Parallelization of Minimum Spanning Tree (MST)

Manohar Maripe Chandrahas Gurrala
Dr. Sathya Peri

Indian Institute of Technology Hyderabad

Introduction

- **Minimum Spanning Tree (MST)**: A **MST** of a connected, undirected graph is a subset of edges that connects all vertices without cycles and with the minimum possible total edge weight, used in network design, clustering, and resource optimisation.
- **Goal**: Parallelize MST algorithm to improve performance for large graphs.
- **Motivation**: As graphs become large and dense, **parallelizing MST algorithms** is essential to improve performance and scale efficiently, enabling faster computation for large-scale, real-world graphs.

Minimum Spanning Trees

- Sequential MST Algorithms:
  - **Prim's Algorithm** $O(E \log V)$: Grows the MST from a starting node by repeatedly adding the smallest edge to an unvisited node.
  - **Kruskal's Algorithm** $O(E \log E)$: Sorts edges and adds the smallest edge that does not form a cycle (uses Union-Find data structure).
  - **Borůvka's Algorithm** $O(E \log V)$: Adds the smallest edge from each connected component, merging them iteratively until all vertices are connected. It is highly parallelizable due to independent component processing.
- Challenges in Sequential Computation:
  - High *time complexity*.
  - Poor scalability with large and dense graphs.

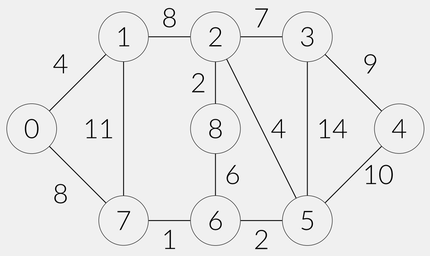

Figure 1. Graph with all edges.

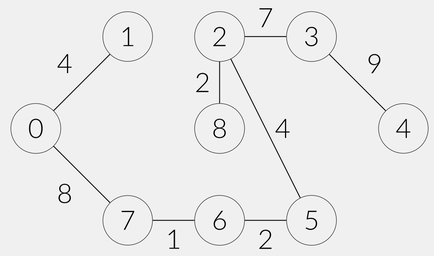

Figure 2. Graph with only MST edges shown.

Problem Statement

Objective: The goal of this project is to design and implement a faster, parallel version of the MST algorithm to compute the Minimum Spanning Tree (MST) more efficiently. This will help speed up the computation and performance, especially for large and complex graphs.

Why Boruvka's Algorithm for Parallelization

- **Borůvka's Algorithm**: Repeatedly selects the minimum weight edge for each component and merges the components, ensuring the tree remains connected. The process continues until only one component remains, which is the MST.
- **Why Boruvka for Parallelization?**: Boruvka's algorithm is inherently parallelizable because at each step, it finds the minimum edge for each connected component. Since these operations are independent for each component, they can be executed in parallel. This makes it more suitable for modern parallel processing architectures.

Parallel Implementation

- This project implements two parallel versions of Borůvka's algorithm using two different approaches: using an edge list and an adjacency list.
- In the parallel implementation, the basic idea is to partition the graph into multiple components and assign each thread the task of finding the minimum edge connecting its assigned component to another. Each thread performs a greedy search to find the closest neighboring component, then stores the minimum edge found. These edges are then merged in parallel by multiple threads, speeding up the process, and the process repeats until only one component remains, which forms the Minimum Spanning Tree (MST).
- Multiple components can be processed simultaneously, significantly reducing execution time.

Edge List Approach

- Stores all graph edges in a list and disjoint sets are maintained to track connected components.
- **Concurrency:** The workload is divided by splitting the edges and assigning each thread a chunk of edges to inspect.
- Each thread handles different portions of the edge list to identify the cheapest edge for each component.
- **Locking:** A mutex is used for each component to ensure that the minimum edge of each component is only updated by one thread at a time, preventing race conditions.
- *This method uses a fine-grained lock (one for each component), instead of a single lock for the entire array. This minimizes the contention between threads, as the minimum found edge for different components can be updated concurrently.*

Adjacency List Approach

- Stores neighboring edges for each vertex and disjoint sets are maintained to track connected components.
- **Concurrency:** The workload is divided by splitting the graph into components (disjoint sets) and assigning each thread a chunk of components to inspect.
- Each thread handles a chunk of the component list. It iterates over the vertices of each component, looks at all the edges connected to these vertices, and selects the minimum edge for that component.
- **Locking:** No need for locks here, as each component is assigned exclusively to a single thread and is not shared across multiple threads.

Common Aspects: Merging Edges in Parallel

- After finding the cheapest edges, it tries to merge the disjoint sets of the two vertices of the cheapest edge.
- **Concurrency**: This is again parallelized by splitting the list of cheapest edges into chunks and assigning each thread a chunk of minimum edges.
- Each thread performs the merge operation on the corresponding edges, trying to merge the sets and update the MST.
- **Locking**: A mutex is used during the merging process, ensuring that only one thread can modify the MST at any given time.

Experimental Results

- **Setup**: Tested on 28-core CPU (physical cores) using C++ with `thread`, sparse/dense graphs (1K–10K vertices).
- **Metrics**: Execution time, Speedup.

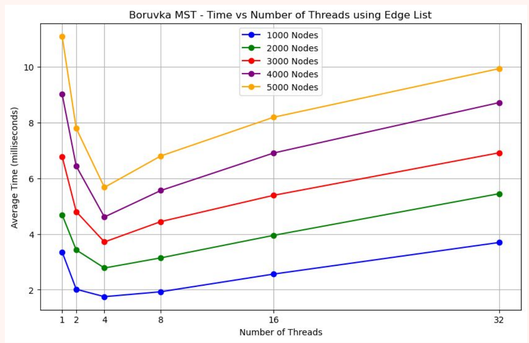

Figure 3. Edge List: Time vs Number of threads (1k-5k Nodes)

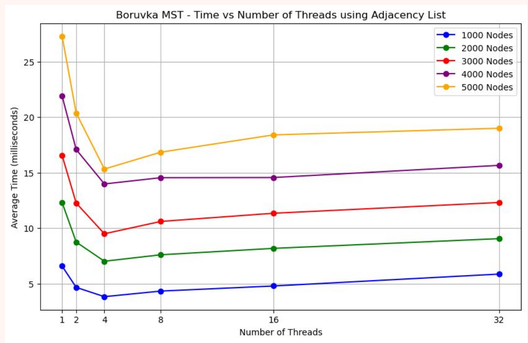

Figure 4. Adjacency List: Time vs Number of threads (1k-5k Nodes)

- As the number of threads increases, the time decreases sharply, reaches an optimal point, and then starts to increase again due to too much synchronization overhead and thread contention. Threads spend more time waiting on each other instead of performing useful work.
- As threads increase further (8, 16, 32), time increases more rapidly in the edge list approach compared to the adjacency list approach. This happens because the threads compete for resources, the time spent managing these threads outweighs the benefits of parallelism, causing the performance to degrade. In contrast, the adjacency list has lower overhead as the load balancing is uneven and depends on the degree of the node.
- Load balancing is uniform in the edge list approach, as it depends on the number of edges. In contrast, it is uneven in the adjacency list approach because it depends on the degree of the nodes.
- Sparse graphs with few edges per node are better suited for the adjacency list approach, while dense graphs with uniform edge distributions are better suited for the edge list approach.

Comparison with Sequential MST

- **Sequential Borůvka's**: $O(E \log V)$ time, single-threaded.
- **Parallel (Edge List & Adjacency List)**: Up to 2x faster for large graphs, same MST weight as sequential.
- **Trade-off**: Synchronization overhead in parallel.

Figure 5. Edge List: Sequential vs Parallel.

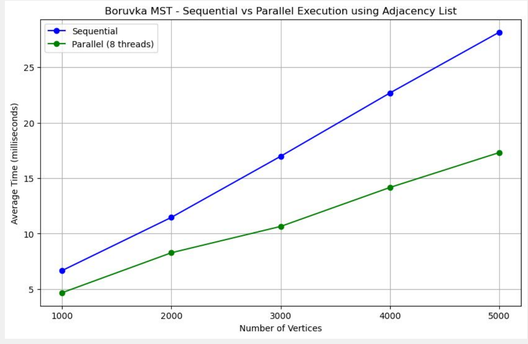

Figure 6. Adjacency List: Sequential vs Parallel.

Algorithm	Time (ms)	Speedup
Sequential	6.9	1x
Parallel	4.45	1.55x

Table 1. Edge List performance (3K vertices).

Algorithm	Time (ms)	Speedup
Sequential	17.6	1x
Parallel	10.4	1.7x

Table 2. Adjacency List performance (3K vertices).

- Parallel execution benefits from the fact that Boruvka's algorithm processes many components independently, enabling different threads to work simultaneously and reduce overall computation time, especially as the number of vertices increases.
- Among the two, using an edge list with parallelism gives the best performance because it has easier data access, better workload sharing, and better synchronization requirements during computation.
- Using an adjacency list has higher overall execution time due to complex neighbor traversal, while the edge list is faster because edge scanning is simpler.

Optimizing the Adjacency List:

- Converting the adjacency list to an edge list significantly reduces MST computation time, as edge lists enable faster edge access and processing.
- We implemented an efficient transformation of the graph representation from an adjacency list to an edge list in O(V + E) time, with results showing computation times as low as 800–3,000 microseconds for graphs with 1,000–5,000 vertices.
- This optimisation enhances the performance of MST algorithms which have an adjacency list as input.

Future Work

- Optimize synchronization with lock-free union-find.
- Extend to GPU-based parallelization for massive graphs.
- Optimize these parallel strategies for better scalability and performance, particularly for large graphs with millions of edges.

Access Github Repository

Mini Project Jan-Apr 2025