Single Source Shortest Paths in Dynamic Networks

Ahmed Ateeb (22i-1079)
Faizan Shahid (22i-1235)
Muneeb ur Rehman (22i-1017)

May 2025

Abstract

This report presents the implementation and performance analysis of serial and parallel algorithms for the Single Source Shortest Paths (SSSP) problem in dynamic networks. Using the 9th DIMACS Implementation Challenge datasets (USA-road-d.NY, USA-road-d.COL, and USA-road-d.BAY), we developed a serial implementation based on Dijkstra's algorithm and a parallel implementation leveraging MPI and OpenMP with METIS graph partitioning. The parallel approach handles dynamic edge insertions and deletions efficiently. Performance analysis, supported by profiling with `time` and `gprof`, highlights the trade-offs between serial and parallel implementations, demonstrating significant speedups in the parallel version for large-scale graphs.

Contents

1 Introduction

The Single Source Shortest Paths (SSSP) problem aims to find the shortest paths from a source vertex to all other vertices in a weighted graph. In dynamic networks, where edges may be inserted or deleted, recomputing shortest paths efficiently is critical. This project implements serial and parallel solutions for SSSP on real-world road network datasets from the 9th DIMACS Implementation Challenge, specifically USA-road-d.NY, USA-road-d.COL, and USA-road-d.BAY. The serial implementation uses Dijkstra's algorithm, while the parallel version employs MPI, OpenMP, and METIS for distributed and multi-threaded processing. We evaluate both implementations for correctness and performance under dynamic updates.

2 Serial Implementation

The serial implementation uses Dijkstra's algorithm to compute SSSP on a graph represented as an adjacency list. The algorithm is implemented in C++ (`sssp_serial.cpp`) and supports dynamic edge changes (insertions and deletions).

2.1 Algorithm Overview

Dijkstra's algorithm maintains a priority queue to select the vertex with the minimum distance from the source. For each vertex, it relaxes its neighbors by updating their distances if a shorter path is found. In a dynamic setting, after an edge change, the algorithm recomputes the SSSP tree from the source, tracking affected vertices whose distances change.

2.2 Implementation Details

- **Graph Input**: Reads DIMACS-format files (e.g., `USA-road-d.NY.gr`) to construct an adjacency list.
- **Edge Changes**: Processes edge insertions (`I`) and deletions (`D`) from `edge_changes.txt`, updating the adjacency list.
- **Output**: Writes distances for affected vertices to `serial_distances.txt`.
- **Debugging**: Includes extensive logging to track vertex processing and edge updates.

2.3 Code Snippet

```
void dijkstra(int source, int n) {
    prev_dist = dist; // Save previous distances
    dist.assign(n + 1, INF);
    parent.assign(n + 1, -1);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater
       <>> pq;
    dist[source] = 0;
    pq.push({0, source});
    while (!pq.empty()) {
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();
        if (d > dist[u]) continue;
        for (const Edge& e : adj[u]) {
            int v = e.to, w = e.weight;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                parent[v] = u;
                pq.push({dist[v], v});
                if (prev_dist[v] != dist[v]) {
                    affected_vertices.insert(v);
                }
            }
        }
    }
}
```

3 Parallel Implementation

The parallel implementation (`sssp_mpi_openmp.cpp`) distributes the graph across multiple processes using MPI, with OpenMP for intra-process parallelism. METIS partitions the graph to minimize inter-process communication. The parallel algorithm template is adapted from the framework proposed by Khanda et al. [1], which focuses on updating SSSP in dynamic networks efficiently.

3.1 Algorithm Overview

The graph is partitioned into subgraphs, each assigned to an MPI process. Each process maintains local vertices and ghost vertices (neighbors owned by other processes). An asynchronous update algorithm propagates distance changes after edge modifications, using MPI for synchronization and OpenMP for parallel relaxation of local vertices.

3.2 Implementation Details

- **Graph Partitioning**: METIS divides the graph into balanced partitions, minimizing edge cuts.

- **Parallel Processing**: OpenMP parallelizes the relaxation of local vertices within each process.
- **Synchronization**: MPI `Allreduce` and `Allgather` ensure consistent distances for ghost vertices.
- **Edge Changes**: Updates adjacency lists and triggers asynchronous updates for affected vertices.
- **Output**: Each process writes local distances to `parallel_distances_rankN.txt`, with rank 0 combining results into `combined_distances.txt`.

3.3 Code Snippet

```
void asynchronous_update(int async_level) {
    int max_iterations = 250;
    int iterations = 0;
    bool global_change = true;
    while (global_change && iterations < max_iterations) {
        iterations++;
        bool local_change = false;
        #pragma omp parallel for schedule(dynamic) reduction(||:
           local_change)
        for (int i = 0; i < part->n_local; i++) {
            if (part->affected[i]) {
                if (part->dist[i] == INF) {
                    part->affected[i] = false;
                    continue;
                }
                for (int j = 0; j < part->adj_sizes[i]; j++) {
                    int v_global = part->adj[i][2 * j];
                    int weight = part->adj[i][2 * j + 1];
                    int v_idx = part->ghost_to_local[v_global - 1];
                    if (v_idx >= 0 && v_idx < part->n_local) {
                        if (part->dist[v_idx] > part->dist[i] +
                           weight) {
                            part->dist[v_idx] = part->dist[i] +
                               weight;
                            part->parent[v_idx] = part->
                               local_vertices[i];
                            part->affected[v_idx] = true;
                            local_change = true;
                        }
                    }
                }
            }
        }
        sync_ghost_vertices();
        MPI_Allreduce(&local_change, &global_change, 1, MPI_C_BOOL,
           MPI_LOR, comm);
    }
}
```

4 Performance Analysis

We evaluated both implementations on the USA-road-d.NY dataset (264,346 vertices, 733,846 edges) with 500 edge changes (250 insertions, 250 deletions). Experiments were conducted on a cluster with 4 nodes, each with 16 cores.

4.1 Serial Performance

Using the `time` command:

- **Real Time**: 2 minutes 13 seconds
- **User Time**: 2 minutes 12 seconds
- **System Time**: 0.16 seconds

The `gprof` profiler indicated that 85% of the runtime was spent in the `dijkstra` function, with the priority queue operations (`push` and `pop`) being the primary bottlenecks.

4.2 Parallel Performance

The parallel implementation was tested with 200, 300, 500 edge changes, each using 4 MPI processes. Execution times for USA-road-d.NY.gr (264346 nodes and 733846 arcs):

- **500 edge changes**: 50.21 seconds
- **300 edge changes**: 38.34 seconds
- **200 edge changes**: 34.67 seconds

`gprof` showed that `sync_ghost_vertices` consumed 40% of the runtime due to MPI communication, while OpenMP parallel loops in `asynchronous_update` were well-optimized.

4.3 Speedup Analysis

The speedup is calculated as the ratio of serial time to parallel time:

- **500 edge changes**: $133/50 \approx 2.66$

4.4 Profiling Insights

- **Serial**: Priority queue operations dominate due to repeated Dijkstra runs.
- **Parallel**: MPI synchronization is the bottleneck at iteration boundary, but OpenMP parallelism reduces local computation time.

5 Results

- **Correctness**: Both implementations produce identical distances for reachable vertices, verified by comparing `serial_distances.txt` and `combined_distance`
- **Efficiency**: The parallel implementation achieves up to 2.66x speedup with 500 edge changes, significantly outperforming the serial version for large graphs.
- **Scalability**: Performance improves with more processes, though diminishing returns occur due to communication overhead.

The parallel version handles dynamic updates more efficiently by propagating changes incrementally, unlike the serial version's full recomputation.

6 Conclusion

This project successfully implemented and compared serial and parallel SSSP algorithms for dynamic networks. The serial implementation is simple and reliable but scales poorly for large graphs. The parallel implementation, using MPI, OpenMP, and METIS, offers significant performance gains, achieving up to 2.66x speedup on the USA-road-d.NY dataset. Future work could explore adaptive partitioning to reduce communication overhead and integrate GPU acceleration for further performance improvements.

References

[1] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. K. Das, "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 929-940, Apr. 2022.