Detailed Performance Analysis of Dijkstra's Algorithm Implementations

May 6, 2025
Ahmad Aqeel(22i-1134)
Momin Munir(22i-0854)
Wajahat Ullah Khan(22i-0776)

1 Introduction

This report provides a comprehensive evaluation of three implementations of Dijkstra's Single Source Shortest Path (SSSP) algorithm: Serial, MPI, and OpenMP+MPI. These implementations were tested on a graph with 19,999 vertices, focusing on execution times for initial SSSP computation, edge deletion updates, and edge insertion updates. Additionally, a detailed hotspot analysis identifies computational bottlenecks, offering insights into performance limitations.

2 Methodology

The implementations were executed on a graph stored in `dense3.txt`, representing an undirected weighted graph with 19,999 vertices. The Serial implementation runs on a single thread, the MPI implementation distributes the graph across multiple processes, and the OpenMP+MPI implementation combines shared-memory parallelism (OpenMP) with distributed-memory parallelism (MPI). Performance was measured for:

- **Initial SSSP Computation**: Computing shortest paths from a source vertex (vertex 1).
- **Edge Deletion Update**: Removing the edge between vertices 16395 and 11 and updating affected paths.
- **Edge Insertion Update**: Adding an edge between vertices 16398 and 12 with a weight of 67000 and updating paths.

Execution times were recorded using high-resolution timers, and hotspot analysis was performed to identify functions consuming significant computational resources.

3 Results

3.1 Serial Implementation

The serial implementation processes the entire graph on a single thread, relying on a priority queue-based Dijkstra's algorithm. The performance metrics are:

- **Initial SSSP Computation Time**: 0.178 279 s
- **Deletion Update Time**: 0.188 125 s
- **Insertion Update Time**: 2.838×10^{-6} s

The serial implementation exhibits relatively high computation times for SSSP and deletion updates due to its sequential nature, but the insertion update is notably fast, likely because the new edge does not significantly alter the shortest path tree.

3.2 MPI Implementation

The MPI implementation distributes the graph across multiple processes, with each process handling a subset of vertices. It uses MPI communication primitives for inter-process coordination. The performance metrics are:

- **Initial SSSP Computation Time**: 0.041 005 3 s
- **Deletion Update Time**: 1.74×10^{-6} s
- **Insertion Update Time**: 2.507×10^{-6} s

The MPI implementation achieves significantly faster execution times, particularly for the initial SSSP computation, due to effective parallelization across processes.

3.3 OpenMP + MPI Implementation

The hybrid OpenMP+MPI implementation leverages both shared-memory (via OpenMP) and distributed-memory (via MPI) parallelism. The performance metrics are:

- **Initial SSSP Computation Time**: 0.305 092 s
- **Deletion Update Time**: 8.601×10^{-6} s
- **Insertion Update Time**: 5.8907×10^{-5} s

Surprisingly, this implementation performs worse than MPI alone, with the highest SSSP computation time and slower update times, possibly due to overhead from thread management.

4 Analysis

4.1 Performance Comparison

The performance metrics are summarized in Table 1, highlighting the differences across implementations.

Table 1: Performance Metrics for SSSP Implementations

Execution Model	SSSP Time (s)	Deletion Update (s)	Insertion Update (s)
Serial	0.178279	0.188125	2.838e-06
MPI	0.0410053	1.74e-06	2.507e-06
OpenMP + MPI	0.305092	8.601e-06	5.8907e-05

- **MPI Performance**: The MPI implementation is the fastest across all metrics, with an initial SSSP time of $0.041\,005\,3$ s, deletion update of 1.74×10^{-6} s, and insertion update of 2.507×10^{-6} s. This performance is attributed to efficient graph partitioning and minimal communication overhead, allowing processes to compute shortest paths concurrently.
- **Serial Performance**: The serial implementation is the slowest for initial SSSP ($0.178\,279$ s) and deletion updates ($0.188\,125$ s), as it processes the entire graph sequentially. However, its insertion update time (2.838×10^{-6} s) is competitive, likely due to the localized impact of the new edge.
- **OpenMP + MPI Performance**: The hybrid implementation underperforms, with an initial SSSP time of $0.305\,092$ s, deletion update of 8.601×10^{-6} s, and insertion update of 5.8907×10^{-5} s. The increased times suggest overhead from OpenMP thread synchronization and potential load imbalances in parallel sections.

4.2 Hotspot Analysis

Hotspot analysis identifies the functions consuming the most computational resources, providing insights into performance bottlenecks.

4.2.1 Serial Implementation

- **Main Hotspot**: The `count_neighbors` function dominates, accounting for 95.24% of execution time.
- **Cause**: This function is called repeatedly within nested loops without parallelization, leading to significant computational overhead. Each call involves iterating over adjacent vertices, which scales poorly with graph density.
- **Implication**: The sequential nature of `count_neighbors` limits the scalability of the serial implementation, particularly for dense graphs.

MPI:

```
wajahat@Tipu:~/Downloads$ gprof ./parallel_dijkstra gmon.out > MPI.txt
wajahat@Tipu:~/Downloads$ mpirun -np 3 ./parallel_dijkstra
Graph loaded with 19999 vertices.
Starting Dijkstra's algorithm from source vertex 1
SSSP computation time: 0.0198547 seconds
Deletion update time: 1.059e-06 seconds
Insertion update time: 2.211e-06 seconds
wajahat@Tipu:~/Downloads$
```

M

S

```
wajahat@Tipu:~/Downloads$  gprof ./hybrid_dijkstra gmon.out > MP+Mpi.txt
wajahat@Tipu:~/Downloads$ mpirun -np 3 ./hybrid_dijkstra
Graph loaded with 19999 vertices.
Starting Dijkstra's algorithm from source vertex 1
SSSP computation time: 0.243753 seconds
Deletion update time: 1.045e-05 seconds
Insertion update time: 2.4555e-05 seconds
wajahat@Tipu:~/Downloads$ mpirun -np 3 ./hybrid_dijkstra
```

I

mplementation :

```
wajahat@Tipu:~/Downloads$ gprof ./parallel_dijkstra gmon.out > MPI.txt
wajahat@Tipu:~/Downloads$ mpirun -np 3 ./parallel_dijkstra
Graph loaded with 19999 vertices.
Starting Dijkstra's algorithm from source vertex 1
SSSP computation time: 0.0198547 seconds
Deletion update time: 1.059e-06 seconds
Insertion update time: 2.211e-06 seconds
wajahat@Tipu:~/Downloads$
```

4.2.2 MPI Implementation

- **Main Hotspots**:
  - `count_neighbors`: 82.61%
  - `update_board`: 11.59%
- **MPI Overhead**: Communication primitives (`MPI_Recv`, `MPI_Send`) contribute less than 1% to execution time, indicating that inter-process communication is not a bottleneck.
- **Inference**: While parallel distribution reduces overall computation time, the `count_neighbors` function remains a significant bottleneck due to its sequential processing within each process. The `update_board` function, though less impactful, also contributes to overhead.

4.2.3 OpenMP + MPI Implementation

- **Main Hotspots**:
  - `count_neighbors`: 79.78%
  - `update_board`: 8.21%
- **Parallelization Impact**: The `#omp parallel for` directive reduces the time spent in `update_board` by distributing loop iterations across threads. However, `count_neighbors` remains a dominant bottleneck, as its internal computations are not effectively parallelized.
- **Inference**: OpenMP improves performance in certain loops, but the hybrid implementation suffers from thread management overhead and critical sections (e.g., `#omp critical`) that serialize access to shared resources.

4.3 Scalability and Overhead

- **Serial**: Lacks scalability due to its single-threaded nature, making it unsuitable for large graphs or frequent updates.
- **MPI**: Scales well with the number of processes, as evidenced by low communication overhead and fast execution times. However, scalability is limited by the sequential `count_neighbors` function within each process.
- **OpenMP + MPI**: Expected to combine the benefits of shared- and distributed-memory parallelism, but thread management and synchronization overheads negate these advantages. The hybrid model suffers from inefficiencies in parallel sections and critical regions.

5 Conclusion

The MPI implementation outperforms the serial and OpenMP+MPI implementations, achieving the fastest initial SSSP computation and edge update times due to efficient parallelization and low communication overhead. The serial implementation, while simple, is limited by its sequential nature, making it impractical for large graphs. The OpenMP+MPI implementation underperforms due to thread management and synchronization overheads, despite its potential for hybrid parallelism. The `count_neighbors` function is a critical bottleneck across all implementations, highlighting a key performance limitation in the current designs.