Parallel Implementation of Borůvka's Algorithm for Minimum Spanning Tree

Gadekar S.B.(CS22BTECH11022)
Siddhant Godbole(CS22BTECH11054)

November 9, 2025

Contents

1 Introduction

1.1 The Minimum Spanning Tree Problem

1.2 Borůvka's Algorithm Overview

Borůvka's algorithm, discovered by Czech mathematician Otakar Borůvka in 1926, is one of the algorithms for finding MSTs.

The algorithm works by repeatedly finding the smallest edge connecting each connected component to other components, then merging these components together. This process continues until only one component remains, which is the MST.

2 Theoretical Background

2.1 Algorithm Description

Borůvka's algorithm operates in iterations. Each iteration consists of two main phases:

1. **Selection Phase**: For each connected component, find the minimum-weight edge that connects it to a different component
2. **Merge Phase**: Add all selected edges to the MST and merge the components they connect

The algorithm terminates when all vertices belong to a single component.

2.2 Detailed Algorithm Steps

Algorithm 1 Borůvka's Algorithm

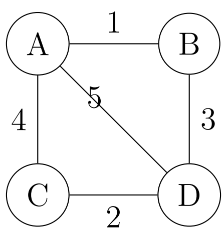
```
Initialize each vertex as its own component
MST ← ∅                                              ▷ Set of MST edges
numComponents ← n                                    ▷ Number of components
while numComponents > 1 do
    Create array minEdge of size n, initialized to infinity
    for each edge (u, v, w) in the graph do
        comp_u ← component of u
        comp_v ← component of v
        if comp_u ≠ comp_v then
            if w < minEdge[comp_u].weight then
                minEdge[comp_u] ← (u, v, w)
            end if
            if w < minEdge[comp_v].weight then
                minEdge[comp_v] ← (u, v, w)
            end if
        end if
    end for
    for each component i do
        if minEdge[i] exists then
            Add minEdge[i] to MST (if not already added)
            Merge the components connected by minEdge[i]
            Decrement numComponents
        end if
    end for
end while
return MST
```

2.3 Key Properties

- **Time Complexity**: $O(E \log V)$ where E is the number of edges and V is the number of vertices. In each iteration, the number of components is reduced by at least half, leading to $O(\log V)$ iterations.
- **Space Complexity**: $O(V + E)$ for storing the graph and auxiliary data structures.
- **Number of Iterations**: At most $O(\log V)$ iterations, since each iteration reduces the number of components by at least a factor of two in the best case.

2.4 Example Walkthrough

Consider a simple graph with 4 vertices and the following edges:



Iteration 1:

- Each vertex starts as its own component: $\{A\}, \{B\}, \{C\}, \{D\}$
- Component $\{A\}$: Minimum edge is $(A, B, 1)$
- Component $\{B\}$: Minimum edge is $(A, B, 1)$
- Component $\{C\}$: Minimum edge is $(C, D, 2)$
- Component $\{D\}$: Minimum edge is $(C, D, 2)$
- Add edges: $(A, B, 1)$ and $(C, D, 2)$ to MST
- New components: $\{A, B\}, \{C, D\}$

Iteration 2:

- Component $\{A, B\}$: Minimum edge to other component is $(B, D, 3)$
- Component $\{C, D\}$: Minimum edge to other component is $(B, D, 3)$
- Add edge: $(B, D, 3)$ to MST
- New component: $\{A, B, C, D\}$

Result: MST with edges $\{(A, B, 1), (C, D, 2), (B, D, 3)\}$ and total weight $= 6$

3 Correctness of the Algorithm

3.1 The Cut Property

To prove that Borůvka's algorithm produces a correct MST, we rely on the **cut property**, a fundamental theorem in graph theory.

Definition (Cut): A cut $(S, V - S)$ of a graph $G = (V, E)$ is a partition of the vertices into two non-empty sets S and $V - S$.

Definition (Crossing Edge): An edge (u, v) crosses the cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.

Cut Property Theorem: Let $G = (V, E)$ be a connected, weighted graph. For any cut $(S, V - S)$ of G, if edge e is the minimum-weight edge crossing the cut, and no edge in the current MST crosses the cut, then e is a **safe edge** that can be added to the MST without violating the MST properties.

3.2 Proof of Correctness

We prove that Borůvka's algorithm produces a correct MST through the following arguments:

Claim: Every edge selected by Borůvka's algorithm is a safe edge that belongs to some MST.

Proof:

1. **Base Case:** At the start, each vertex is its own component, and the MST is empty. Any edge we add connects two components and is valid.

2. **Inductive Step:** Assume that after k iterations, all edges in our partial MST are safe edges. Consider iteration $k+1$:
   - Let $C_1, C_2, \ldots, C_m$ be the current components
   - For each component C_i, we select the minimum-weight edge $e_i = (u, v)$ where $u \in C_i$ and $v \notin C_i$
   - Consider the cut $(C_i, V - C_i)$. The edge e_i is the minimum-weight edge crossing this cut
   - By the cut property, e_i is a safe edge that can be added to the MST
   - Since this holds for all components, all edges selected in iteration $k+1$ are safe
3. **Termination:** The algorithm terminates when only one component remains. At this point, we have selected $n-1$ edges (where n is the number of vertices), and all vertices are connected. This forms a spanning tree.
4. **Minimality:** Since every edge was selected using the cut property (which guarantees minimum weight for crossing edges), the resulting spanning tree has minimum total weight.

3.3 No Cycles Are Created

Claim: Borůvka's algorithm never creates a cycle.

Proof: In each iteration, we only add edges that connect different components. By definition, vertices in different components are not yet connected in the current forest. Therefore, adding an edge between two different components cannot create a cycle. A cycle would require both endpoints of an edge to already be in the same component, which our algorithm explicitly avoids.

3.4 Optimality Guarantee

Since Borůvka's algorithm:

1. Only adds safe edges (by the cut property)
2. Connects all vertices (forms a spanning tree)
3. Terminates with exactly $n-1$ edges
4. Each edge added is the minimum-weight edge connecting its component to another

We conclude that the algorithm produces a Minimum Spanning Tree.

4 Parallelization Concept

4.1 Parallel Phases

The parallelization strategy divides the algorithm into two phases:

4.1.1 Phase 1: Parallel Minimum Edge Finding

This is the main parallelizable phase. Multiple threads work simultaneously to find the minimum-weight edge for different components:

- Divide the components among available threads
- Each thread processes a subset of components
- For each assigned component, the thread scans all edges to find the minimum edge connecting to a different component
- Threads work independently without interfering with each other

4.1.2 Phase 2: Sequential Merging

After all threads complete their searches, the main thread merges components:

- Collect minimum edges found by all threads
- Add edges to the MST (avoiding duplicates)
- Update the Union-Find structure to reflect merged components
- This phase must be sequential to maintain consistency

4.2 Synchronization Requirements

To ensure correctness in a parallel environment, the implementation must handle:

1. **Data Race Prevention:** Multiple threads must not simultaneously modify shared data structures
2. **Atomic Operations:** Union-Find operations (find and unite) must be atomic to prevent corruption
3. **Thread Coordination:** All threads must complete the search phase before the merge phase begins
4. **Logging Coordination:** Thread-safe logging to avoid interleaved output

4.3 Synchronization Mechanisms

The implementation uses several synchronization primitives:

- **Mutex Locks:** Protect critical sections like Union-Find operations and logging
- **Thread Joining:** Ensures all threads complete before proceeding to the merge phase

5 Implementation in MAIN.cpp

5.1 Overall Structure

The implementation consists of several key components:

1. `Edge` struct: Represents a weighted edge
2. `UnionFind` class: Manages component connectivity
3. `ParallelMST` class: Orchestrates the parallel MST computation
4. `main` function: Entry point and parameter handling

5.2 Edge Structure

```
struct Edge {
    int u, v, weight;
    Edge(int u = 0, int v = 0, int w = 0)
        : u(u), v(v), weight(w) {}
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};
```

The `Edge` struct stores:

- `u, v`: The two endpoints of the edge
- `weight`: The edge weight
- Comparison operator: Enables sorting edges by weight

5.3 Union-Find Data Structure

The Union-Find (also called Disjoint Set Union) data structure efficiently tracks which vertices belong to which component:

```
class UnionFind {
private:
    vector<int> parent, rank;
    mutex uf_mutex;
public:
    UnionFind(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);
        return parent[x];
    }
```

```cpp

    bool unite(int x, int y) {
        lock_guard<mutex> lock(uf_mutex);
        int px = find(x), py = find(y);
        if (px == py) return false;

        if (rank[px] < rank[py]) swap(px, py);
        parent[py] = px;
        if (rank[px] == rank[py]) rank[px]++;
        return true;
    }
};
```

Key Operations:

- **find(x)**: Returns the representative (root) of the component containing x. Uses path compression for efficiency.
- **unite(x, y)**: Merges the components containing x and y. Returns true if they were in different components. Uses union by rank for balanced trees.
- **Thread Safety:** The `unite` operation is protected by a mutex since it modifies shared state.

5.4 ParallelMST Class

5.4.1 Data Members

```cpp
class ParallelMST {
private:
    int n;                              // Number of vertices
    vector<vector<int>> adj_matrix;     // Adjacency matrix
    vector<Edge> mst_edges;             // MST result
    ofstream log_file;                  // Log output
    mutex log_mutex, mst_mutex;         // Thread safety
    int num_threads;                    // Thread count
    map<thread::id, int> thread_id_map; // Thread identification
```

5.4.2 Constructor and Graph Loading

```cpp
ParallelMST(const string& input_file, int threads = 4)
    : num_threads(threads) {
    log_file.open("mst_log.txt");
    // ... logging setup ...

    ifstream fin(input_file);
    fin >> n;
    adj_matrix.resize(n, vector<int>(n));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
```

```
            fin >> adj_matrix[i][j];
        }
    }
    fin.close();
}
```

The constructor:

1. Opens a log file for detailed execution tracking
2. Reads the input file containing the adjacency matrix
3. Stores the graph as an $n \times n$ matrix where `adj_matrix[i][j]` is the weight of edge (i, j), or 0 if no edge exists

5.4.3 Parallel Minimum Edge Finding

```
void find_min_edges_parallel(UnionFind& uf,
                             vector<Edge>& edges,
                             vector<Edge>& min_edges,
                             int start, int end) {
    log("Thread␣started␣processing␣components␣" +
        to_string(start) + "␣to␣" + to_string(end));

    for (int comp = start; comp < end; comp++) {
        Edge min_edge(-1, -1, INT_MAX);

        for (const auto& e : edges) {
            int comp_u = uf.find(e.u);
            int comp_v = uf.find(e.v);

            if (comp_u == comp && comp_v != comp) {
                if (e.weight < min_edge.weight) {
                    min_edge = e;
                }
            } else if (comp_v == comp && comp_u != comp) {
                if (e.weight < min_edge.weight) {
                    min_edge = e;
                }
            }
        }

        if (min_edge.u != -1) {
            min_edges[comp] = min_edge;
        }
    }

    log("Thread␣finished␣processing␣components");
}
```

This function is executed by each worker thread:

1. **Input:** A range of component IDs to process (`start` to `end`)

2. **Process:** For each assigned component:
   - Scan all edges in the graph
   - Find edges where one endpoint is in the current component and the other is not
   - Track the minimum-weight such edge
3. **Output:** Store the minimum edge in the `min_edges` array

Thread Safety: This function only reads from shared structures (`uf`, `edges`) and writes to non-overlapping positions in `min_edges`, avoiding race conditions.

5.4.4 Main MST Computation

```
void compute_mst() {
    auto start_time = high_resolution_clock::now();
    thread_id_map[this_thread::get_id()] = 0;

    // Convert adjacency matrix to edge list
    vector<Edge> edges;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (adj_matrix[i][j] > 0) {
                edges.push_back(Edge(i, j, adj_matrix[i][j]));
            }
        }
    }

    UnionFind uf(n);
    int num_components = n;
    int iteration = 0;

    while (num_components > 1) {
        iteration++;
        vector<Edge> min_edges(n, Edge(-1, -1, INT_MAX));

        // PARALLEL PHASE: Spawn worker threads
        vector<thread> threads;
        int chunk_size = (n + num_threads - 1) / num_threads;

        for (int t = 0; t < num_threads; t++) {
            int start = t * chunk_size;
            int end = min(start + chunk_size, n);
            if (start < n) {
                threads.emplace_back([this, t, &uf, &edges,
                                     &min_edges, start, end]() {
                    {
                        lock_guard<mutex> lock(log_mutex);
                        thread_id_map[this_thread::get_id()] = t +
                            1;
                    }
```

```
                        this->find_min_edges_parallel(uf, edges,
                                                      min_edges, start, end);
                    });
                }
            }

            // Wait for all threads to complete
            for (auto& th : threads) {
                th.join();
            }

            // SEQUENTIAL PHASE: Merge components
            int edges_added = 0;
            for (int i = 0; i < n; i++) {
                if (min_edges[i].u != -1) {
                    int comp_u = uf.find(min_edges[i].u);
                    int comp_v = uf.find(min_edges[i].v);

                    if (comp_u != comp_v) {
                        if (uf.unite(comp_u, comp_v)) {
                            lock_guard<mutex> lock(mst_mutex);
                            mst_edges.push_back(min_edges[i]);
                            edges_added++;
                            num_components--;
                            // ... logging ...
                        }
                    }
                }
            }

            if (edges_added == 0 && num_components > 1) {
                log("WARNING: Graph may be disconnected");
                break;
            }
        }

        auto end_time = high_resolution_clock::now();
        // ... timing and logging ...
}
```

Key Steps:

1. **Initialization:**
    - Convert adjacency matrix to edge list
    - Initialize Union-Find with n components
    - Start timing
2. **Main Loop:** While more than one component exists:
    - Calculate chunk size for dividing work among threads

- Spawn worker threads, each processing a range of components
- Each thread finds minimum edges for its assigned components
- Wait for all threads using `join()`
- Sequentially add selected edges and merge components

3. **Termination:**
   - Algorithm stops when one component remains
   - Handles disconnected graphs by detecting when no edges are added

5.5 Thread Coordination

5.5.1 Work Distribution

The work is divided among threads using a simple chunking strategy:

```
int chunk_size = (n + num_threads - 1) / num_threads;
for (int t = 0; t < num_threads; t++) {
    int start = t * chunk_size;
    int end = min(start + chunk_size, n);
    // Spawn thread to process components [start, end)
}
```

This ensures that:

- Each thread gets approximately $n/\text{num\_threads}$ components
- All components are covered
- The last thread handles any remainder

5.5.2 Barrier Synchronization

```
for (auto& th : threads) {
    th.join();
}
```

The `join()` calls create a barrier: the main thread waits until all worker threads complete before proceeding to the merge phase. This is crucial for correctness.

5.5.3 Mutex Protection

Critical sections are protected by mutexes:

```
// Protecting MST edge addition
lock_guard<mutex> lock(mst_mutex);
mst_edges.push_back(min_edges[i]);
```

```
// Protecting Union-Find operations
bool unite(int x, int y) {
    lock_guard<mutex> lock(uf_mutex);
    // ... merge components ...
}
```

The `lock_guard` uses RAII (Resource Acquisition Is Initialization) to automatically release locks when exiting scope, preventing deadlocks.

6 Performance Discussion

6.1 Speedup Analysis

The parallel implementation offers speedup through:

1. **Parallel Minimum Edge Search:** The most computationally intensive phase is finding minimum edges for each component. With p threads processing n components, the work is divided approximately as $O(En/p)$ per thread, where E is the number of edges.

2. **Iteration Reduction:** Borůvka's algorithm typically requires $O(\log n)$ iterations, which is relatively small. The benefit of parallelization within each iteration outweighs the overhead.

3. **Load Balancing:** Chunking strategy ensures relatively balanced workloads across threads, though some imbalance may occur if component sizes vary significantly.

6.2 Limitations and Overheads

6.2.1 Synchronization Overhead

- Thread creation and destruction cost
- Barrier synchronization at the end of each iteration
- Mutex contention in Union-Find operations
- Context switching between threads

For sparse graphs with few edges, synchronization overhead may dominate, reducing speedup.

6.2.2 Memory Contention

Multiple threads accessing the Union-Find structure simultaneously can cause:

- Cache invalidation
- False sharing (threads accessing nearby memory locations)
- Memory bandwidth saturation

6.3 Comparison with Sequential Implementation

6.3.1 Dense Graphs

For dense graphs with $E \approx V^2$:

- Each iteration scans many edges per component
- Parallelization provides significant speedup ($2\times$ to $4\times$ with 4 threads)
- Synchronization overhead is small relative to computation

6.3.2 Sparse Graphs

For sparse graphs with $E \approx V$:

- Less work per iteration
- Synchronization overhead becomes more significant
- Speedup may be modest ($1.5\times$ to $2\times$ with 4 threads)

6.3.3 Small Graphs

For graphs with $V < 100$:

- Thread creation overhead may exceed computational savings
- Sequential version might be faster
- Parallel version still correct but not necessarily faster

7 Complexity Analysis Summary

7.1 Time Complexity

- **Iterations:** $O(\log V)$ iterations maximum
- **Per Iteration (Sequential):** $O(E)$ to scan all edges
- **Per Iteration (Parallel):** $O(E/p)$ with p processors
- **Union-Find Operations:** Nearly $O(1)$ amortized with path compression and union by rank
- **Overall Sequential:** $O(E \log V)$
- **Overall Parallel:** $O((E/p) \log V + \log V \cdot S)$ where S is synchronization cost

7.2 Space Complexity

- Adjacency Matrix: $O(V^2)$
- Edge List: $O(E)$
- Union-Find: $O(V)$
- Minimum Edge Array: $O(V)$
- MST Result: $O(V)$
- **Total:** $O(V^2 + E) = O(V^2)$ for the matrix representation