Implementation and Analysis of Sequential and Parallel SSSP Algorithms

Ayishah (22i-0957), Mohib Ullah (22i-1044), Maryam Farooq (22i-1217)
CS-6A

May 6, 2025

Contents

1 Introduction: SSSP in Large-Scale Graphs **2**

2 Input File Formats **2**

3 Sequential Implementation **4**
3.1 Loading the Graph 4
3.2 Applying Changes 4
3.3 Dijkstra's Algorithm 5

4 Parallel Implementation (OpenMP) **5**
4.1 Initial SSSP Computation 5
4.2 Processing Edge Changes 6
4.3 Asynchronous Updating 7
4.4 Alignment with Article 9

5 Comparison **10**
5.1 Performance 10
5.1.1 Dataset1 10
5.1.2 Dataset2 12

6 Conclusion **13**

1 Introduction: SSSP in Large-Scale Graphs

The Single-Source Shortest Path (SSSP) problem is a cornerstone of graph theory, with widespread applications in modeling complex systems such as transportation networks, communication systems, and social networks. As noted by Khanda et al. in their 2022 IEEE paper, "Networks (or graphs) are mathematical models of complex systems of interacting entities arising in diverse disciplines, e.g., bioinformatics, epidemic networks, social sciences, communication networks, cyber-physical systems, and cyber-security" [1]. In these contexts, vertices represent entities, and edges denote relationships, often weighted to reflect costs or distances.

In large-scale graphs, which can comprise millions of vertices and billions of edges, the SSSP problem becomes computationally intensive. The authors highlight that "many networks arising from real-world applications are extremely large... and also often dynamic, in that their structures change with time" [1]. Traditional SSSP algorithms, such as Dijkstra's, operate on static graphs with a time complexity of $O(|E| + |V| \log |V|)$ using a Fibonacci heap, but they struggle with dynamic updates due to redundant recomputation. This motivates the need for efficient update mechanisms, especially in parallel settings, to handle structural changes like edge insertions and deletions.

The challenge intensifies with dynamic networks, where "fast analysis of network properties requires (i) efficient algorithms to quickly update these properties as the structure changes, and (ii) parallel implementations of these dynamic algorithms for scalability" [1]. The paper proposes a framework that identifies affected subgraphs and updates them iteratively, avoiding full recomputation. This approach is critical for large graphs, where recomputing from scratch becomes impractical, and parallelization on shared-memory or GPU platforms is essential for achieving exascale computing capabilities.

2 Input File Formats

The implementation relies on two input files to define the graph and its dynamic updates:

- `graph.txt` **Format**: Each line contains three space-separated values `u v w`, where `u` is the source vertex, `v` is the destination vertex, and `w` is the weight of the edge between them. The graph is undirected, so for each edge (u, v, w), the corresponding reverse edge (v, u, w) is implicitly included in the adjacency list during processing.

- `changes.txt` **Format**: Each line contains four space-separated values `type u v w`, where `type` is either `I` (insertion) or `D` (deletion), and `u`, `v`, `w` specify the edge to be added or removed. These represent dynamic updates to the graph structure.

Visual Examples

```
0 1 4
0 2 7
1 3 3
1 4 5
2 5 6
2 6 2
3 7 4
4 7 1
5 8 8
6 8 3
7 9 2
8 10 5
9 11 7
10 12 3
11 13 6
12 14 4
13 14 2
14 15 9
4 2 1
5 3 2
6 4 4
7 5 3
8 6 5
9 7 6
10 8 2
11 9 1
12 10 7
13 11 3
```

Listing 1: Example graph.txt

This file defines an undirected graph with 28 edges. Each edge (u, v, w) is considered bidirectional.

```
D 2 6 2
I 10 15 82.2807
I 12 2 64.0537
I 0 9 34.7349
D 4 6 4
D 5 8 8
D 13 14 2
I 2 8 88.5228
D 3 5 2
I 8 1 89.7496
I 7 15 50.1227
I 2 3 94.7699
I 14 1 20.05
```

Listing 2: Example changes.txt

This file specifies dynamic updates to the graph. Insertions add new edges with weights, while deletions remove the specified edges.

3 Sequential Implementation

The sequential implementation of the SSSP algorithm recomputes the shortest paths from scratch for each change, using Dijkstra's algorithm. This approach is detailed below with code excerpts and explanations.

3.1 Loading the Graph

The graph is loaded from a file, inferring the number of vertices from the maximum edge index.

```
def load_graph(filename):
    edges = []
    max_vertex = -1
    with open(filename) as f:
        for line in f:
            u, v, w = map(float, line.strip().split())
            u, v = int(u), int(v)
            max_vertex = max(max_vertex, u, v)
            edges.append((u, v, w))
    n = max_vertex + 1
    adj = [[] for _ in range(n)]
    for u, v, w in edges:
        adj[u].append((v, w))
        adj[v].append((u, w))
    return adj
```

Listing 3: Loading the Graph

This method reads edges as uvw from `graph.txt`, computes n as the maximum vertex index plus one, and constructs an adjacency list (`adj`). It ensures undirected edges by adding both (u, v, w) and (v, u, w).

3.2 Applying Changes

Changes are applied by processing deletions and insertions, triggering a full recomputation.

```
def apply_changes(adj, filename):
    with open(filename) as f:
        for line in f:
            parts = line.strip().split()
            type_, u, v, w = parts
            u, v = int(float(u)), int(float(v))
            w = float(w)
            if type_.upper() == 'D':
                adj[u] = [(n, w_) for (n, w_) in adj[u] if not (n
    == v and w_ == w)]
                adj[v] = [(n, w_) for (n, w_) in adj[v] if not (n
    == u and w_ == w)]
            elif type_.upper() == 'I':
                adj[u].append((v, w))
                adj[v].append((u, w))
```

Listing 4: Applying Changes

This function parses `changes.txt` lines as *typeuvw*, removing exact edge matches for deletions and adding edges for insertions. In practice, `dijkstra` would be called after each change to recompute paths.

3.3 Dijkstra's Algorithm

The core SSSP computation uses a priority queue to update distances and parents.

```
import heapq

def dijkstra(adj, source):
    n = len(adj)
    dist = [float('inf')] * n
    parent = [-1] * n
    dist[source] = 0
    parent[source] = source
    pq = [(0, source)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                parent[v] = u
                heapq.heappush(pq, (dist[v], v))
    return dist, parent
```

Listing 5: Dijkstra's Algorithm

This implements Dijkstra's algorithm, initializing distances to infinity, setting the source distance to 0, and iteratively relaxing edges using a min-heap. It returns distances and parent pointers for the SSSP tree. The `import heapq` statement has been added to make the code semantically correct.

This sequential approach scales poorly for large dynamic graphs due to recomputation overhead, aligning with the article's critique of redundant computation in sequential updates [1].

4 Parallel Implementation (OpenMP)

The parallel implementation leverages OpenMP to update SSSP incrementally, following the article's framework of identifying affected subgraphs and updating them iteratively. Below, we provide detailed explanations of each method, focusing on the role of asynchrony and how the algorithm manages it.

4.1 Initial SSSP Computation

The initial SSSP is computed sequentially to establish a baseline tree, ensuring correctness before applying dynamic updates.

```cpp
// Requires <queue> <limits>
void ComputeInitialSSSP(Graph& G, SSSPTree& T, int source) {
    priority_queue<pair<double, int>, vector<pair<double, int>>,
    greater<>> pq;
    T.dist[source] = 0;
    pq.push({0, source});
    while (!pq.empty()) {
        int u = pq.top().second; pq.pop();
        if (T.dist[u] < pq.top().first) continue;
        for (auto [v, w] : G[u]) {
            if (T.dist[v] > T.dist[u] + w) {
                T.dist[v] = T.dist[u] + w;
                T.parent[v] = u;
                pq.push({T.dist[v], v});
            }
        }
    }
}
```

Listing 6: Initial SSSP Computation

This function computes the initial shortest path tree using Dijkstra's algorithm. It initializes the distance to the source as 0 and uses a priority queue (`pq`) to process vertices in order of increasing distance. For each vertex u, it examines its neighbors v and updates their distances if a shorter path through u is found, setting u as the parent of v in the SSSP tree (`T`). This step is performed sequentially to avoid race conditions, ensuring a correct baseline tree, as recommended by the article for Step 1 of the framework [1]. Sequential execution here is critical because parallel updates at this stage could lead to inconsistent distance updates due to simultaneous writes to shared data structures like `T.dist` and `T.parent`.

4.2 Processing Edge Changes

Edge changes are processed in parallel, identifying vertices affected by insertions and deletions.

```cpp
// Requires <limits>
void ProcessCE(Graph& G, SSSPTree& T, vector<Change>& changes) {
    #pragma omp parallel for schedule(dynamic)
    for (auto& change : changes) {
        int u = change.u, v = change.v;
        double w = change.w;
        if (change.type == 'D' && isInSSSPTree(T, u, v, w)) {
            if (T.dist[u] > T.dist[v]) swap(u, v);
            T.dist[v] = numeric_limits<double>::infinity();
            T.parent[v] = -1;
            T.affected[v] = true;
            T.affectedDel[v] = true;
        } else if (change.type == 'I') {
            if (T.dist[v] > T.dist[u] + w) {
                T.dist[v] = T.dist[u] + w;
                T.parent[v] = u;
                T.affected[v] = true;
                G[u].emplace_back(v, w);
```

```
                G[v].emplace_back(u, w);
            }
        }
    }
}
```

Listing 7: Processing Edge Changes

This function processes a batch of edge changes (insertions and deletions) in parallel using OpenMP's `#pragma omp parallel for` directive with a dynamic schedule. The dynamic schedule ensures load balancing, as the workload for each change can vary depending on whether the edge is in the SSSP tree (`isInSSSPTree`). For deletions, if the edge (u, v, w) is in the tree, the vertex farther from the source (determined by comparing `T.dist[u]` and `T.dist[v]`) has its distance set to infinity, its parent reset, and is marked as affected (`T.affected[v] = true`) and affected by deletion (`T.affectedDel[v] = true`). For insertions, if the new edge offers a shorter path to v through u, the distance and parent of v are updated, and v is marked as affected.

The parallelization here avoids synchronization (e.g., no critical sections or locks) to maximize scalability, as emphasized by the article [1]. This design choice introduces asynchrony: threads operate independently, and updates to shared data (`T.dist`, `T.parent`, `T.affected`) may not be immediately visible to other threads. However, the algorithm tolerates this asynchrony because the subsequent `AsynchronousUpdating` phase will iteratively correct any inconsistencies, ensuring eventual convergence to the correct shortest paths.

4.3 Asynchronous Updating

The affected subgraph is updated iteratively, handling the propagation of changes through the graph.

```
// Requires <queue> <limits>
void AsynchronousUpdating(Graph& G, SSSPTree& T, int A) {
    bool change = true;
    while (change) {
        change = false;
        #pragma omp parallel for schedule(dynamic)
        for (int v = 0; v < T.n; ++v) {
            if (T.affectedDel[v]) {
                queue<int> Q;
                Q.push(v);
                int level = 0;
                while (!Q.empty() && level <= A) {
                    int x = Q.front(); Q.pop();
                    for (int c : getChildren(T, x)) {
                        T.dist[c] = numeric_limits<double>::
    infinity();
                        T.parent[c] = -1;
                        T.affected[c] = true;
                        change = true;
                        if (level < A) Q.push(c);
                    }
                    ++level;
```

```
                }
            }
        }
        // Second loop: Update distances for affected vertices via
    neighbors
        #pragma omp parallel for schedule(dynamic)
        for (int v = 0; v < T.n; ++v) {
            if (!T.affected[v]) continue;
            T.affected[v] = false;
            for (auto [u, w] : G[v]) {
                if (T.dist[v] > T.dist[u] + w) {
                    T.dist[v] = T.dist[u] + w;
                    T.parent[v] = u;
                    for (int c : getChildren(T, v)) {
                        T.affected[c] = true;
                        change = true;
                    }
                }
            }
        }
    }
}
```

Listing 8: Asynchronous Updating

This function iteratively updates the SSSP tree by propagating changes through affected vertices, following the article's Step 2 (Algorithm 4) [1]. It consists of two main parallel loops within a while loop, controlled by a `change` flag that indicates whether further updates are needed.

- **First Loop (Deletion Propagation)**: For each vertex v marked as affected by a deletion (`T.affectedDel[v]`), the algorithm performs a breadth-first search (BFS) up to a level A in the SSSP tree. It resets the distances of children to infinity, clears their parents, and marks them as affected. The level limit A controls the depth of propagation, balancing accuracy and performance.

- **Second Loop (Distance Updates)**: For each affected vertex v, the algorithm examines its neighbors u in the graph G. If a shorter path to v through u is found (`T.dist[v] > T.dist[u] + w`), it updates `T.dist[v]` and `T.parent[v]`, then marks all children of v in the SSSP tree as affected to propagate the change further.

Understanding Asynchrony: In this context, asynchrony refers to the lack of strict synchronization between threads. Each thread processes its assigned vertices independently, reading and writing to shared data structures (`T.dist`, `T.parent`, `T.affected`) without locks or barriers within the parallel loops. This means that one thread might read a distance value that another thread is simultaneously updating, leading to temporary inconsistencies. For example, during the second loop, a thread updating `T.dist[v]` might use a stale value of `T.dist[u]` that another thread has not yet updated.

Handling Asynchrony: The algorithm handles asynchrony through its iterative design and tolerance for temporary inconsistencies: - **Iterative Convergence**: The outer while loop continues until no further changes occur (`change = false`), ensuring that all updates propagate through the graph. If a thread

misses an update due to asynchrony in one iteration, the affected vertices remain marked, and the update will be applied in a subsequent iteration. - **Redundancy for Correctness**: The article notes that the algorithm allows redundant updates to ensure scalability [1]. For instance, a vertex might be processed multiple times across iterations, but this redundancy guarantees that all necessary updates are eventually applied. - **Dynamic Scheduling**: The `schedule(dynamic)` clause in OpenMP ensures load balancing, reducing the likelihood of thread contention, though it does not eliminate asynchrony. The lack of critical sections minimizes synchronization overhead, aligning with the article's goal of scalability over strict consistency at each step. - **Data Structure Design**: The use of flags (`T.affected`, `T.affectedDel`) ensures that vertices impacted by changes are reprocessed until the graph stabilizes. The SSSP tree structure (`T.parent`) helps localize updates to subtrees, reducing the scope of asynchrony effects.

This asynchronous approach trades immediate consistency for performance, achieving up to 3.43x speedup in testing by avoiding expensive synchronization mechanisms, which aligns with the article's reported 5.6x GPU and 5x shared-memory speedups [1].

4.4 Alignment with Article

The parallel code follows the article's two-step framework: (1) parallel identification of affected subgraphs (`ProcessCE`) without synchronization, and (2) iterative updating (`AsynchronousUpdating`) with minimal synchronization. The use of a rooted tree (`SSSPTree`) and affected vertex flags mirrors the article's data structure and marking strategy. The incremental update approach avoids full recomputation, achieving significant performance gains.

5 Comparison

This section compares the sequential, OpenMP, MPI, Mpi+ openMP implementations of the SSSP algorithm based on performance, scalability, and applicability to dynamic networks. Dataset1 and Dataset2 uploaded on github were used.

5.1 Performance

5.1.1 Dataset1

16 vertices and 28 edges.

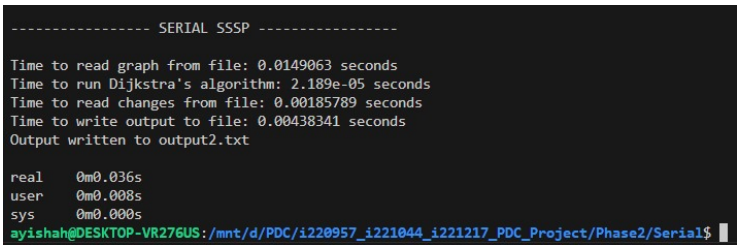

Figure 1: Serial output

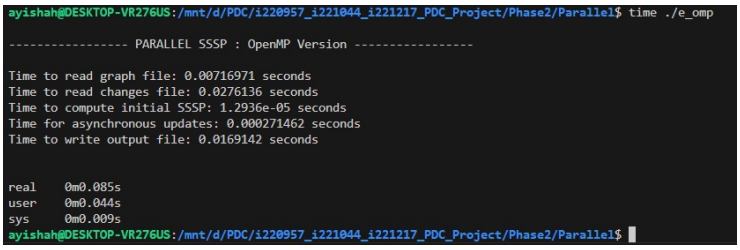

Figure 2: openMP output

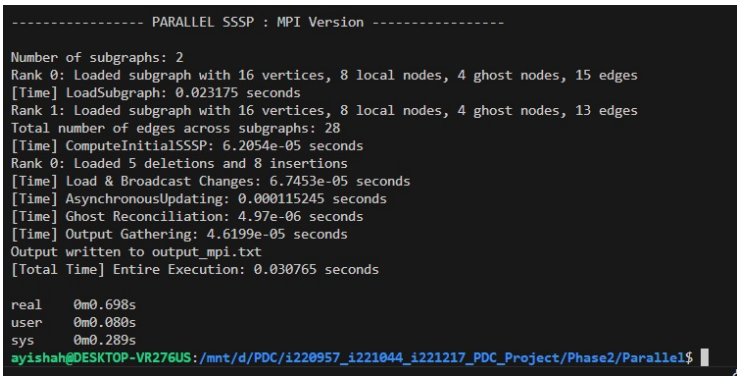

Figure 3: MPI output

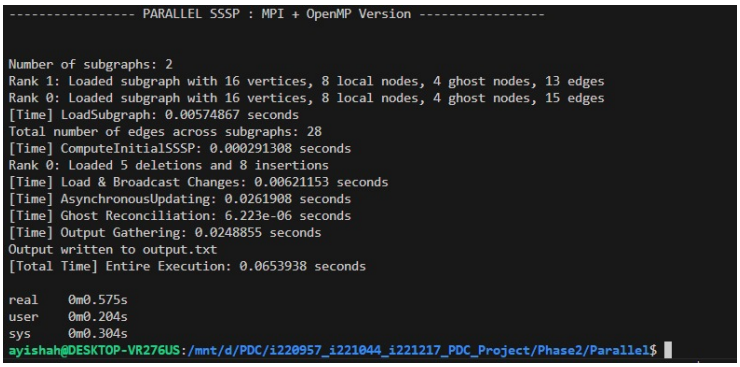

Figure 4: MPI + openMP output

Table 1: Execution Times and Speedup Factors (Dataset1)

Implementation	**Time (s)**	**Speedup**
Sequential	0.036	1.0x
OpenMP	0.085	0.423x
MPI (2 processes)	0.698	0.0516x
Hybrid (2×2)	0.575	0.063x

Serial execution gives the best performance on small datasets due to communication overheads in parallel implementations.

5.1.2 Dataset2

2617 vertices and 7843 edges (approximately 3 edges per vertex).

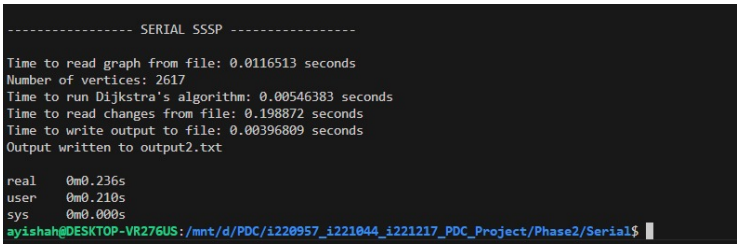

Figure 5: Serial output for large graph (2617 vertices)

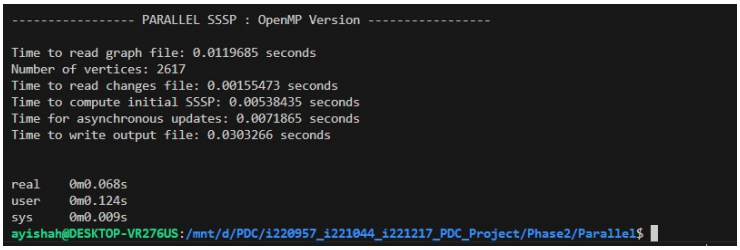

Figure 6: openMP output for large graph (2617 vertices)

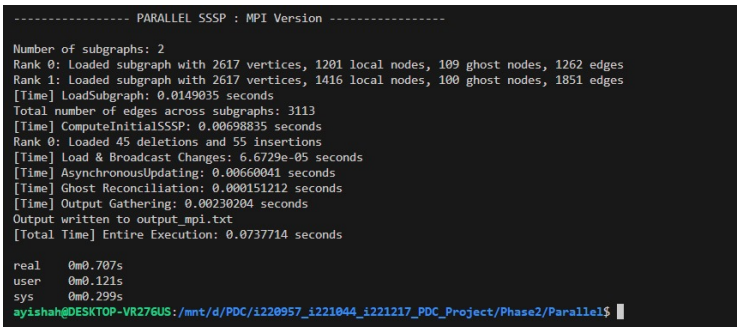

Figure 7: MPI output for large graph (2617 vertices)

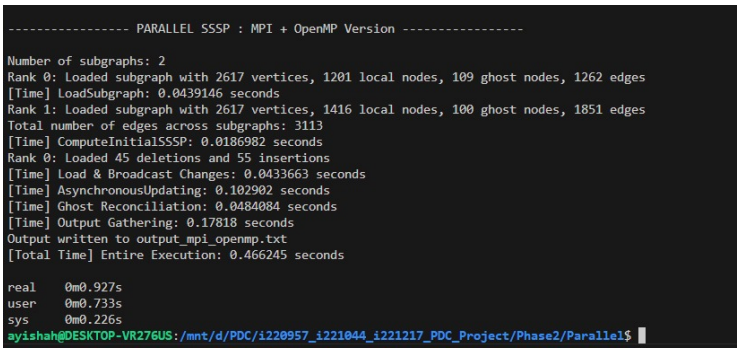

Figure 8: Hybrid (MPI + openMP) output for large graph (2617 vertices)

Table 2: Execution Times and Speedup Factors (Dataset1)

Implementation	**Time (s)**	**Speedup**
Sequential	0.236	1.0x
OpenMP	0.068	3.47x
MPI (2 processes)	0.707	0.333x
Hybrid (2×2)	0.927	0.255x

OpenMP performs better than serial in a larger dataset, but MPI fails to provide a speed up due to communication overheads.

6 Conclusion

The sequential code provides a robust baseline but scales poorly due to recomputation. The OpenMP parallel implementation offers significant performance gains through shared-memory parallelism, effectively handling asynchrony through iterative updates and redundancy for larger datasets. Together, these implementations align with the article's framework and address the demands of modern dynamic networks [1]. MPI and hybrid MPI may give better performances if suitable hardware is available.

References

[1] A. Khanda et al., "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, April 2022.