Comparative Performance Report: Serial vs Parallel Implementation of SSSP Algorithm

1. Introduction

This report evaluates the performance and scalability of three implementations of the Single-Source Shortest Path (SSSP) algorithm:
- Serial (baseline)
- OpenMP (shared-memory parallelism)
- Hybrid MPI + OpenMP (distributed-memory parallelism with intra-node threading)

We analyze each version in terms of algorithmic structure, commands used, implementation complexity, and empirical performance.

2. Problem Overview

The SSSP problem involves computing the shortest path from a source node to all other nodes in a graph with non-negative weights. Our implementations focus on a modified Dijkstra's algorithm, adapted for parallel processing where feasible.

3. Serial Implementation

- Language: C++
- Graph Representation: Adjacency matrix
- Algorithm: Basic Dijkstra with linear search for minimum

Code Highlights:

```
int minDistance(int dist[], bool visited[], int V);
void dijkstra(int graph[V][V], int src, int V);
```

Build & Run:

```
g++ serialImplementation.cpp -o serial_sssp
./serial_sssp
```

Performance:

Execution Time: ~0.29 seconds
Threads/Processes: 1
Time complexity: $O(V^2)$

4. OpenMP Implementation

Parallelism: Shared memory using OpenMP
Optimized Sections:
- Minimum vertex search via reduction
- Distance updates via parallel loops

Code Highlights:

```
#pragma omp parallel for reduction(min: min, min_index)
for (int v = 0; v < V; v++) {
   if (!visited[v] && dist[v] <= min) {
      min = dist[v], min_index = v;
   }
}
```

Build & Run:

```
g++ -fopenmp OpenMpImplementation.cpp -o omp_sssp
OMP_NUM_THREADS=16 ./omp_sssp
```

Performance Metrics:

Total Execution Time: 0.20974 seconds
Initialization Time: 0.05312 seconds
I/O Time: 0.14895 seconds
Affected Vertices Identification Time: 1.88e-6 seconds
Update Time: 0.00024 seconds
  - Synchronous: 0.0002446 seconds
  - Asynchronous: 0.0002442 seconds
Number of Threads: 16

Update Method: Both
Speedup Over Serial: 1.386x

5. Hybrid MPI + OpenMP Implementation

Distributed + Shared Parallelism: Each MPI process owns a subgraph and uses OpenMP internally.
Architecture Used: 8 processes per node × 2 nodes = 16 total (Master + Slaves)

Key Techniques:

MPI_Bcast, MPI_Allreduce, #pragma omp parallel for

Build & Run:

mpic++ -fopenmp mpi_openmp_sssp.cpp -o hybrid_sssp
mpirun -np 16 ./hybrid_sssp

Performance :

Total Execution Time: ~0.078 seconds
Speedup Over Serial: ~3.7x
Speedup Over OpenMP: ~2.6x
Processes: 16 (8 per node)
Threads per Process: 1–2
Communication Overhead: Present but minimal
Update Method: Master-Slave + Both

6. Comparative Summary

Comparative Summary Table:

Metric	Serial	OpenMP (16 threads)	MPI + OpenMP (16 procs)
Total Time (approx)	0.29 s	0.2097 s	0.078 s
Speedup vs Serial	1x	1.386x	3.7x
Threads/Processes	1	16 threads	16 processes
Scalability	None	Moderate	High
Complexity	Low	Medium	High
Best Use Case	Small Graphs	Medium Graphs	Large Graphs

7. Conclusion

The report demonstrates that parallelization offers tangible benefits for graph algorithms, especially for larger datasets:

- Serial implementations are straightforward but inefficient for large graphs.
- OpenMP improves performance modestly, particularly in compute-bound phases.
- MPI + OpenMP provides superior speedup and is most suitable for large-scale graph processing on clusters.

Using 16 MPI processes (8 per node) with intra-process OpenMP threads, we achieved a speedup of ~3.7x over the serial baseline, and ~2.6x over the standalone OpenMP solution.