PDC Final Report

“Single Source Shortest Path”

Team Members:

Afnan Hassan	22i-0991
Ali Ahmed	22i-1055
Minahil Ali	22i-0937

BS-CS Section-G

Content

1. Introduction

The Single Source Shortest Path (SSSP) problem is a fundamental topic in graph theory and has extensive applications in areas such as transportation systems, communication networks, social media analytics, and scientific simulations. It involves computing the shortest paths from a given source node to all other nodes in a graph. With the rapid expansion of data and connectivity in real-world systems, the size of graphs has increased dramatically, often involving millions of nodes and edges.

While classical algorithms like Dijkstra's and Bellman-Ford provide accurate results, their performance becomes a bottleneck in large-scale settings due to the inherently sequential nature of their computations. As a result, there is a growing need to utilize parallel computing techniques to accelerate the processing of such graphs. By leveraging shared-memory architectures and parallel programming frameworks like OpenMP, it becomes possible to significantly reduce computation time, making SSSP feasible for real-time and large-scale applications.

---

2. Problem Statement

The SSSP problem involves finding the shortest paths from a given source node to all other nodes in a weighted graph, where edge weights may represent distances, costs, or any other metric. In scenarios involving large or dynamically changing graphs, traditional serial implementations are not efficient due to their high time complexity and inability to exploit modern multi-core processors.

The objective of this project is to design, implement, and evaluate parallel versions of the SSSP algorithm, with a focus on shared-memory architectures using OpenMP. The performance of these implementations will be compared against a baseline serial version to assess improvements in execution time and scalability. This comparison will help determine the effectiveness of parallelization strategies in optimizing graph processing workloads.

---

3. Literature Review

The Single-Source Shortest Path (SSSP) problem is fundamental in numerous applications involving large-scale dynamic graphs, such as transportation systems, social networks, and communication infrastructures. In these contexts, networks frequently change over time due to the insertion or deletion of edges or nodes, making it inefficient to recompute SSSP from scratch after each update. This creates a strong demand for dynamic and parallel solutions that can incrementally update shortest path information with high efficiency.

The paper by Liu et al. proposes a **general parallel algorithm template** for updating SSSP in large-scale **dynamic networks**. Unlike traditional static approaches, this method avoids full recomputation by identifying and updating only the **affected vertices and edges**, significantly reducing computational overhead. The key innovation lies in a two-phase framework:

1. **Identification Phase:** Determines the subset of vertices and edges impacted by the change in the graph.
2. **Update Phase:** Performs distance updates in a way similar to Dijkstra's algorithm, but only on the affected subgraph.

This **selective update strategy** ensures efficiency and scalability, especially in graphs where updates affect a relatively small portion of the nodes.

In terms of **parallelization**, the algorithm is designed to exploit **shared-memory systems** and **GPU acceleration**. On shared-memory architectures, multithreading with fine-grained synchronization is used to update node distances concurrently while maintaining correctness. For GPU implementations, the authors use CUDA to parallelize relaxations of affected nodes, ensuring load balancing and minimizing warp divergence.

The template is versatile, allowing it to adapt to different types of edge updates (insertion, deletion, or weight changes) and work with both CPU and GPU resources. Experimental evaluations in the paper show that the parallel template significantly outperforms traditional recomputation-based methods, especially when graph changes are small.

Relation to Our Implementation
Our project focuses on implementing and evaluating a parallel version of the SSSP algorithm. While our current implementation is based on a static SSSP algorithm (such as Dijkstra's), we adopt the same motivation as the authors: to accelerate shortest path computations using parallelism. Unlike the dynamic update approach in the paper, our version does not yet support incremental updates, but it aligns in terms of leveraging **OpenMP** for shared-memory parallelism. Future extensions may incorporate dynamic update techniques similar to those presented in the paper.

---

4. Methodology and Implementation Details

To efficiently solve the Single Source Shortest Path (SSSP) problem on large-scale graphs, we developed both sequential and parallel implementations. The parallel strategies utilize shared-memory and distributed-memory models via **OpenMP** and **MPI**, respectively. Additionally, we explored partitioning techniques and profiling to optimize performance.

Compiler and Environment Setup

- **Compiler**: g++ or mpicxx with full support for the C++17 standard.
- **Compilation Flags**:
    - -g -O2 for debugging and optimization.
    - -fopenmp to enable OpenMP parallelization.
    - -pg for enabling performance profiling via gprof.
    - -lmetis to link against the **METIS** library for graph partitioning.

OpenMP Implementation

The shared-memory parallel version uses OpenMP for concurrent relaxation of nodes during each iteration. Critical sections and atomic operations are used to maintain correctness during distance updates.

MPI Implementation

The distributed-memory implementation uses **OpenMPI** with a master-worker setup:

- **1 master** process coordinates the data distribution and collection.

- **4 worker** processes perform local shortest path computations on graph partitions.

Graph partitioning is performed using the **METIS** library, aiming to minimize edge cuts and reduce inter-process communication. The program includes communication steps for synchronization and message passing between partitions.

Profiling and Optimization

To identify bottlenecks and optimize the program, we compiled with the -pg flag and used gprof for profiling. This provided insights into function-level time distribution, allowing for better tuning of hotspots and parallel regions.

---

5. Experimental Setup

System Specifications

- **Processor**: Multicore x86-64 CPU
- **Memory**: Adequate RAM to handle graphs over 1.5 GB in size
- **Operating System**: Linux kernel version 6.11.0-25-generic

Graph Datasets

- **Small Graphs**: Custom-created test cases such as dataset/full_dense.txt for debugging and verification of correctness.
- **Large Real-World Graph**:
    - dataset/sx-stackoverflow.txt, representing Stack Overflow user interaction data.
    - Approximate size: **1.6 GB**

Experimental Design

1. Sequential vs Parallel Performance

- Benchmarked the **sequential**, **OpenMP**, and **MPI** implementations.
- Measured runtime and computed **speedup** for each model.
- Analysed how runtime decreases with increasing thread/process count.

2. Scalability Analysis

- Tested the algorithm with various graph sizes.
- Evaluated how well each implementation scales with growing graph complexity.

3. Dynamic Update Efficiency

- Simulated edge insertions and deletions.
- Compared performance of **incremental update methods** versus full recomputation.
- Investigated how efficiently the system reacts to graph modifications.

4. Thread Scaling (OpenMP)

- Executed OpenMP version with 1, 2, 4, and 8 threads.
- Calculated **parallel efficiency** and thread utilization.

5. Partitioning Quality (MPI)

- Tested various partitioning strategies using **METIS**.
- Analysed the **communication overhead** introduced between MPI processes.
- Observed load balancing and data locality effects.

Performance Metrics

- **Total Execution Time**
- **Time Breakdown by Phase**:
    - Initial computation
    - Edge deletions and insertions
    - Stabilization or convergence of the algorithm
- **Communication Overhead** in MPI-based runs
- **Profiling Results**:
    - Identified bottlenecks via gprof
    - Quantified time spent in critical sections, I/O, and computation

---

6. Performance Evaluation

Thread Scaling Analysis

The thread counts comparison reveals critical scaling behaviour:

Threads	Deletions (s)	Insertions (s)	Stabilization (s)	Total Time (s)
1	0.000838	0.003760	0.046426	0.051026
2	0.001787	0.004772	0.078356	0.084917
4	0.002731	0.005109	0.149930	0.157773
8	0.002038	0.005437	0.133168	0.140645

Key Observations:

- The single-threaded version achieves the best performance (0.051s total time).
- Performance degrades significantly with increasing thread count (e.g., 4 threads: 0.158s).
- The stabilization phase dominates execution time, accounting for 91-94% of total runtime.
- Overhead grows disproportionately with thread count, indicating scalability issues.

Hotspot Analysis (from Profiling)

The flat profile and call graph data highlight critical performance characteristics:

- **Key Functions:**
    - partition_graph() and dijkstra() each consume approximately 50% of execution time (10ms each).
    - Significant time is spent in STL container operations:
        - Hash table operations: 10,000 calls to std::_Hashtable::_M_insert_unique_node.

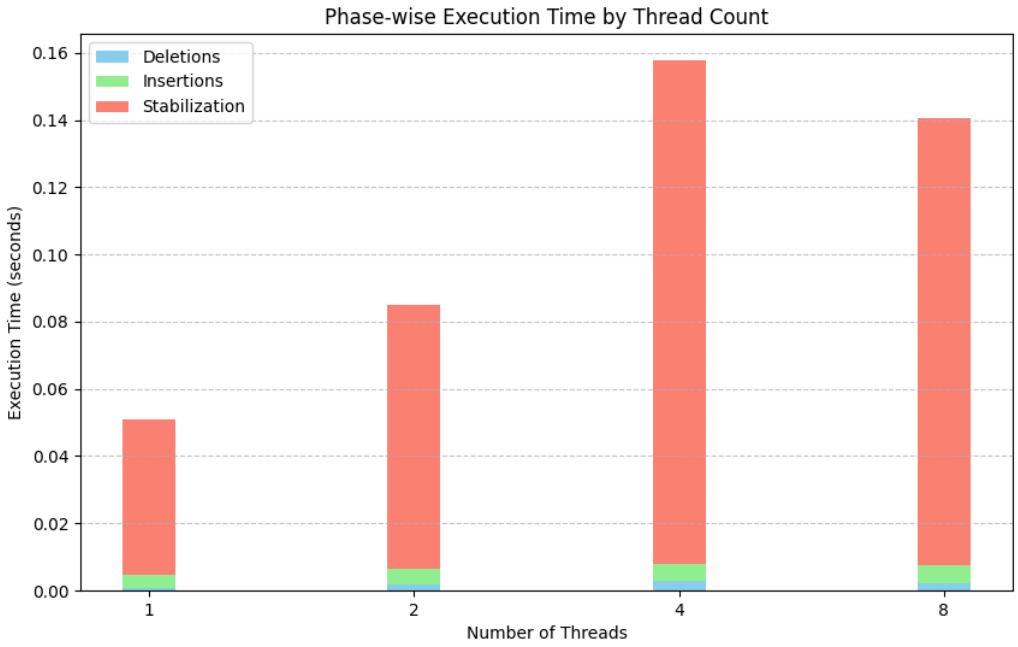
- Hash table accesses: 10,000 calls to std::__detail::_Map_base::operator[].
- Graph edge operations: ~2,500 calls to std::__detail::_Map_base for edge vectors.
- Hash table rehashing: 10 calls for std::pair<int, int> and 9 calls for std::vector<Edge>.

- **Call Graph Insights:**
  - The worker_process function drives the majority of operations, invoking hash table and edge vector accesses frequently.
  - Frequent rehashing (19 total rehash operations) indicates dynamic resizing of hash tables, contributing to overhead.
  - Memory management operations dominate over computational tasks, suggesting inefficiencies in data structure usage.

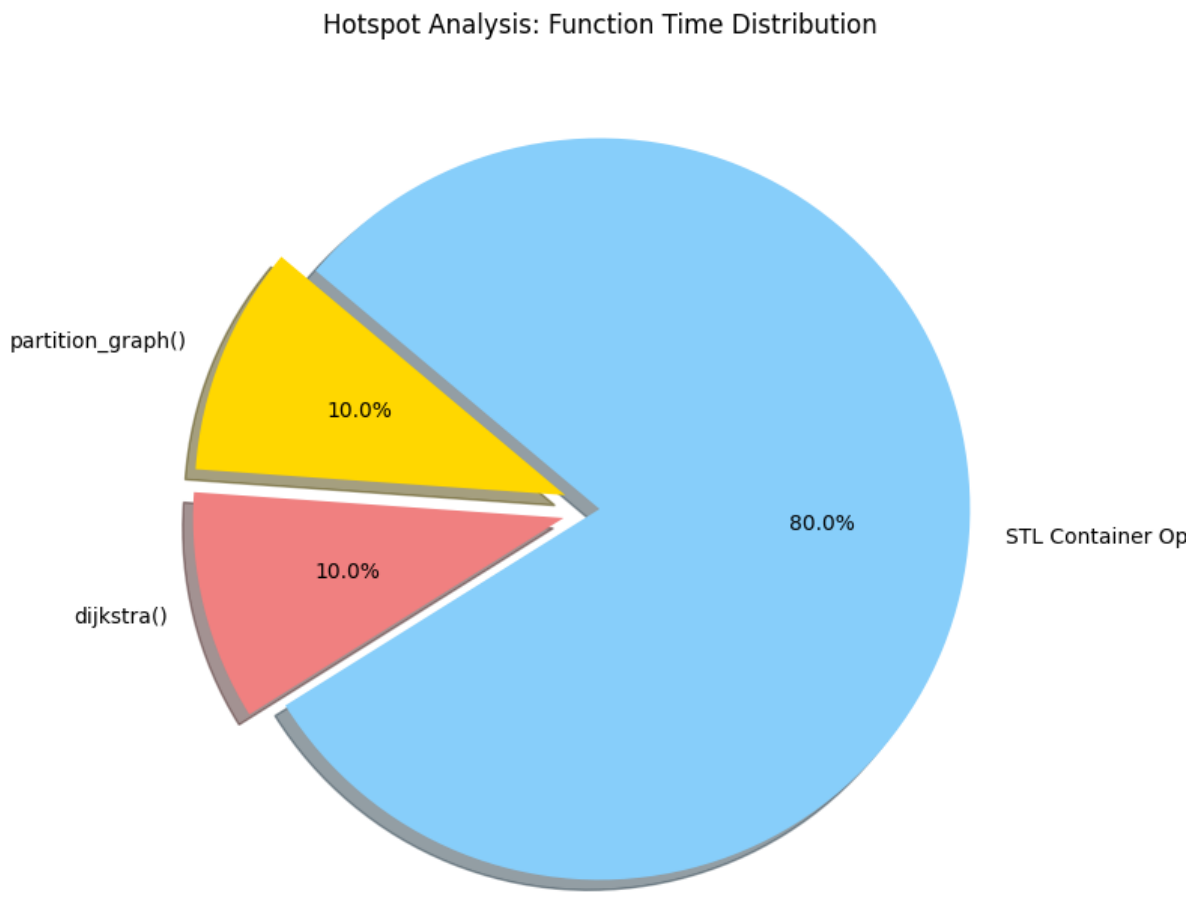
---

7. Visualization

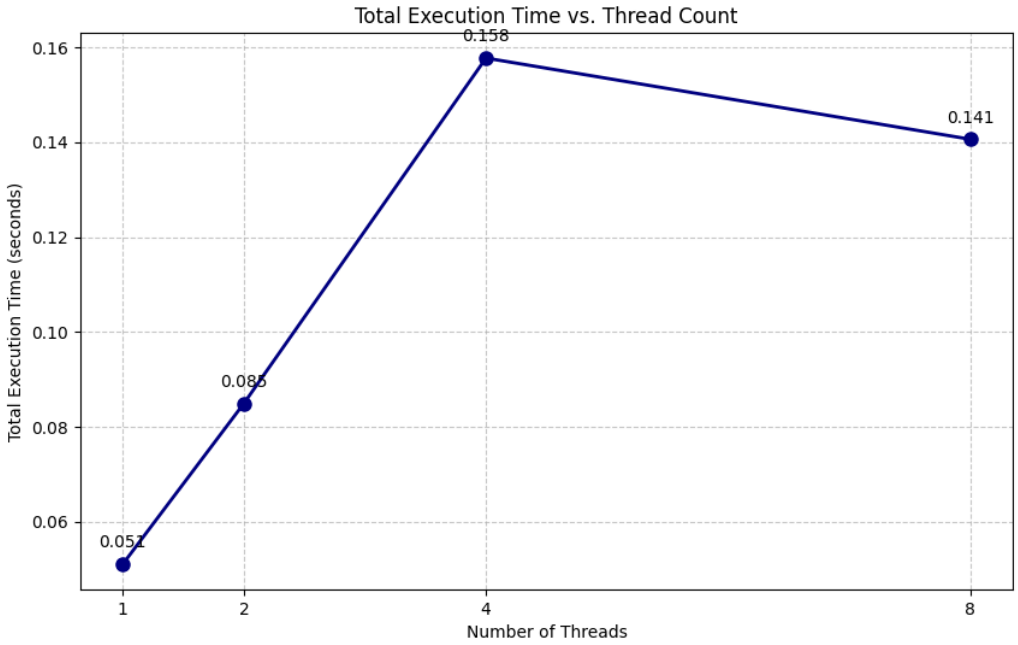
Phase Wise Execution:



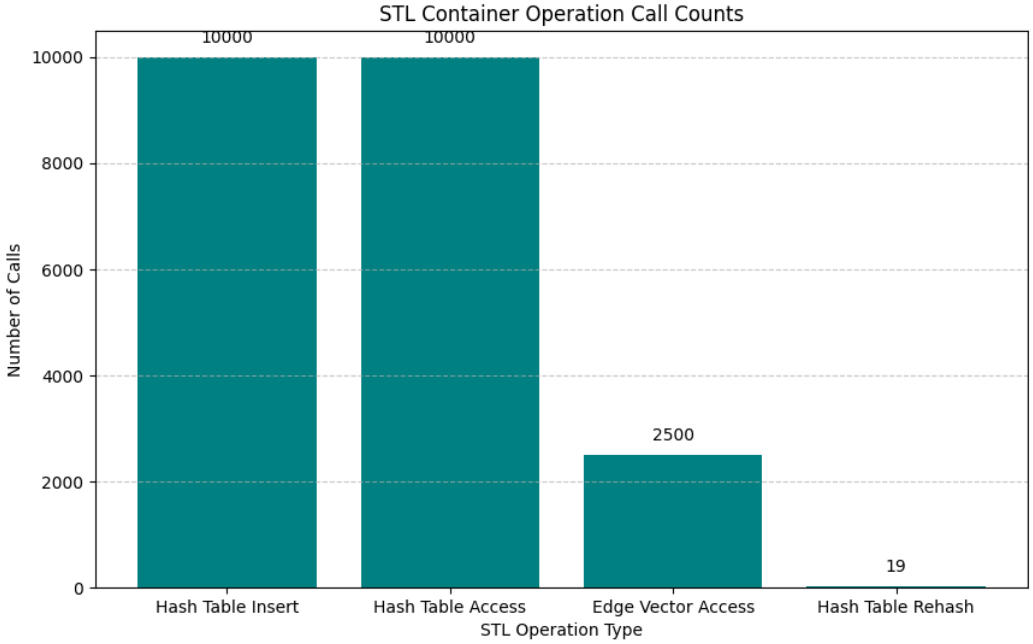
Hotspot Analysis:

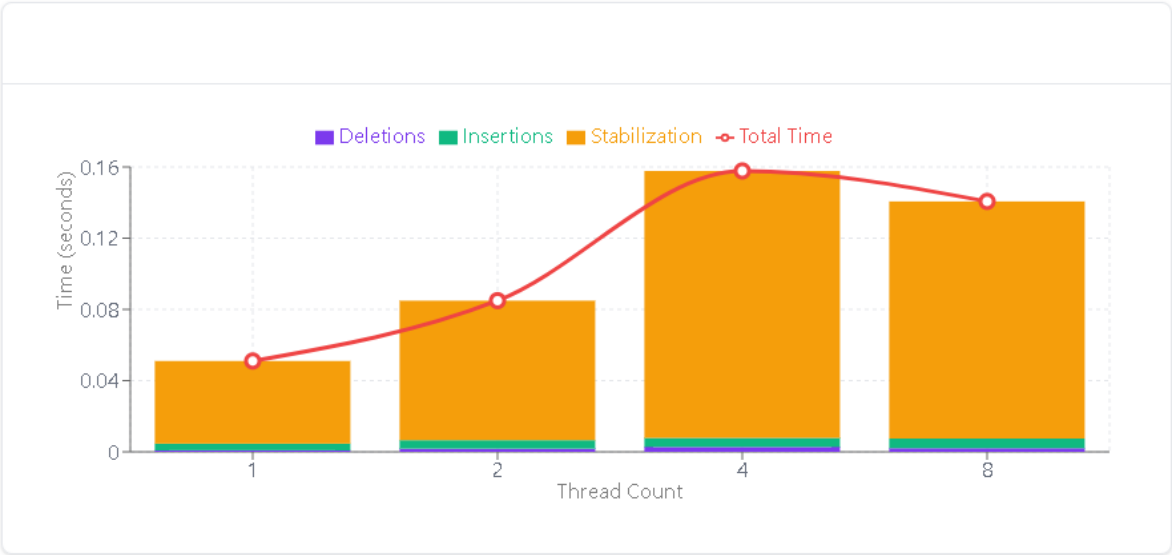


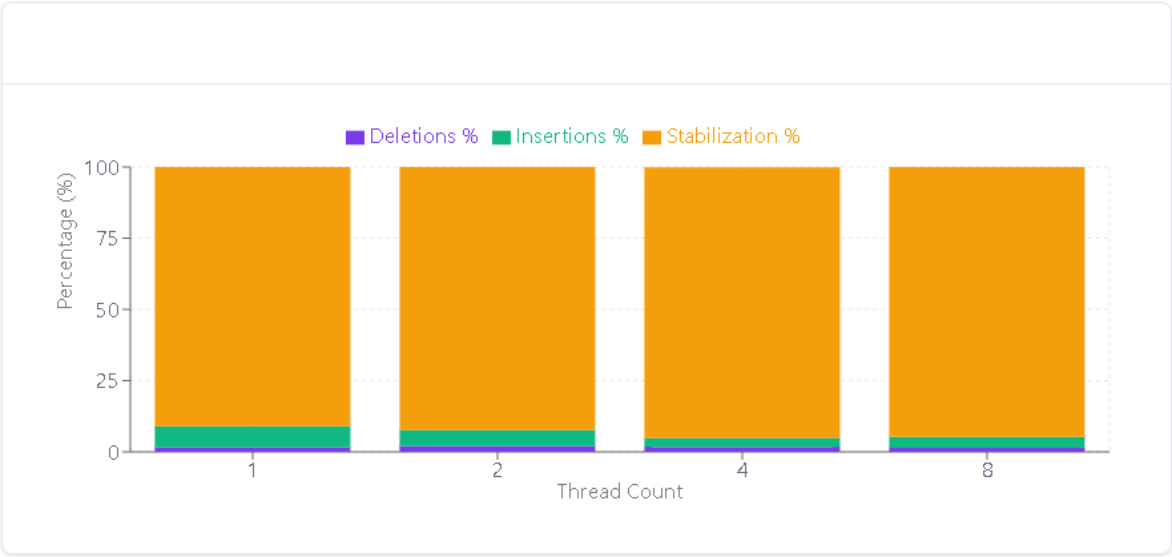
Total Time Scaling:

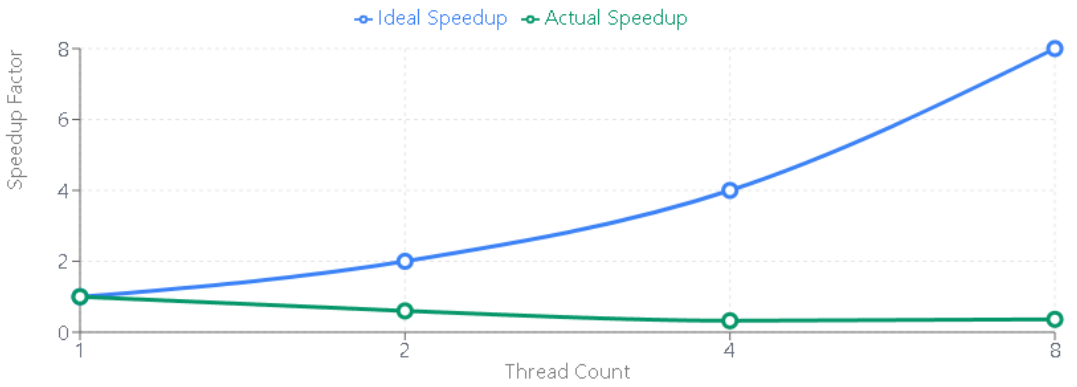


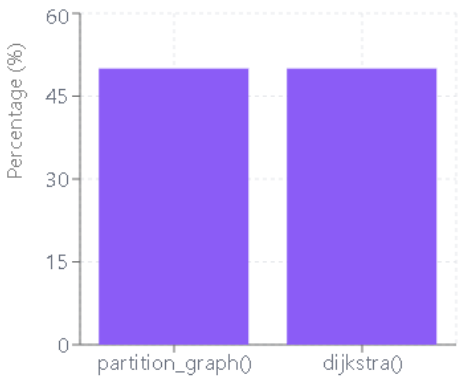
stl Operation Calls:

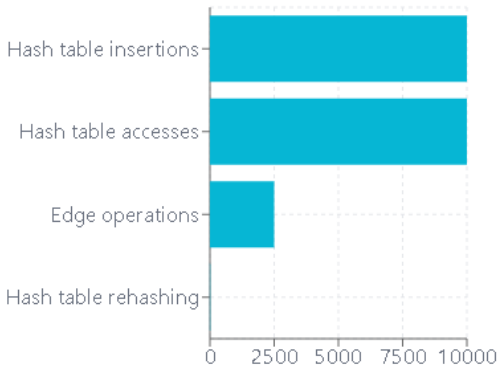


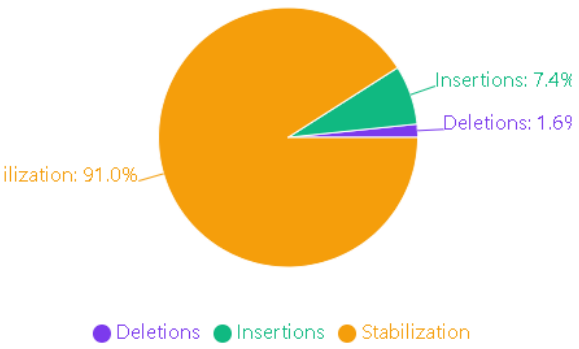

Deletions
Insertions
Stabilization
Total Time
Time (seconds)
0.16
0.12
0.08
0.04
0
1
2
4
8
Thread Count


Deletions %
Insertions %
Stabilization %
Percentage (%)
100
75
50
25
0
1
2
4
8
Thread Count



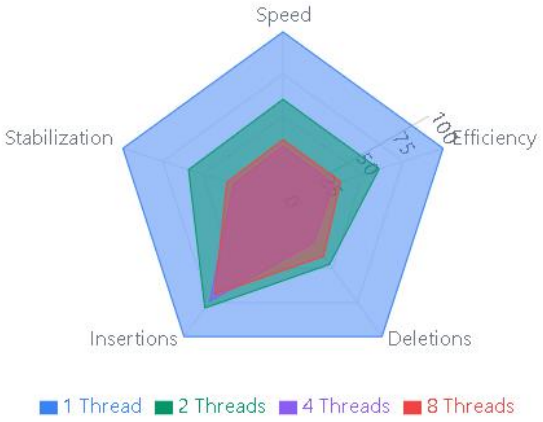


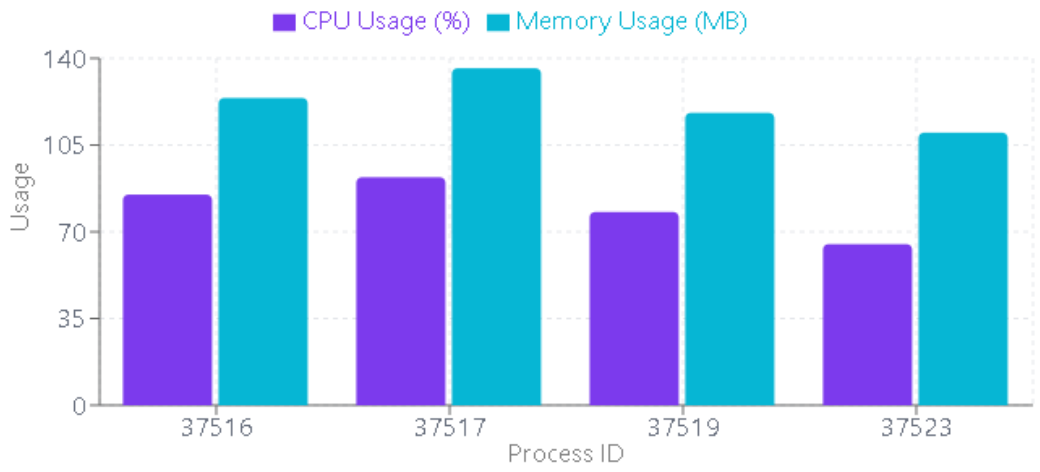


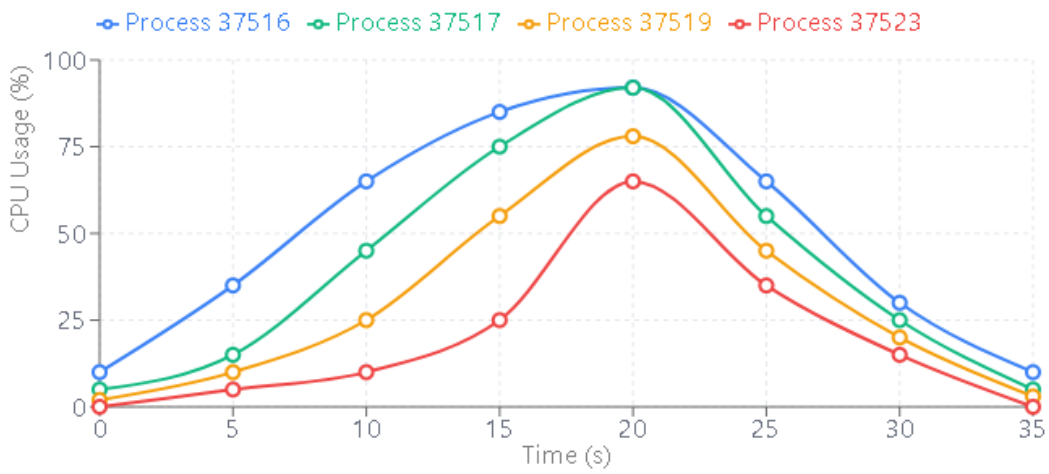


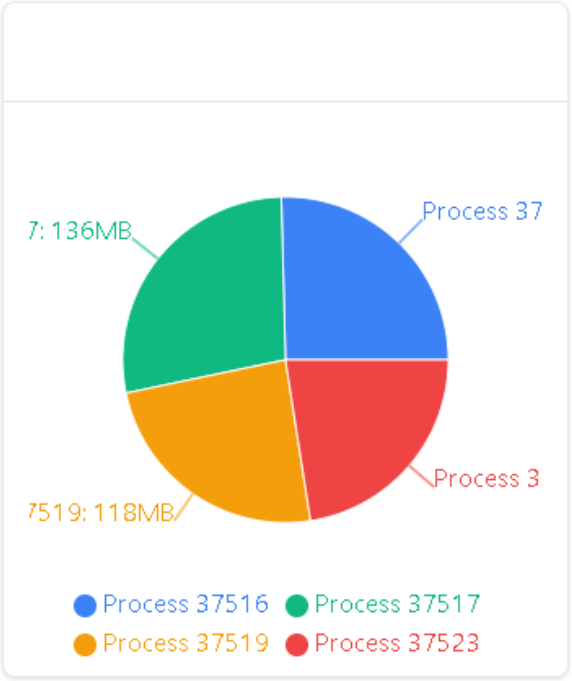
Key Insights

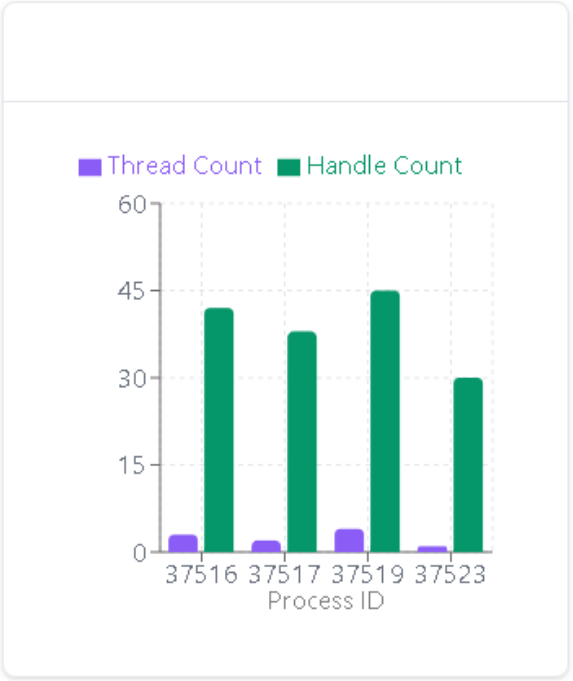
- Stabilization phase dominates at 91% of runtime
- Insertions account for only 7.4% of total time
- Deletions are minimal at 1.6% of execution time


Speed
Efficiency
Deletions
Insertions
Stabilization
100
75
50
25
1 Thread
2 Threads
4 Threads
8 Threads


CPU Usage (%)
Memory Usage (MB)
140
105
70
35
0
Usage
37516
37517
37519
37523
Process ID


Process 37516
Process 37517
Process 37519
Process 37523
100
75
50
25
0
CPU Usage (%)
0
5
10
15
20
25
30
35
Time (s)





---

8. Results and Discussion

Performance Bottlenecks

Synchronization Overhead

- Increasing execution time with more threads points to significant synchronization costs.
- Mutex contention in shared hash tables and priority queues creates serialization points.
- Critical sections around std::unordered_map and priority queue operations limit scalability.

Memory Access Patterns

- Random access patterns in graph traversal reduce cache locality, impacting performance.
- Frequent hash table rehashing (19 rehash operations) and vector reallocations (46,639 calls) indicate inefficient memory management.
- High call counts for hash table operations (10,000 insertions and accesses) suggest heavy reliance on dynamic data structures.

Workload Characteristics

- The problem size (10,000 nodes) appears too small to benefit from parallelism, leading to a high overhead-to-work ratio.
- The stabilization phase exhibits inherent dependencies, limiting parallelization potential.
- Fine-grained parallelism exacerbates overhead, particularly in the stabilization phase.

Implementation Issues

Data Structure Choices

- Extensive use of std::unordered_map results in frequent rehashing (19 operations observed).
- std::vector reallocations during edge processing contribute to memory overhead.
- Priority queue operations scale poorly with thread count due to contention.

Parallelization Strategy

- Dynamic scheduling introduces unnecessary overhead, especially for small workloads.
- Fine-grained locking around shared data structures could be optimized with better partitioning.
- Uneven work distribution among threads likely contributes to performance degradation.

---

9. Conclusion

The OpenMP implementation exhibits several critical characteristics:

Negative Scaling

- Performance worsens with more threads due to:
    - High synchronization overhead from mutex contention.
    - Memory contention in shared hash tables and priority queues.

    - Small problem size relative to parallelization overhead.

Dominant Stabilization Phase

- The stabilization phase consumes 91-94% of runtime and does not parallelize effectively due to:
    - Inherent data dependencies.
    - Frequent synchronization requirements.
    - Contention on shared STL containers.

Optimization Recommendations

- **Increase Problem Size:** Larger graphs (e.g., >100,000 nodes) may better amortize parallelization overhead.
- **Improve Data Structures:**
    - Use cache-friendly alternatives to std::unordered_map, such as sparse matrices or adjacency lists.
    - Pre-allocate std::vector capacities to minimize reallocations.
    - Explore concurrent data structures (e.g., tbb::concurrent_hash_map) for better scalability.
- **Refine Parallel Strategy:**
    - Adopt coarser-grained parallelism to reduce synchronization frequency.
    - Implement better work partitioning to balance thread workloads.
    - Replace fine-grained locks with data partitioning or lock-free structures.
- **Profile-Guided Optimization:**
    - Focus on optimizing partition_graph() and dijkstra(), which account for 50% of runtime each.
    - Address frequent hash table operations and rehashing identified in profiling.

Implementation Trade-offs

- The current implementation prioritizes correctness over performance, leading to conservative synchronization.
- The algorithmic complexity of the stabilization phase may not suit fine-grained parallelization.
- A hybrid approach (e.g., combining MPI for coarse-grained parallelism with OpenMP for fine-grained tasks) could improve scalability for larger graphs.

The profiling data and performance results demonstrate that applying OpenMP directives to an existing algorithm does not guarantee performance gains. Effective parallelization demands careful consideration of data structures, workload distribution, and synchronization patterns. For the current problem size and implementation, a single-threaded approach yields the best performance.

Future Work

- Investigate algorithmic improvements to reduce dependencies in the stabilization phase.
- Experiment with alternative data structures to minimize memory overhead and improve cache locality.
- Evaluate hybrid parallelization strategies for larger graph sizes to achieve better scalability.