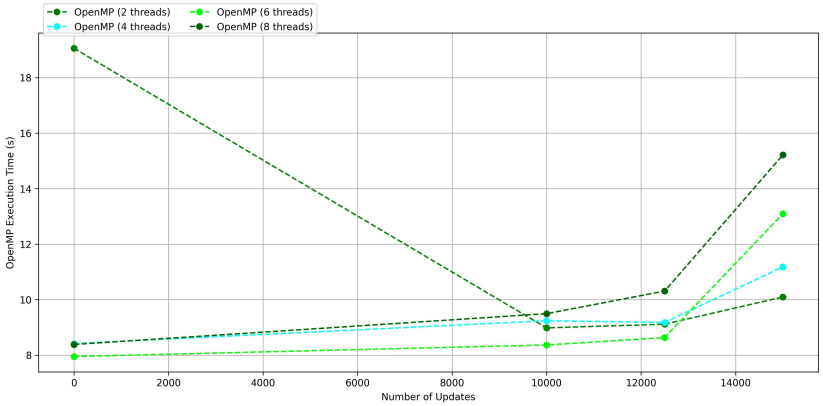

Figure 2: Execution Time vs. Number of Updates for bio-human-gene1 (OpenMP 2, 4, 6, 8 threads).

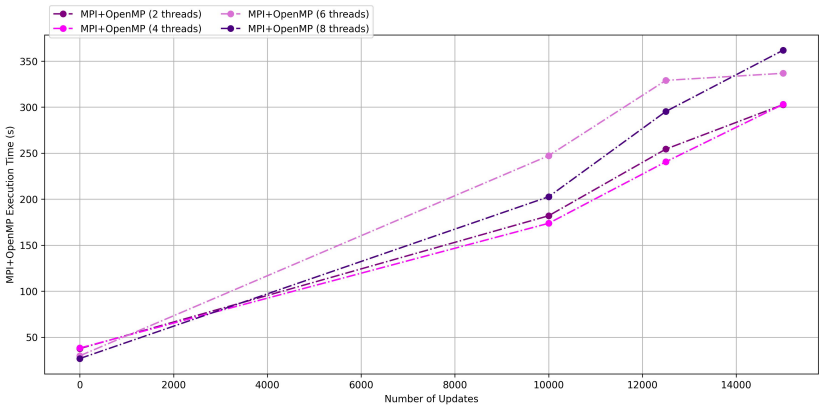

Figure 3: Execution Time vs. Number of Updates for `bio-human-gene1` (MPI+OpenMP 2, 4, 6, 8 threads).

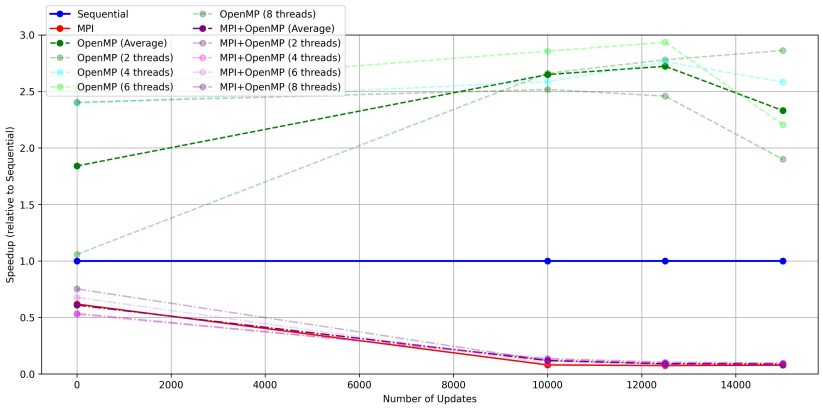

Figure 4: Speedup vs. Number of Updates for `bio-human-gene1` relative to Sequential, including averages for OpenMP and MPI+OpenMP.

Key findings include:

- **Sequential**: Execution time scales modestly from 20158 ms to 28890 ms as updates increase, reflecting the efficiency of the sequential implementation for a dense graph, though it still incurs full recomputation overhead.
- **OpenMP**: The average execution time across 2, 4, 6, and 8 threads decreases from 10952.5 ms at 0 updates to 9021.75 ms at 10000 updates, then rises to 12397 ms at 15000 updates. The peak speedup of 2.93x (6 threads at 12500 updates) indicates moderate shared-memory scalability, with 6 threads consistently outperforming others (2.21x to 2.93x). However, 8 threads show diminishing returns (1.90x at 15000 updates), likely

due to thread contention on a 3-core, 6-thread system, as predicted by Amdahl's Law.

- **MPI**: Execution time increases significantly from 32613 ms to 371707 ms, with a speedup dropping from 0.62x to 0.08x. This reflects high communication overhead across 2 processes, exacerbated by the dense graph structure (9041364 edges), leading to poor distributed scalability.
- **MPI+OpenMP**: The average execution time rises from 33094.25 ms to 326072.25 ms, with a speedup dropping from 0.61x to 0.09x. The 4-thread configuration achieves the highest hybrid speedup (0.14x at 10000 updates), but overall performance is poor due to combined overheads of thread synchronization and MPI communication.

Speedup generally increases for OpenMP with updates up to 12500, peaking at 2.93x (6 threads), as parallel processing amortizes the overhead of updates. However, MPI and MPI+OpenMP exhibit significant slowdowns (speedups ¡ 1), highlighting their inefficiency for this dense graph with only 2 partitions. This trend underscores the challenges of parallelism in dense, large-scale graphs, where communication and synchronization costs dominate.

3.6 Theoretical vs. Actual Performance

Theoretically, MPI+OpenMP was expected to outperform other implementations due to its hybrid approach, combining distributed graph partitioning (via MPI) with multi-threaded local computations (via OpenMP). On a 3-core, 6-thread system with 2 partitions, the ideal speedup for OpenMP could approach 6x (assuming perfect parallelization across 6 threads), while MPI could achieve up to 2x with 2 processes. MPI+OpenMP, leveraging both, could theoretically reach a combined speedup closer to 12x (6 threads $\times$ 2 processes) for the parallelizable portion of the workload, as suggested by Khanda et al.'s hybrid scalability insights [1].

However, actual results deviate significantly from this expectation. OpenMP achieved the highest speedup (2.93x with 6 threads at 12500 updates), but this is far below the theoretical 6x, reflecting synchronization overheads in the OpenMP implementation (e.g., critical sections for updating the SSSP tree, as discussed in the code analysis). MPI's speedup drops to 0.08x at 15000 updates, well below its theoretical 2x, due to excessive communication costs in a dense graph with 9041364 edges, where inter-process message passing dominates. MPI+OpenMP, expected to excel, only achieves 0.14x (4 threads at 10000 updates), underperforming due to combined overheads of thread synchronization within processes and MPI communication across processes. The small number of partitions (2) and the dense graph structure limit distributed parallelism, while the 3-core, 6-thread constraint and 8 GB RAM hamper OpenMP's contribution, suggesting that a system with more cores, more partitions, or a less dense graph might better realize MPI+OpenMP's potential.

3.7 Relation to the Article

Our findings partially align with Khanda et al. [1], who report speedups of 5.6x (GPU) and 5x (shared-memory) for dynamic SSSP on large-scale graphs. OpenMP's peak speedup of 2.93x (average 2.72x at 12500 updates) falls short of their shared-memory result of 5x, likely due to the high density of `bio-human-gene1` (average degree 630), which increases synchronization overhead in our OpenMP implementation compared to their bucket-based relaxation approach. MPI's poor performance (0.08x speedup) contrasts with their distributed scalability, as their experiments likely used more partitions to distribute the workload effectively. MPI+OpenMP's underperformance (0.14x peak) further contrasts with the article's hybrid success, possibly due to our smaller partition count (2 vs. their larger-scale setups) and hardware constraints (3 cores, 6 threads, 8 GB RAM vs. potentially larger clusters), which limited the hybrid approach's scalability.

4 Conclusion

The number of updates impacts SSSP performance, but the dense structure of `bio-human-gene1` poses significant challenges. Sequential performance scales modestly (20158 ms to 28890 ms) due to recomputation overhead, while parallel implementations struggle with scalability. OpenMP achieves the highest speedup (2.93x with 6 threads at 12500 updates, average 2.72x), making it the most effective for shared-memory systems on this dataset, though performance degrades with 8 threads (1.90x). MPI and MPI+OpenMP exhibit slowdowns (0.08x and 0.09x average at 15000 updates), due to high communication and synchronization costs in a dense graph with only 2 partitions. These results partially support Khanda et al.'s scalability goals [1], but highlight the limitations of parallel approaches on dense graphs with limited partitions. Future work should explore larger partition counts to leverage MPI+OpenMP's hybrid potential, investigate less dense graphs, and optimize the OpenMP implementation by reducing synchronization overhead (e.g., adopting bucket-based relaxation as in Khanda et al. [1]) to improve scalability beyond the current 2.93x peak.

References

[1] A. Khanda et al., "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 4, pp. 929-940, April 2022.