Performance Analysis Report: Parallel Single Source Shortest Path (SSSP) Implementation

Authors:
Mesam E. Tamaar Khan (i221304)
Tashfeen Hassan (i220860)
Aamna Saeed (i221179)
Date: May 2025

1. Introduction

This report presents a detailed performance analysis of a parallel implementation of the Single Source Shortest Path (SSSP) algorithm, developed using the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP). The implementation distributes a graph across multiple processes, employs the METIS library for graph partitioning, and leverages Dijkstra's algorithm in a distributed framework. The analysis evaluates the algorithm's scalability, efficiency, and performance characteristics, focusing on its hybrid parallel design, dynamic load balancing, and communication strategies.

2. Implementation Strategy

The implementation, encapsulated in `parallel_algo.cpp`, integrates MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism. The key components of the strategy are as follows:

- **Graph Partitioning:**
    - Large graphs (num_vertices > size * 5) are partitioned using METIS (`METIS_PartGraphKway`).
    - Smaller graphs use a modulo-based distribution.
    - Goal: balanced workloads and reduced communication overhead.
- **Local Computation:**
    - Each process runs Dijkstra's algorithm on its local subgraph.
    - A priority queue and distance array are maintained.
    - OpenMP parallelizes edge relaxation operations.
- **Boundary Vertex Management:**
    - Vertices with cross-partition edges are tracked as boundary vertices.
    - Distance updates for these vertices are exchanged between processes.
- **Inter-Process Communication:**
    - `MPI_Allgatherv` is used to synchronize boundary vertex updates.
    - A timeout mechanism prevents potential deadlocks.

- **Synchronization:**
    - `MPI_Allreduce` and barriers ensure consistency and coordinated termination.
- **Load Balancing:**
    - Repartitioning is triggered when imbalance exceeds 1.5x the mean activity.
- **Termination:**
    - Ends when all distances are finalized and no updates remain.

3. Experimental Configuration

Hardware:

- Four nodes, each with a 16-core Intel Xeon processor (32 threads) and 64 GB RAM

Software:

- MPICH 4.0.2 (MPI)
- METIS 5.1.0 (partitioning)
- GCC 11.2+ (with OpenMP support)

Threads:

- Each MPI process uses 4 OpenMP threads (`omp_set_num_threads(4)`).

Input Datasets:

- **Large Graph:** 100,000 nodes, ~500,000 edges (avg. degree 10)
- **Small Graph:** 10,000 nodes, ~50,000 edges
- **Format:** Edge-list (`graph2.txt`) with lines `u v w`

Process Configurations:

- Tests with 1, 2, 4, 8, and 16 MPI processes

4. Performance Results

4.1 Timing Metrics

Table 1: Performance Metrics for 100,000-Node Graph

MPI Processes	Real Time (s)	User Time (s)	System Time (s)
1	410.234	408.392	1.842
2	198.513	304.589	11.458
4	105.223	188.923	7.210
8	60.114	132.482	5.320
16	36.734	99.301	4.812

4.2 Speedup and Efficiency

Speedup: **S = T1 / Tp**
Efficiency: **E = S / p**

Table 2: Speedup and Efficiency for 100,000-Node Graph

MPI Processes	Real Time (s)	Speedup	Efficiency
1	410.234	1.000	1.000
2	198.513	2.066	1.033

4	105.223	3.898	0.975
8	60.114	6.824	0.853
16	36.734	11.169	0.698

Efficiency > 1.0 at 2 processes likely due to cache effects and OpenMP.
Efficiency drops with more processes due to communication overhead.

4.3 Scalability

Table 3: Performance Comparison (2 MPI Processes)

Nodes	**Real Time (s)**	**User Time (s)**	**System Time (s)**
100,000	198.513	304.589	11.458
10,000	2.531	3.218	0.641

5. Observations

- **Scalability:**
    - The 10,000-node graph runs ~78x faster than 100,000-node.
    - Super-linear speedup from better cache use and fewer communications.

- **Communication Overhead:**
    - Peaks at 11.458s (2 processes), reduces with more processes.
- **OpenMP Effectiveness:**
    - Significantly reduces user time.
- **Load Balancing:**
    - Repartitioning triggered if imbalance > 1.5x average.
    - METIS provides good initial distribution.
- **Deadlock Prevention:**
    - `mpiAllgathervWithTimeout` uses 10s timeout — no timeouts observed.
- **Boundary Efficiency:**
    - Communication limited to boundary vertex updates.
    - Volume increases with graph size and process count.

6. Conclusion

The parallel SSSP algorithm demonstrates effective speedup (up to 11.169x with 16 processes) on large graphs. It balances performance and reliability through MPI, OpenMP, METIS, and dynamic load management. The main bottleneck remains inter-process communication. Future improvements could target reducing communication frequency, overlapping communication with computation, and scaling tests on denser or even larger graphs.

This project forms a solid base for scalable, high-performance graph processing systems.