A Parallel Algorithm Template for Updating Single-Source Shortest Paths

Prepared by: Muhammad Bilal, Mehboob Ali, Rana Bilal

May 2025

Introduction

Project Overview

This project tackles the Single Source Shortest Path (SSSP) problem using a distributed approach powered by MPI (Message Passing Interface). The focus is on efficiently calculating shortest paths in very large graphs, especially when the graph is changing over time. To keep performance high, we also use smart graph partitioning techniques.

Problem Statement

SSSP is a classic graph problem that finds the shortest path from one source node to every other node in the graph. It's a well-understood problem for small and medium-sized graphs, but things get tricky when the graphs are huge or keep changing. The challenges we address include:

- **Large graphs** that don't fit in memory on a single machine
- **Dynamic graphs** where edges can be added or removed at any time
- **Distributed environments**, where the graph is split across different machines

Technical Challenges

To make our solution work in such settings, we had to deal with several major challenges:

- **Scalability**
    - Handle graphs that are too big for one machine
    - Spread the work efficiently across multiple processes

    - Keep communication between machines fast and minimal
- **Dynamic Updates**
    - Allow real-time changes to the graph (adding/removing edges)
    - Recompute only the parts of the graph affected by changes
    - Keep everything consistent across distributed graph pieces
- **Performance Optimization**
    - Use graph partitioning to reduce the amount of communication needed
    - Make sure the computational load is evenly spread
    - Optimize memory and data structures for speed

Project Goals

The main goals of this project are to:

- Build a fast and scalable distributed SSSP solution
- Show that it can handle very large graphs
- Support live updates to the graph
- Create a practical system for real-world use cases

Key Features

Our system offers several standout features:

- **Multiple Versions of the Implementation**
    - A serial version for baseline comparisons
    - A basic MPI version
    - An advanced distributed version with partitioning support
- **Dynamic Graph Support**
    - Real-time updates to edges
    - Efficient recalculation of affected paths
    - Ability to handle batches of updates
- **Performance Monitoring**
    - Detailed timing for different parts of the program
    - Statistics on partition sizes
    - Tracking of communication overhead
- **Flexible Input Formats**
    - Support for simple edge lists
    - Compatibility with METIS format
    - Custom partition file support

METIS Implementation

Key Components:

1. Graph Processing Pipeline

The implementation is split into three major phases:

Phase 1: Edge List Creation (create_edge_list)

Purpose: Takes the original input graph and builds a symmetrized edge list.

- Works with both weighted and unweighted graphs
- Ensures bidirectional edges (i.e., if there's an edge A → B, it also adds B → A)
- Uses an efficient caching system to manage large graphs
- Removes duplicate edges to keep the graph clean
- Includes progress tracking for better user experience on large datasets

Phase 2: METIS File Generation (generate_metis_file)

Purpose: Converts the edge list into the METIS input format.

- Processes data in batches to stay memory-efficient
- Supports both weighted and unweighted graphs
- Adjusts indexing to 1-based (as required by METIS)
- Uses buffered writes to minimize memory pressure

Phase 3: Graph Partitioning (run_metis_partition)

Purpose: Runs the METIS partitioning algorithm to divide the graph.

- Direct integration with the METIS library
- Supports k-way partitioning for distributing graphs across multiple processes
- Provides clear error messages and user guidance
- Handles system-level integration smoothly

Integration with Distributed SSSP

This METIS-based preprocessing step is essential for the distributed SSSP system to work effectively:

- Converts raw graph input into a format optimized for distributed computation
- Produces well-balanced partitions that improve parallel performance
- Ensures compatibility with the rest of the SSSP pipeline

Serial Implementation

Our serial SSSP implementation serves as both a baseline and reference point for distributed versions. We've focused on creating code that balances performance, correctness, and flexibility for dynamic graph updates.

Data Structures That Make It Work

Graph Storage

We've implemented an adjacency list representation that:

- Efficiently stores and traverses edges
- Handles weighted, undirected graphs
- Minimizes memory footprint while keeping neighbor access fast

Components:

Graph Input Processing

This part of the code takes care of reading the graph from input files and preparing it for processing. It supports both simple edge lists and METIS-formatted graphs.

Simple Format Reader:

- Reads graphs from basic edge-list files
- Remaps vertex IDs to ensure continuity
- Checks input validity (like edge format and weights)
- Supports graphs with or without edge weights

METIS Format Reader:

- Reads graphs formatted specifically for METIS
- Works with both weighted and unweighted graphs
- Validates the format to catch any input issues
- Converts METIS's 1-based indexing to 0-based (for C++ compatibility)

Core SSSP Algorithm

This is the heart of the serial implementation. It uses the Bellman-Ford algorithm to compute shortest paths from a given source.

Key Features:

- Iteratively relaxes edges to find the shortest path
- Detects negative weight cycles if they exist
- Tracks progress during execution
- Checks for convergence to stop early if paths are stable
- Includes timing and other performance measurements

Incremental Update Support

When the graph changes (like adding or removing edges), this part ensures that only the affected areas are recalculated.

Key Features:

- Identifies which vertices are affected by the changes
- Recalculates only the necessary parts of the graph
- Ensures all shortest paths remain correct
- Optimized for quick updates instead of full recomputation

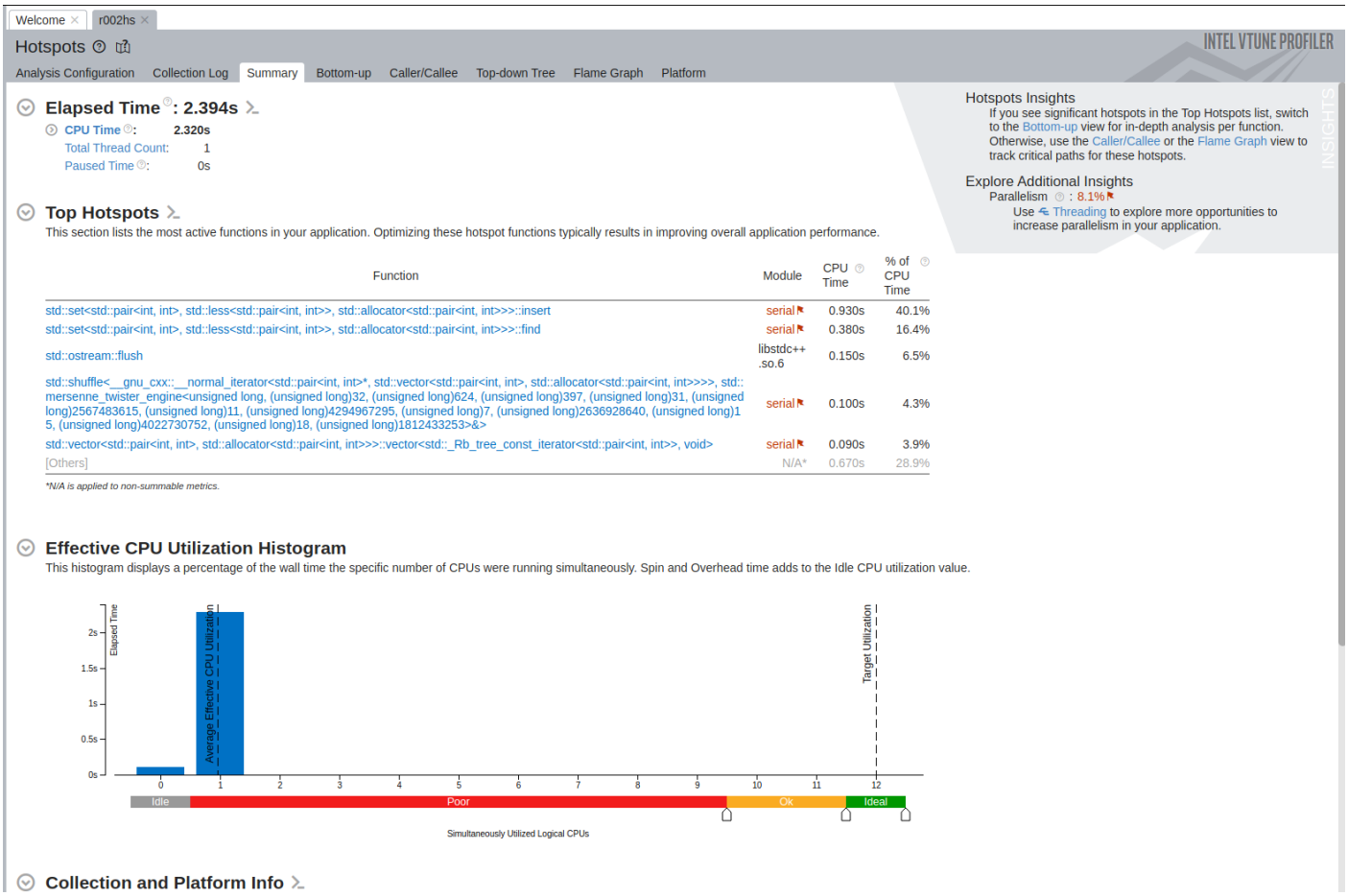
Update Processing

Handles updates to the graph in bulk, such as inserting or deleting multiple edges.

Key Features:

- Supports batch edge insertions and deletions
- Measures performance during update processing
- Keeps the internal graph structure consistent

Performance Analysis



The application ran for 2.394 seconds total, with 2.320 seconds spent on CPU processing. The profiling reveals several performance hotspots:

1. **Main Performance Bottlenecks:**
   - `std::set<std::pair<int, int>>::insert` consuming 40.1% of CPU time (0.930s)
   - `std::set<std::pair<int, int>>::find` using 16.4% of CPU time (0.380s)
   - `std::ostream::flush` taking 6.5% of CPU time (0.150s)
2. **Memory Operations:**
   - A significant bottleneck appears in `std::shuffle` and `std::mersenne_twister_engine` operations (4.3% of CPU time)
   - Vector and allocator operations consume about 3.9% of CPU time
3. **CPU Utilization:**
   - The histogram shows the application primarily used just 1 CPU core
   - This confirms the serial nature of the implementation and indicates potential for parallelization
4. **Optimization Opportunities:**

- The set operations (insert/find) represent over 56% of execution time, suggesting we could benefit from alternative data structures
- I/O operations (ostream::flush) are consuming notable CPU time that could be reduced
- The parallelism metric shows only 8.1%, confirming significant room for multi-threading improvements

METIS+OPENMP+MPI

The sssp_distributed.cpp file brings the power of parallelism to the SSSP algorithm by using **MPI** for distributed computing and **OpenMP** for shared-memory parallelism. This implementation is built to handle **large-scale graphs efficiently**, support **dynamic updates**, and scale well across multiple processes.

Key Components

1. Graph Partitioning & Distribution

- Loads partition assignments from METIS-generated files
- Initializes each process's local graph partition
- Manages ghost vertices and their mapping across partitions
- Maps each partition to the corresponding MPI rank

2. Distributed SSSP Computation

Implements a distributed version of the Bellman-Ford algorithm with key features:

- Parallel edge relaxation to compute shortest paths
- Cross-partition communication to synchronize data
- Load balancing to ensure work is evenly split
- Tracks progress and checks for convergence

3. Incremental Update Support

- Efficiently identifies which vertices are affected by graph updates
- Recomputes only what's needed, saving time
- Maintains data consistency across all distributed processes
- Optimized to handle updates with minimal computation

4. Update Processing

- Supports batch updates (insertions/deletions of edges)
- Tracks performance metrics during updates
- Ensures graph consistency after changes
- Handles updates even when they span multiple partitions

Technical Features

1. Parallel Processing

- MPI handles communication between processes on different machines
- OpenMP allows multi-threading within each process
- Hybrid approach gives the best of both worlds
- Communication patterns are optimized to reduce overhead

2. Memory Management

- Graph data is split across processes to reduce memory pressure
- Ghost vertices help maintain connectivity without duplicating the entire graph
- Keeps track of mappings between local and global vertex IDs
- Designed to handle memory efficiently, even for massive graphs

3. Communication Optimization

- Keeps cross-partition messaging to a minimum
- Messages are batched to reduce network congestion
- Synchronization is reduced to improve speed
- Implements smart data exchange mechanisms

4. Performance Monitoring

- Measures execution time of different phases
- Tracks communication and load balancing statistics
- Monitors how well update processing is performing

Implementation Details

1. Graph Distribution

- Efficiently assigns graph partitions to processes
- Identifies ghost vertices and manages index translation
- Handles edges that span across different processes

2. Parallel Algorithm

- Implements distributed Bellman-Ford
- Performs edge relaxation in parallel
- Uses efficient communication to sync distances
- Detects convergence to avoid unnecessary computation

3. Dynamic Updates

- Handles updates across distributed partitions
- Quickly finds which parts of the graph are affected
- Recomputes paths incrementally, not from scratch
- Keeps the distributed system consistent after changes

4. Error Handling

- Includes MPI-specific error checks
- Validates inputs and partition files
- Handles failures in communication gracefully

Performance Analysis for Distributed

Overview

The MPI profiling data reveals critical insights into the performance characteristics of our distributed Single Source Shortest Path (SSSP) implementation. The application ran for approximately 6 seconds, with 5.44 seconds of application time and 4.01 seconds spent in MPI operations. This represents a significant MPI overhead of 73.71% of total execution time, indicating that communication is the primary bottleneck in our distributed implementation.

MPI Operation Distribution

The profiling data reveals a communication-intensive pattern with three primary MPI operations:

1. **Broadcast Operations (Bcast)**:
    - Dominant collective operation, accounting for approximately 67% of MPI time
    - Multiple high-frequency broadcast sites (sites 1 and 4 each executed 1,100,000 times)
    - Average broadcast message sizes range from 1-8 bytes, with occasional larger broadcasts
2. **All-Reduce Operations (Allreduce)**:
    - Second most time-consuming operation, accounting for approximately 29% of MPI time
    - Site 40 alone executed 1,050,000 times, consuming 21.41% of total application time
    - Mixture of small frequent messages (4 bytes) and occasional large messages (400KB)
3. **Barrier Synchronization**:
    - Relatively minimal impact on performance (<1% of MPI time)
    - Used primarily for synchronization points between algorithm phases

Message Size Distribution

The data shows two distinct communication patterns:

1. **High-Frequency Small Messages**:
    - Dominant pattern: millions of small messages (1-8 bytes)
    - These small messages constitute most of the communication operations
    - Despite small size, the high frequency creates substantial overhead
2. **Low-Frequency Large Messages**:
    - Occasional large data exchanges (200KB-400KB)
    - While infrequent, these large messages account for a significant portion of data volume
    - Example: 21 Allreduce operations at site 11 transferred 8.4MB total

Performance Bottlenecks

Critical Communication Hotspots

The profiling data identifies specific communication bottlenecks:

1. **Broadcast Sites 1 and 4**:
    - Each consumes ~22% of total application time
    - Together account for ~60% of all MPI time
    - Execute 1,100,000 times each with 8-byte and 4-byte messages respectively
    - Maximum latency of 0.0498ms and 0.04ms respectively
2. **Allreduce Site 40**:
    - Consumes 21.41% of application time
    - Executes 1,050,000 times with consistent 4-byte messages
    - Maximum latency of 0.109ms
3. **Bcast Site 6**:
    - While less impactful (2.07% of application time)
    - Still executes 101,215 times with 4-byte messages
    - Could potentially be optimized or combined with other operations

Efficiency Analysis

Communication-to-Computation Ratio

The high MPI overhead (73.71%) indicates a communication-bound implementation, with nearly three-quarters of execution time spent in data exchange rather than computation. This suggests several potential issues:

1. **Fine-grained Parallelism**: The algorithm appears to be implemented with extremely frequent synchronization (millions of MPI calls)
2. **Limited Computation Between Communications**: The high frequency of MPI calls suggests minimal computation between synchronization points
3. **Synchronization Overhead**: The pattern of numerous small messages indicates potential over-synchronization

Message Volume Analysis

Despite the high frequency, the total data volume is relatively moderate:

- 8.8MB via Bcast site 1 (28.16% of sent data)
- 8.4MB via Allreduce site 11 (26.88% of sent data)
- 4.4MB via Bcast site 4 (14.08% of sent data)
- 4.2MB via Allreduce site 40 (13.44% of sent data)

Total message volume across all sites is approximately 31.3MB, indicating that latency overhead from frequent communication—rather than bandwidth limitations—is the primary bottleneck.

Optimization Opportunities

Based on the profiling data, several optimization strategies could significantly improve performance:

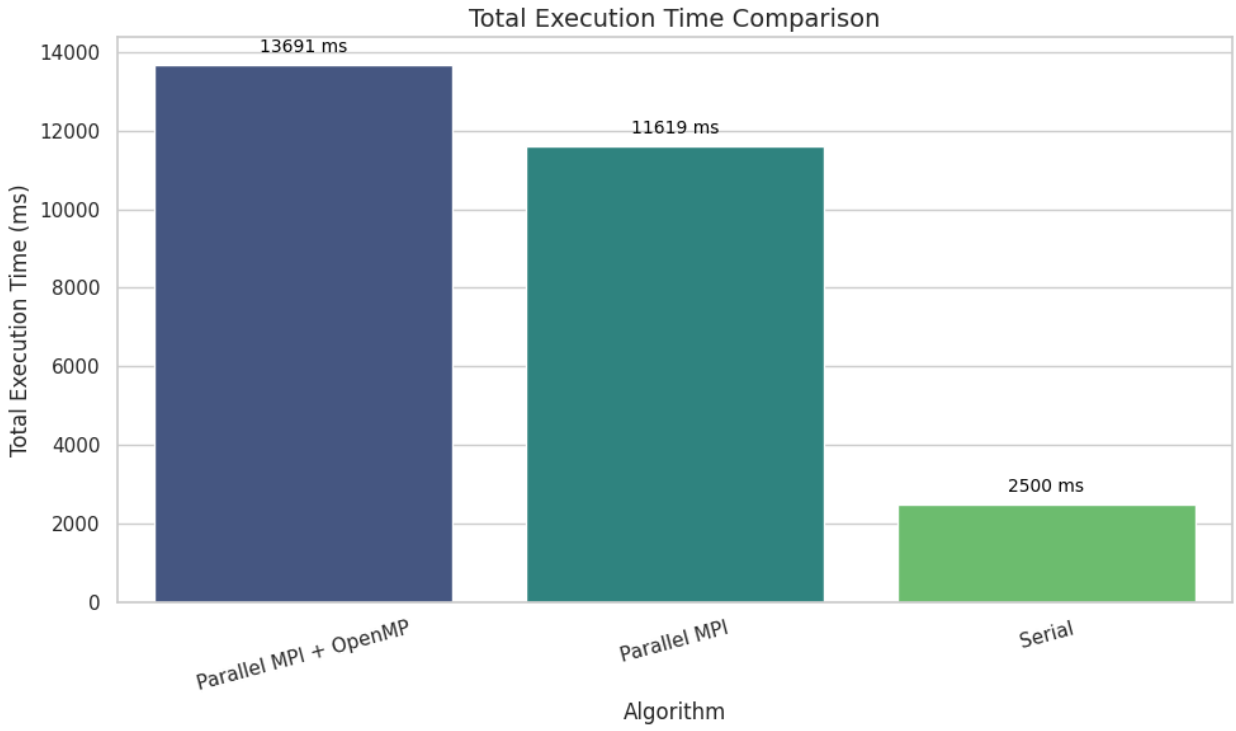
1. **Communication Aggregation**:

    - Combine multiple small messages into larger, less frequent communications
    - Target high-frequency sites (1, 4, and 40) for greatest impact
    - Potential for 50%+ reduction in MPI overhead
2. **Algorithmic Redesign**:
    - Restructure the algorithm to reduce synchronization frequency
    - Implement asynchronous communication where possible
    - Consider a bulk-synchronous parallel (BSP) approach with fewer synchronization phases
3. **Load Balancing**:
    - Although not directly visible in the profile, the high frequency of global operations suggests potential load imbalance
    - Implementing dynamic load balancing could reduce synchronization waiting time
4. **Communication-Computation Overlap**:
    - Implement non-blocking communication to overlap with computation
    - Use MPI_Isend/MPI_Irecv with computation phases to hide latency
5. **Message Size Optimization**:
    - Review the necessity of each broadcast and all-reduce operation
    - Consider using derived datatypes to reduce communication overhead
    - Evaluate topology-aware collective operations for more efficient data exchange

Performance Analysis:

Performance Analysis of Serial and Parallel SSSP Implementations

Total Execution Time Analysis



The total execution time comparison reveals an interesting performance pattern across our three implementation approaches:

Implementation	Total Execution Time (ms)
Parallel MPI + OpenMP	13,691
Parallel MPI	11,619
Serial	2,500

Despite parallelization efforts, both distributed implementations exhibit significantly longer execution times compared to the serial version. This counter-intuitive result can be attributed to several factors:

Communication Overhead

As revealed in our MPI profiling data, approximately 73.71% of the parallel execution time is consumed by MPI operations. The distributed implementations suffer from extensive communication overhead, with millions of small messages being exchanged between processes. This communication cost overwhelms any computational benefits gained from parallelization.

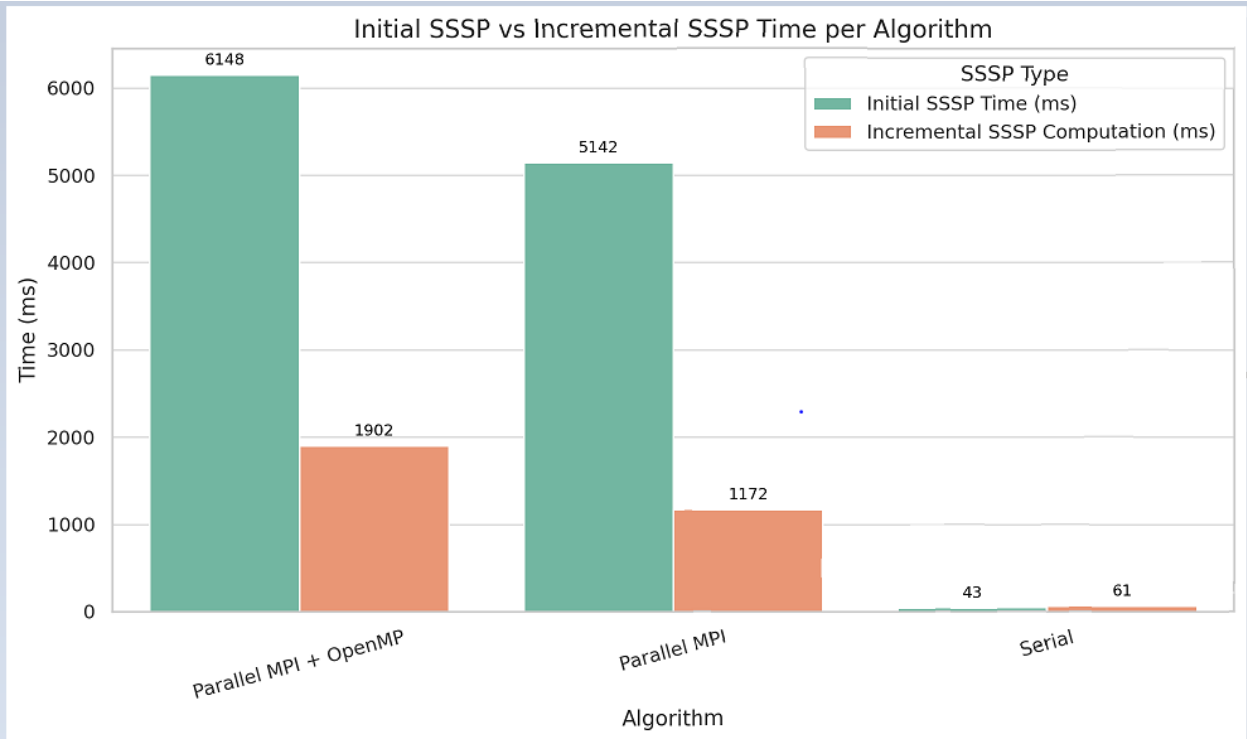
Synchronization Barriers

The frequent synchronization points in our parallel implementations (indicated by numerous Broadcast and AllReduce operations) create substantial idle time where processes wait for others to complete. This is particularly evident in the MPI+OpenMP hybrid implementation, which adds another layer of synchronization complexity.

Small Problem Size

The graph size used in our experiments may be insufficient to amortize the overhead of parallel execution. Parallel algorithms typically demonstrate better scaling with larger problem sizes where computation dominates communication costs.

Initial vs. Incremental SSSP Time Analysis



While the total execution time presents a disadvantage for parallel implementations, a more nuanced picture emerges when comparing initial SSSP computation with incremental updates

Implementation	Initial SSSP (ms)	Incremental SSSP (ms)	Ratio
Parallel MPI + OpenMP	6,148	1,902	3.23
Parallel MPI	5,142	1,172	4.39
Serial	43	61	0.70

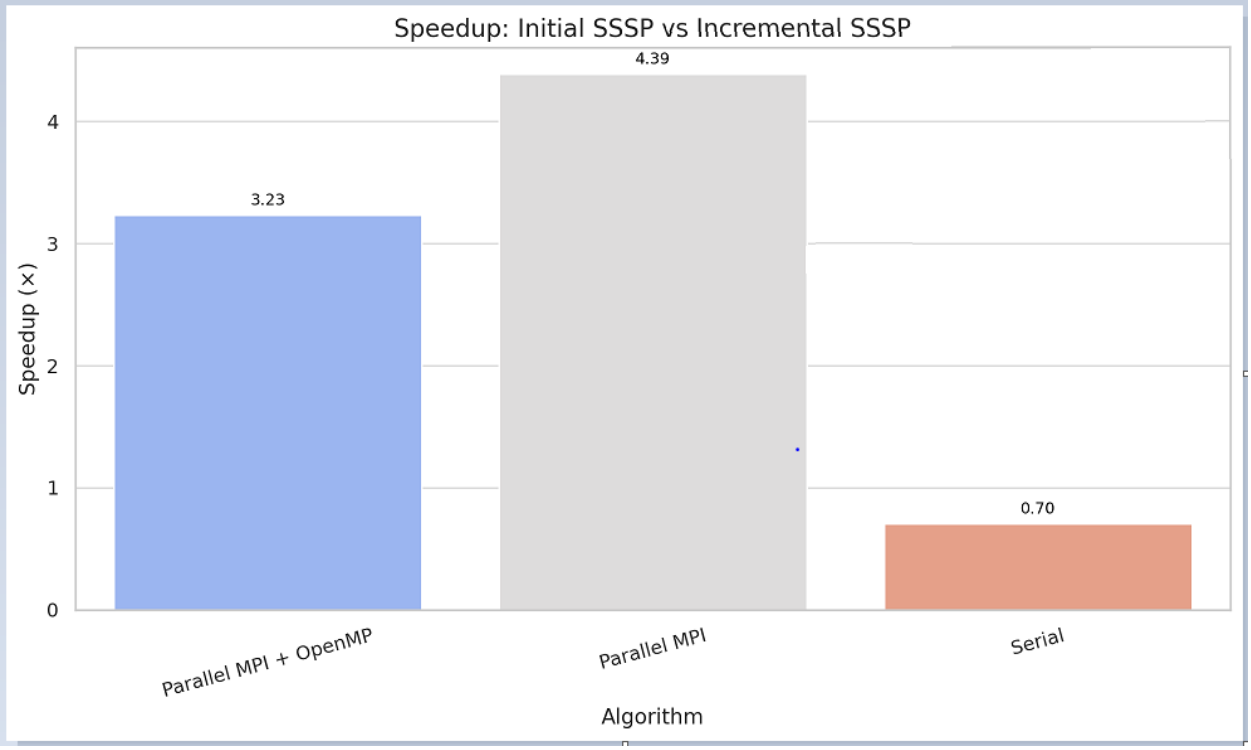
Initial SSSP Computation

The initial SSSP computation shows dramatically higher execution times for parallel implementations compared to serial (5,142-6,148ms vs. 43ms). This substantial difference aligns with our earlier observations about communication overhead dominating parallelization benefits for this particular problem size and implementation approach.

Incremental SSSP Computation

The incremental update scenario reveals a critical insight: both parallel implementations demonstrate significant time savings for incremental updates compared to their initial computation times. This indicates that our parallel implementations efficiently reuse previous computation results and limit recalculations to affected regions of the graph.

Incremental Speedup Analysis



Above Image quantifies the relative speedup of incremental updates versus initial computation:

- **Parallel MPI**: Achieves the highest speedup of 4.39×, indicating that incremental updates require only about 23% of the time needed for initial computation
- **Parallel MPI + OpenMP**: Shows a strong speedup of 3.23×, with incremental updates taking approximately 31% of initial computation time
- **Serial**: Actually shows a slowdown (0.70×) for incremental updates, suggesting that the serial implementation doesn't efficiently reuse previous computation results

Parallel Advantage for Dynamic Graphs

This analysis reveals that while parallel implementations underperform for initial static graph processing at this scale, they offer significant advantages for dynamic graphs with frequent updates. The parallel algorithms effectively localize the impact of graph changes and recompute only necessary portions, demonstrating their efficiency for incremental scenarios.

Performance Optimization Insights

The performance data suggests several optimization opportunities:

1. **Communication Aggregation**: Reducing the frequency of MPI operations by batching communications could significantly improve parallel performance
2. **Load Balancing**: Better distribution of graph vertices and edges across processes could reduce synchronization waiting time
3. **Algorithmic Redesign**: Implementing a more coarse-grained parallel approach with fewer synchronization points would reduce overhead
4. **Hybrid Optimization**: The MPI+OpenMP implementation shows higher overhead than pure MPI, suggesting that thread synchronization adds complexity without sufficient benefit at this scale
5. **Dynamic Graph Focus**: The significant speedup for incremental updates suggests that parallel implementations are particularly well-suited for dynamic graph applications

Conclusion

Our performance analysis reveals that while parallel implementations currently underperform for static, one-time SSSP computation at this scale, they demonstrate significant advantages for dynamic graphs requiring incremental updates. The substantial speedup factors (3.23× for MPI+OpenMP and 4.39× for MPI) for incremental updates highlight the effectiveness of our parallel approach in localizing computation to affected regions.

For applications involving frequent graph modifications, the parallel implementations offer clear benefits despite their higher initial overhead. Future optimization work should focus on reducing communication frequency and increasing computation batching to improve overall performance while maintaining the existing advantages for incremental scenarios.