

Parallel Implementation of Borůvka's Algorithm for Minimum Spanning Tree

Gadekar S.B.(CS22BTECH11022)

Siddhant Godbole(CS22BTECH11054)

November 9, 2025

Contents

1	Introduction	3
1.1	The Minimum Spanning Tree Problem	3
1.2	Borůvka's Algorithm Overview	3
2	Theoretical Background	3
2.1	Algorithm Description	3
2.2	Detailed Algorithm Steps	3
2.3	Key Properties	4
2.4	Example Walkthrough	4
3	Correctness of the Algorithm	5
3.1	The Cut Property	5
3.2	Proof of Correctness	5
3.3	No Cycles Are Created	6
3.4	Optimality Guarantee	6
4	Parallelization Concept	6
4.1	Parallel Phases	6
4.1.1	Phase 1: Parallel Minimum Edge Finding	7
4.1.2	Phase 2: Sequential Merging	7
4.2	Synchronization Requirements	7
4.3	Synchronization Mechanisms	7
5	Implementation in MAIN.cpp	8
5.1	Overall Structure	8
5.2	Edge Structure	8
5.3	Union-Find Data Structure	8
5.4	ParallelMST Class	9
5.4.1	Data Members	9
5.4.2	Constructor and Graph Loading	9
5.4.3	Parallel Minimum Edge Finding	10

5.4.4	Main MST Computation	11
5.5	Thread Coordination	13
5.5.1	Work Distribution	13
5.5.2	Barrier Synchronization	13
5.5.3	Mutex Protection	13
6	Performance Discussion	14
6.1	Speedup Analysis	14
6.2	Limitations and Overheads	14
6.2.1	Synchronization Overhead	14
6.2.2	Memory Contention	14
6.3	Comparison with Sequential Implementation	15
6.3.1	Dense Graphs	15
6.3.2	Sparse Graphs	15
6.3.3	Small Graphs	15
7	Complexity Analysis Summary	15
7.1	Time Complexity	15
7.2	Space Complexity	16

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 Boruvka's Algorithm

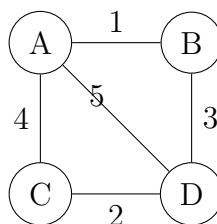
```
1: Initialize each vertex as its own component
2:  $MST \leftarrow \emptyset$                                  $\triangleright$  Set of MST edges
3:  $numComponents \leftarrow n$                           $\triangleright$  Number of components
4: while  $numComponents > 1$  do
5:   Create array  $minEdge$  of size  $n$ , initialized to infinity
6:   for each edge  $(u, v, w)$  in the graph do
7:      $comp_u \leftarrow$  component of  $u$ 
8:      $comp_v \leftarrow$  component of  $v$ 
9:     if  $comp_u \neq comp_v$  then
10:      if  $w < minEdge[comp_u].weight$  then
11:         $minEdge[comp_u] \leftarrow (u, v, w)$ 
12:      end if
13:      if  $w < minEdge[comp_v].weight$  then
14:         $minEdge[comp_v] \leftarrow (u, v, w)$ 
15:      end if
16:    end if
17:   end for
18:   for each component  $i$  do
19:     if  $minEdge[i]$  exists then
20:       Add  $minEdge[i]$  to  $MST$  (if not already added)
21:       Merge the components connected by  $minEdge[i]$ 
22:       Decrement  $numComponents$ 
23:     end if
24:   end for
25: end while
26: 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, \dots, 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

```
1 struct Edge {
2     int u, v, weight;
3     Edge(int u = 0, int v = 0, int w = 0)
4         : u(u), v(v), weight(w) {}
5     bool operator<(const Edge& other) const {
6         return weight < other.weight;
7     }
8 };
```

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:

```
1 class UnionFind {
2 private:
3     vector<int> parent, rank;
4     mutex uf_mutex;
5 public:
6     UnionFind(int n) : parent(n), rank(n, 0) {
7         for (int i = 0; i < n; i++)
8             parent[i] = i;
9     }
10
11     int find(int x) {
12         if (parent[x] != x)
13             parent[x] = find(parent[x]);
14         return parent[x];
15     }
}
```

```

16
17     bool unite(int x, int y) {
18         lock_guard<mutex> lock(uf_mutex);
19         int px = find(x), py = find(y);
20         if (px == py) return false;
21
22         if (rank[px] < rank[py]) swap(px, py);
23         parent[py] = px;
24         if (rank[px] == rank[py]) rank[px]++;
25         return true;
26     }
27 }

```

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

```

1 class ParallelMST {
2 private:
3     int n;                                     // Number of vertices
4     vector<vector<int>> adj_matrix;           // Adjacency matrix
5     vector<Edge> mst_edges;                   // MST result
6     ofstream log_file;                        // Log output
7     mutex log_mutex, mst_mutex;               // Thread safety
8     int num_threads;                         // Thread count
9     map<thread::id, int> thread_id_map; // Thread identification

```

5.4.2 Constructor and Graph Loading

```

1 ParallelMST(const string& input_file, int threads = 4)
2     : num_threads(threads) {
3     log_file.open("mst_log.txt");
4     // ... logging setup ...
5
6     ifstream fin(input_file);
7     fin >> n;
8     adj_matrix.resize(n, vector<int>(n));
9
10    for (int i = 0; i < n; i++) {
11        for (int j = 0; j < n; j++) {

```

```

12         fin >> adj_matrix[i][j];
13     }
14 }
15 fin.close();
16 }
```

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

```

1 void find_min_edges_parallel(UnionFind& uf,
2                             vector<Edge>& edges,
3                             vector<Edge>& min_edges,
4                             int start, int end) {
5     log("Thread started processing components " +
6         to_string(start) + " to " + to_string(end));
7
8     for (int comp = start; comp < end; comp++) {
9         Edge min_edge(-1, -1, INT_MAX);
10
11         for (const auto& e : edges) {
12             int comp_u = uf.find(e.u);
13             int comp_v = uf.find(e.v);
14
15             if (comp_u == comp && comp_v != comp) {
16                 if (e.weight < min_edge.weight) {
17                     min_edge = e;
18                 }
19             } else if (comp_v == comp && comp_u != comp) {
20                 if (e.weight < min_edge.weight) {
21                     min_edge = e;
22                 }
23             }
24         }
25
26         if (min_edge.u != -1) {
27             min_edges[comp] = min_edge;
28         }
29     }
30
31     log("Thread finished processing components");
32 }
```

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

```
1 void compute_mst() {
2     auto start_time = high_resolution_clock::now();
3     thread_id_map[this_thread::get_id()] = 0;
4
5     // Convert adjacency matrix to edge list
6     vector<Edge> edges;
7     for (int i = 0; i < n; i++) {
8         for (int j = i + 1; j < n; j++) {
9             if (adj_matrix[i][j] > 0) {
10                 edges.push_back(Edge(i, j, adj_matrix[i][j]));
11             }
12         }
13     }
14
15     UnionFind uf(n);
16     int num_components = n;
17     int iteration = 0;
18
19     while (num_components > 1) {
20         iteration++;
21         vector<Edge> min_edges(n, Edge(-1, -1, INT_MAX));
22
23         // PARALLEL PHASE: Spawn worker threads
24         vector<thread> threads;
25         int chunk_size = (n + num_threads - 1) / num_threads;
26
27         for (int t = 0; t < num_threads; t++) {
28             int start = t * chunk_size;
29             int end = min(start + chunk_size, n);
30             if (start < n) {
31                 threads.emplace_back([this, t, &uf, &edges,
32                                     &min_edges, start, end]() {
33
34                     lock_guard<mutex> lock(log_mutex);
35                     thread_id_map[this_thread::get_id()] = t +
36                     1;
37                 });
38             }
39         }
40
41         for (int i = 0; i < n; i++) {
42             if (uf.parent[i] == -1) {
43                 num_components--;
44             }
45         }
46
47         for (int i = 0; i < n; i++) {
48             if (uf.parent[i] != -1) {
49                 for (int j = 0; j < n; j++) {
50                     if (uf.parent[j] == -1) {
51                         if (edges[i].weight < edges[j].weight) {
52                             edges[j] = edges[i];
53                         }
54                     }
55                 }
56             }
57         }
58
59         for (int i = 0; i < n; i++) {
60             if (uf.parent[i] == -1) {
61                 num_components--;
62             }
63         }
64
65         for (int i = 0; i < n; i++) {
66             if (uf.parent[i] != -1) {
67                 for (int j = 0; j < n; j++) {
68                     if (uf.parent[j] == -1) {
69                         if (edges[i].weight < edges[j].weight) {
70                             edges[j] = edges[i];
71                         }
72                     }
73                 }
74             }
75         }
76
77         for (int i = 0; i < n; i++) {
78             if (uf.parent[i] == -1) {
79                 num_components--;
80             }
81         }
82
83         for (int i = 0; i < n; i++) {
84             if (uf.parent[i] != -1) {
85                 for (int j = 0; j < n; j++) {
86                     if (uf.parent[j] == -1) {
87                         if (edges[i].weight < edges[j].weight) {
88                             edges[j] = edges[i];
89                         }
90                     }
91                 }
92             }
93         }
94
95         for (int i = 0; i < n; i++) {
96             if (uf.parent[i] == -1) {
97                 num_components--;
98             }
99         }
100
101         for (int i = 0; i < n; i++) {
102             if (uf.parent[i] != -1) {
103                 for (int j = 0; j < n; j++) {
104                     if (uf.parent[j] == -1) {
105                         if (edges[i].weight < edges[j].weight) {
106                             edges[j] = edges[i];
107                         }
108                     }
109                 }
110             }
111         }
112
113         for (int i = 0; i < n; i++) {
114             if (uf.parent[i] == -1) {
115                 num_components--;
116             }
117         }
118
119         for (int i = 0; i < n; i++) {
120             if (uf.parent[i] != -1) {
121                 for (int j = 0; j < n; j++) {
122                     if (uf.parent[j] == -1) {
123                         if (edges[i].weight < edges[j].weight) {
124                             edges[j] = edges[i];
125                         }
126                     }
127                 }
128             }
129         }
130
131         for (int i = 0; i < n; i++) {
132             if (uf.parent[i] == -1) {
133                 num_components--;
134             }
135         }
136
137         for (int i = 0; i < n; i++) {
138             if (uf.parent[i] != -1) {
139                 for (int j = 0; j < n; j++) {
140                     if (uf.parent[j] == -1) {
141                         if (edges[i].weight < edges[j].weight) {
142                             edges[j] = edges[i];
143                         }
144                     }
145                 }
146             }
147         }
148
149         for (int i = 0; i < n; i++) {
150             if (uf.parent[i] == -1) {
151                 num_components--;
152             }
153         }
154
155         for (int i = 0; i < n; i++) {
156             if (uf.parent[i] != -1) {
157                 for (int j = 0; j < n; j++) {
158                     if (uf.parent[j] == -1) {
159                         if (edges[i].weight < edges[j].weight) {
160                             edges[j] = edges[i];
161                         }
162                     }
163                 }
164             }
165         }
166
167         for (int i = 0; i < n; i++) {
168             if (uf.parent[i] == -1) {
169                 num_components--;
170             }
171         }
172
173         for (int i = 0; i < n; i++) {
174             if (uf.parent[i] != -1) {
175                 for (int j = 0; j < n; j++) {
176                     if (uf.parent[j] == -1) {
177                         if (edges[i].weight < edges[j].weight) {
178                             edges[j] = edges[i];
179                         }
180                     }
181                 }
182             }
183         }
184
185         for (int i = 0; i < n; i++) {
186             if (uf.parent[i] == -1) {
187                 num_components--;
188             }
189         }
190
191         for (int i = 0; i < n; i++) {
192             if (uf.parent[i] != -1) {
193                 for (int j = 0; j < n; j++) {
194                     if (uf.parent[j] == -1) {
195                         if (edges[i].weight < edges[j].weight) {
196                             edges[j] = edges[i];
197                         }
198                     }
199                 }
200             }
201         }
202
203         for (int i = 0; i < n; i++) {
204             if (uf.parent[i] == -1) {
205                 num_components--;
206             }
207         }
208
209         for (int i = 0; i < n; i++) {
210             if (uf.parent[i] != -1) {
211                 for (int j = 0; j < n; j++) {
212                     if (uf.parent[j] == -1) {
213                         if (edges[i].weight < edges[j].weight) {
214                             edges[j] = edges[i];
215                         }
216                     }
217                 }
218             }
219         }
220
221         for (int i = 0; i < n; i++) {
222             if (uf.parent[i] == -1) {
223                 num_components--;
224             }
225         }
226
227         for (int i = 0; i < n; i++) {
228             if (uf.parent[i] != -1) {
229                 for (int j = 0; j < n; j++) {
230                     if (uf.parent[j] == -1) {
231                         if (edges[i].weight < edges[j].weight) {
232                             edges[j] = edges[i];
233                         }
234                     }
235                 }
236             }
237         }
238
239         for (int i = 0; i < n; i++) {
240             if (uf.parent[i] == -1) {
241                 num_components--;
242             }
243         }
244
245         for (int i = 0; i < n; i++) {
246             if (uf.parent[i] != -1) {
247                 for (int j = 0; j < n; j++) {
248                     if (uf.parent[j] == -1) {
249                         if (edges[i].weight < edges[j].weight) {
250                             edges[j] = edges[i];
251                         }
252                     }
253                 }
254             }
255         }
256
257         for (int i = 0; i < n; i++) {
258             if (uf.parent[i] == -1) {
259                 num_components--;
260             }
261         }
262
263         for (int i = 0; i < n; i++) {
264             if (uf.parent[i] != -1) {
265                 for (int j = 0; j < n; j++) {
266                     if (uf.parent[j] == -1) {
267                         if (edges[i].weight < edges[j].weight) {
268                             edges[j] = edges[i];
269                         }
270                     }
271                 }
272             }
273         }
274
275         for (int i = 0; i < n; i++) {
276             if (uf.parent[i] == -1) {
277                 num_components--;
278             }
279         }
280
281         for (int i = 0; i < n; i++) {
282             if (uf.parent[i] != -1) {
283                 for (int j = 0; j < n; j++) {
284                     if (uf.parent[j] == -1) {
285                         if (edges[i].weight < edges[j].weight) {
286                             edges[j] = edges[i];
287                         }
288                     }
289                 }
290             }
291         }
292
293         for (int i = 0; i < n; i++) {
294             if (uf.parent[i] == -1) {
295                 num_components--;
296             }
297         }
298
299         for (int i = 0; i < n; i++) {
300             if (uf.parent[i] != -1) {
301                 for (int j = 0; j < n; j++) {
302                     if (uf.parent[j] == -1) {
303                         if (edges[i].weight < edges[j].weight) {
304                             edges[j] = edges[i];
305                         }
306                     }
307                 }
308             }
309         }
310
311         for (int i = 0; i < n; i++) {
312             if (uf.parent[i] == -1) {
313                 num_components--;
314             }
315         }
316
317         for (int i = 0; i < n; i++) {
318             if (uf.parent[i] != -1) {
319                 for (int j = 0; j < n; j++) {
320                     if (uf.parent[j] == -1) {
321                         if (edges[i].weight < edges[j].weight) {
322                             edges[j] = edges[i];
323                         }
324                     }
325                 }
326             }
327         }
328
329         for (int i = 0; i < n; i++) {
330             if (uf.parent[i] == -1) {
331                 num_components--;
332             }
333         }
334
335         for (int i = 0; i < n; i++) {
336             if (uf.parent[i] != -1) {
337                 for (int j = 0; j < n; j++) {
338                     if (uf.parent[j] == -1) {
339                         if (edges[i].weight < edges[j].weight) {
340                             edges[j] = edges[i];
341                         }
342                     }
343                 }
344             }
345         }
346
347         for (int i = 0; i < n; i++) {
348             if (uf.parent[i] == -1) {
349                 num_components--;
350             }
351         }
352
353         for (int i = 0; i < n; i++) {
354             if (uf.parent[i] != -1) {
355                 for (int j = 0; j < n; j++) {
356                     if (uf.parent[j] == -1) {
357                         if (edges[i].weight < edges[j].weight) {
358                             edges[j] = edges[i];
359                         }
360                     }
361                 }
362             }
363         }
364
365         for (int i = 0; i < n; i++) {
366             if (uf.parent[i] == -1) {
367                 num_components--;
368             }
369         }
370
371         for (int i = 0; i < n; i++) {
372             if (uf.parent[i] != -1) {
373                 for (int j = 0; j < n; j++) {
374                     if (uf.parent[j] == -1) {
375                         if (edges[i].weight < edges[j].weight) {
376                             edges[j] = edges[i];
377                         }
378                     }
379                 }
380             }
381         }
382
383         for (int i = 0; i < n; i++) {
384             if (uf.parent[i] == -1) {
385                 num_components--;
386             }
387         }
388
389         for (int i = 0; i < n; i++) {
390             if (uf.parent[i] != -1) {
391                 for (int j = 0; j < n; j++) {
392                     if (uf.parent[j] == -1) {
393                         if (edges[i].weight < edges[j].weight) {
394                             edges[j] = edges[i];
395                         }
396                     }
397                 }
398             }
399         }
400
401         for (int i = 0; i < n; i++) {
402             if (uf.parent[i] == -1) {
403                 num_components--;
404             }
405         }
406
407         for (int i = 0; i < n; i++) {
408             if (uf.parent[i] != -1) {
409                 for (int j = 0; j < n; j++) {
410                     if (uf.parent[j] == -1) {
411                         if (edges[i].weight < edges[j].weight) {
412                             edges[j] = edges[i];
413                         }
414                     }
415                 }
416             }
417         }
418
419         for (int i = 0; i < n; i++) {
420             if (uf.parent[i] == -1) {
421                 num_components--;
422             }
423         }
424
425         for (int i = 0; i < n; i++) {
426             if (uf.parent[i] != -1) {
427                 for (int j = 0; j < n; j++) {
428                     if (uf.parent[j] == -1) {
429                         if (edges[i].weight < edges[j].weight) {
430                             edges[j] = edges[i];
431                         }
432                     }
433                 }
434             }
435         }
436
437         for (int i = 0; i < n; i++) {
438             if (uf.parent[i] == -1) {
439                 num_components--;
440             }
441         }
442
443         for (int i = 0; i < n; i++) {
444             if (uf.parent[i] != -1) {
445                 for (int j = 0; j < n; j++) {
446                     if (uf.parent[j] == -1) {
447                         if (edges[i].weight < edges[j].weight) {
448                             edges[j] = edges[i];
449                         }
450                     }
451                 }
452             }
453         }
454
455         for (int i = 0; i < n; i++) {
456             if (uf.parent[i] == -1) {
457                 num_components--;
458             }
459         }
460
461         for (int i = 0; i < n; i++) {
462             if (uf.parent[i] != -1) {
463                 for (int j = 0; j < n; j++) {
464                     if (uf.parent[j] == -1) {
465                         if (edges[i].weight < edges[j].weight) {
466                             edges[j] = edges[i];
467                         }
468                     }
469                 }
470             }
471         }
472
473         for (int i = 0; i < n; i++) {
474             if (uf.parent[i] == -1) {
475                 num_components--;
476             }
477         }
478
479         for (int i = 0; i < n; i++) {
480             if (uf.parent[i] != -1) {
481                 for (int j = 0; j < n; j++) {
482                     if (uf.parent[j] == -1) {
483                         if (edges[i].weight < edges[j].weight) {
484                             edges[j] = edges[i];
485                         }
486                     }
487                 }
488             }
489         }
490
491         for (int i = 0; i < n; i++) {
492             if (uf.parent[i] == -1) {
493                 num_components--;
494             }
495         }
496
497         for (int i = 0; i < n; i++) {
498             if (uf.parent[i] != -1) {
499                 for (int j = 0; j < n; j++) {
500                     if (uf.parent[j] == -1) {
501                         if (edges[i].weight < edges[j].weight) {
502                             edges[j] = edges[i];
503                         }
504                     }
505                 }
506             }
507         }
508
509         for (int i = 0; i < n; i++) {
510             if (uf.parent[i] == -1) {
511                 num_components--;
512             }
513         }
514
515         for (int i = 0; i < n; i++) {
516             if (uf.parent[i] != -1) {
517                 for (int j = 0; j < n; j++) {
518                     if (uf.parent[j] == -1) {
519                         if (edges[i].weight < edges[j].weight) {
520                             edges[j] = edges[i];
521                         }
522                     }
523                 }
524             }
525         }
526
527         for (int i = 0; i < n; i++) {
528             if (uf.parent[i] == -1) {
529                 num_components--;
530             }
531         }
532
533         for (int i = 0; i < n; i++) {
534             if (uf.parent[i] != -1) {
535                 for (int j = 0; j < n; j++) {
536                     if (uf.parent[j] == -1) {
537                         if (edges[i].weight < edges[j].weight) {
538                             edges[j] = edges[i];
539                         }
540                     }
541                 }
542             }
543         }
544
545         for (int i = 0; i < n; i++) {
546             if (uf.parent[i] == -1) {
547                 num_components--;
548             }
549         }
550
551         for (int i = 0; i < n; i++) {
552             if (uf.parent[i] != -1) {
553                 for (int j = 0; j < n; j++) {
554                     if (uf.parent[j] == -1) {
555                         if (edges[i].weight < edges[j].weight) {
556                             edges[j] = edges[i];
557                         }
558                     }
559                 }
560             }
561         }
562
563         for (int i = 0; i < n; i++) {
564             if (uf.parent[i] == -1) {
565                 num_components--;
566             }
567         }
568
569         for (int i = 0; i < n; i++) {
570             if (uf.parent[i] != -1) {
571                 for (int j = 0; j < n; j++) {
572                     if (uf.parent[j] == -1) {
573                         if (edges[i].weight < edges[j].weight) {
574                             edges[j] = edges[i];
575                         }
576                     }
577                 }
578             }
579         }
580
581         for (int i = 0; i < n; i++) {
582             if (uf.parent[i] == -1) {
583                 num_components--;
584             }
585         }
586
587         for (int i = 0; i < n; i++) {
588             if (uf.parent[i] != -1) {
589                 for (int j = 0; j < n; j++) {
590                     if (uf.parent[j] == -1) {
591                         if (edges[i].weight < edges[j].weight) {
592                             edges[j] = edges[i];
593                         }
594                     }
595                 }
596             }
597         }
598
599         for (int i = 0; i < n; i++) {
600             if (uf.parent[i] == -1) {
601                 num_components--;
602             }
603         }
604
605         for (int i = 0; i < n; i++) {
606             if (uf.parent[i] != -1) {
607                 for (int j = 0; j < n; j++) {
608                     if (uf.parent[j] == -1) {
609                         if (edges[i].weight < edges[j].weight) {
610                             edges[j] = edges[i];
611                         }
612                     }
613                 }
614             }
615         }
616
617         for (int i = 0; i < n; i++) {
618             if (uf.parent[i] == -1) {
619                 num_components--;
620             }
621         }
622
623         for (int i = 0; i < n; i++) {
624             if (uf.parent[i] != -1) {
625                 for (int j = 0; j < n; j++) {
626                     if (uf.parent[j] == -1) {
627                         if (edges[i].weight < edges[j].weight) {
628                             edges[j] = edges[i];
629                         }
630                     }
631                 }
632             }
633         }
634
635         for (int i = 0; i < n; i++) {
636             if (uf.parent[i] == -1) {
637                 num_components--;
638             }
639         }
640
641         for (int i = 0; i < n; i++) {
642             if (uf.parent[i] != -1) {
643                 for (int j = 0; j < n; j++) {
644                     if (uf.parent[j] == -1) {
645                         if (edges[i].weight < edges[j].weight) {
646                             edges[j] = edges[i];
647                         }
648                     }
649                 }
650             }
651         }
652
653         for (int i = 0; i < n; i++) {
654             if (uf.parent[i] == -1) {
655                 num_components--;
656             }
657         }
658
659         for (int i = 0; i < n; i++) {
660             if (uf.parent[i] != -1) {
661                 for (int j = 0; j < n; j++) {
662                     if (uf.parent[j] == -1) {
663                         if (edges[i].weight < edges[j].weight) {
664                             edges[j] = edges[i];
665                         }
666                     }
667                 }
668             }
669         }
670
671         for (int i = 0; i < n; i++) {
672             if (uf.parent[i] == -1) {
673                 num_components--;
674             }
675         }
676
677         for (int i = 0; i < n; i++) {
678             if (uf.parent[i] != -1) {
679                 for (int j = 0; j < n; j++) {
680                     if (uf.parent[j] == -1) {
681                         if (edges[i].weight < edges[j].weight) {
682                             edges[j] = edges[i];
683                         }
684                     }
685                 }
686             }
687         }
688
689         for (int i = 0; i < n; i++) {
690             if (uf.parent[i] == -1) {
691                 num_components--;
692             }
693         }
694
695         for (int i = 0; i < n; i++) {
696             if (uf.parent[i] != -1) {
697                 for (int j = 0; j < n; j++) {
698                     if (uf.parent[j] == -1) {
699                         if (edges[i].weight < edges[j].weight) {
700                             edges[j] = edges[i];
701                         }
702                     }
703                 }
704             }
705         }
706
707         for (int i = 0; i < n; i++) {
708             if (uf.parent[i] == -1) {
709                 num_components--;
710             }
711         }
712
713         for (int i = 0; i < n; i++) {
714             if (uf.parent[i] != -1) {
715                 for (int j = 0; j < n; j++) {
716                     if (uf.parent[j] == -1) {
717                         if (edges[i].weight < edges[j].weight) {
718                             edges[j] = edges[i];
719                         }
720                     }
721                 }
722             }
723         }
724
725         for (int i = 0; i < n; i++) {
726             if (uf.parent[i] == -1) {
727                 num_components--;
728             }
729         }
730
731         for (int i = 0; i < n; i++) {
732             if (uf.parent[i] != -1) {
733                 for (int j = 0; j < n; j++) {
734                     if (uf.parent[j] == -1) {
735                         if (edges[i].weight < edges[j].weight) {
736                             edges[j] = edges[i];
737                         }
738                     }
739                 }
740             }
741         }
742
743         for (int i = 0; i < n; i++) {
744             if (uf.parent[i] == -1) {
745                 num_components--;
746             }
747         }
748
749         for (int i = 0; i < n; i++) {
750             if (uf.parent[i] != -1) {
751                 for (int j = 0; j < n; j++) {
752                     if (uf.parent[j] == -1) {
753                         if (edges[i].weight < edges[j].weight) {
754                             edges[j] = edges[i];
755                         }
756                     }
757                 }
758             }
759         }
760
761         for (int i = 0; i < n; i++) {
762             if (uf.parent[i] == -1) {
763                 num_components--;
764             }
765         }
766
767         for (int i = 0; i < n; i++) {
768             if (uf.parent[i] != -1) {
769                 for (int j = 0; j < n; j++) {
770                     if (uf.parent[j] == -1) {
771                         if (edges[i].weight < edges[j].weight) {
772                             edges[j] = edges[i];
773                         }
774                     }
775                 }
776             }
777         }
778
779         for (int i = 0; i < n; i++) {
780             if (uf.parent[i] == -1) {
781                 num_components--;
782             }
783         }
784
785         for (int i = 0; i < n; i++) {
786             if (uf.parent[i] != -1) {
787                 for (int j = 0; j < n; j++) {
788                     if (uf.parent[j] == -1) {
789                         if (edges[i].weight < edges[j].weight) {
790                             edges[j] = edges[i];
791                         }
792                     }
793                 }
794             }
795         }
796
797         for (int i = 0; i < n; i++) {
798             if (uf.parent[i] == -1) {
799                 num_components--;
800             }
801         }
802
803         for (int i = 0; i < n; i++) {
804             if (uf.parent[i] != -1) {
805                 for (int j = 0; j < n; j++) {
806                     if (uf.parent[j] == -1) {
807                         if (edges[i].weight < edges[j].weight) {
808                             edges[j] = edges[i];
809                         }
810                     }
811                 }
812             }
813         }
814
815         for (int i = 0; i < n; i++) {
816             if (uf.parent[i] == -1) {
817                 num_components--;
818             }
819         }
820
821         for (int i = 0; i < n; i++) {
822             if (uf.parent[i] != -1) {
823                 for (int j = 0; j < n; j++) {
824                     if (uf.parent[j] == -1) {
825                         if (edges[i].weight < edges[j].weight) {
826                             edges[j] = edges[i];
827                         }
828                     }
829                 }
830             }
831         }
832
833         for (int i = 0; i < n; i++) {
834             if (uf.parent[i] == -1) {
835                 num_components--;
836             }
837         }
838
839         for (int i = 0; i < n; i++) {
840             if (uf.parent[i] != -1) {
841                 for (int j = 0; j < n; j++) {
842                     if (uf.parent[j] == -1) {
843                         if (edges[i].weight < edges[j].weight) {
844                             edges[j] = edges[i];
845                         }
846                     }
847                 }
848             }
849         }
850
851         for (int i = 0; i < n; i++) {
852             if (uf.parent[i] == -1) {
853                 num_components--;
854             }
855         }
856
857         for (int i = 0; i < n; i++) {
858             if (uf.parent[i] != -1) {
859                 for (int j = 0; j < n; j++) {
860                     if (uf.parent[j] == -1) {
861                         if (edges[i].weight < edges[j].weight) {
862                             edges[j] = edges[i];
863                         }
864                     }
865                 }
866             }
867         }
868
869         for (int i = 0; i < n; i++) {
870             if (uf.parent[i] == -1) {
871                 num_components--;
872             }
873         }
874
875         for (int i = 0; i < n; i++) {
876             if (uf.parent[i] != -1) {
877                 for (int j = 0; j < n; j++) {
878                     if (uf.parent[j] == -1) {
879                         if (edges[i].weight < edges[j].weight) {
880                             edges[j] = edges[i];
881                         }
882                     }
883                 }
884             }
885         }
886
887         for (int i = 0; i < n; i++) {
888             if (uf.parent[i] == -1) {
889                 num_components--;
890             }
891         }
892
893         for (int i = 0; i < n; i++) {
894             if (uf.parent[i] != -1) {
895                 for (int j = 0; j < n; j++) {
896                     if (uf.parent[j] == -1) {
897                         if (edges[i].weight < edges[j].weight) {
898                             edges[j] = edges[i];
899                         }
900                     }
901                 }
902             }
903         }
904
905         for (int i = 0; i < n; i++) {
906             if (uf.parent[i] == -1) {
907                 num_components--;
908             }
909         }
910
911         for (int i = 0; i < n; i++) {
912             if (uf.parent[i] != -1) {
913                 for (int j = 0; j < n; j++) {
914                     if (uf.parent[j] == -1) {
915                         if (edges[i].weight < edges[j].weight) {
916                             edges[j] = edges[i];
917                         }
918                     }
919                 }
920             }
921         }
922
923         for (int i = 0; i < n; i++) {
924             if (uf.parent[i] == -1) {
925                 num_components--;
926             }
927         }
928
929         for (int i = 0; i < n; i++) {
930             if (uf.parent[i] != -1) {
931                 for (int j = 0; j < n; j++) {
932                     if (uf.parent[j] == -1) {
933                         if (edges[i].weight < edges[j].weight) {
934                             edges[j] = edges[i];
935                         }
936                     }
937                 }
938             }
939         }
940
941         for (int i = 0; i < n; i++) {
942             if (uf.parent[i] == -1) {
943                 num_components--;
944             }
945         }
946
947         for (int i = 0; i < n; i++) {
948             if (uf.parent[i] != -1) {
949                 for (int j = 0; j < n; j++) {
950                     if (uf.parent[j] == -1) {
951                         if (edges[i].weight < edges[j].weight) {
952                             edges[j] = edges[i];
953                         }
954                     }
955                 }
956             }
957         }
958
959         for (int i = 0; i < n; i++) {
960             if (uf.parent[i] == -1) {
961                 num_components--;
962             }
963         }
964
965         for (int i = 0; i < n; i++) {
966             if (uf.parent[i] != -1) {
967                 for (int j = 0; j < n; j++) {
968                     if (uf.parent[j] == -1) {
969                         if (edges[i].weight < edges[j].weight) {
970                             edges[j] = edges[i];
971                         }
972                     }
973                 }
974             }
975         }
976
977         for (int i = 0; i < n; i++) {
978             if (uf.parent[i] == -1) {
979                 num_components--;
980             }
981         }
982
983         for (int i = 0; i < n; i++) {
984             if (uf.parent[i] != -1) {
985                 for (int j = 0; j < n; j++) {
986                     if (uf.parent[j] == -1) {
987                         if (edges[i].weight < edges[j].weight) {
988                             edges[j] = edges[i];
989                         }
990                     }
991                 }
992             }
993         }
994
995         for (int i = 0; i < n; i++) {
996             if (uf.parent[i] == -1) {
997                 num_components--;
998             }
999         }
1000
1001         for (int i = 0; i < n; i++) {
1002             if (uf.parent[i] != -1) {
1003                 for (int j = 0; j < n; j++) {
1004                     if (uf.parent[j] == -1) {
1005                         if (edges[i].weight < edges[j].weight) {
1006                             edges[j] = edges[i];
1007                         }
1008                     }
1009                 }
1010             }
1011         }
1012
1013         for (int i = 0; i < n; i++) {
1014             if (uf.parent[i] == -1) {
1015                 num_components--;
1016             }
1017         }
1018
1019         for (int i = 0; i < n; i++) {
1020             if (uf.parent[i] != -1) {
1021                 for (int j = 0; j < n; j++) {
1022                     if (uf.parent[j] == -1) {
1023                         if (edges[i].weight < edges[j].weight) {
1024                             edges[j] = edges[i];
1025                         }
1026                     }
1027                 }
1028             }
1029         }
1030
1031         for (int i = 0; i < n; i++) {
1032             if (uf.parent[i] == -1) {
1033                 num_components--;
1034             }
1035         }
1036
1037         for (int i = 0; i < n; i++) {
1038             if (uf.parent[i] != -1) {
1039                 for (int j = 0; j < n; j++) {

```

```

37         this->find_min_edges_parallel(uf, edges,
38                                         min_edges, start,
39                                         end);
40     });
41 }
42
43 // Wait for all threads to complete
44 for (auto& th : threads) {
45     th.join();
46 }
47
48 // SEQUENTIAL PHASE: Merge components
49 int edges_added = 0;
50 for (int i = 0; i < n; i++) {
51     if (min_edges[i].u != -1) {
52         int comp_u = uf.find(min_edges[i].u);
53         int comp_v = uf.find(min_edges[i].v);
54
55         if (comp_u != comp_v) {
56             if (uf.unite(comp_u, comp_v)) {
57                 lock_guard<mutex> lock(mst_mutex);
58                 mst_edges.push_back(min_edges[i]);
59                 edges_added++;
60                 num_components--;
61                 // ... logging ...
62             }
63         }
64     }
65 }
66
67 if (edges_added == 0 && num_components > 1) {
68     log("WARNING: Graph may be disconnected");
69     break;
70 }
71 }
72
73 auto end_time = high_resolution_clock::now();
74 // ... timing and logging ...
75 }

```

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:

```

1 int chunk_size = (n + num_threads - 1) / num_threads;
2 for (int t = 0; t < num_threads; t++) {
3     int start = t * chunk_size;
4     int end = min(start + chunk_size, n);
5     // Spawn thread to process components [start, end)
6 }
```

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

```

1 for (auto& th : threads) {
2     th.join();
3 }
```

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:

```

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

```

1 // Protecting Union-Find operations
2 bool unite(int x, int y) {
3     lock_guard<mutex> lock(uf_mutex);
4     // ... merge components ...
5 }
```

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