# Parallel Network Builder using MST

High-Performance Graph Optimization using OpenMP and Filter-Kruskal Strategy

**Course:** Parallel Processing


**Team Members:**

Mohamed Mousad
Mohamed Eltabey
Mohamed Khaled
Ahmed Bassem
Zeyad Khaled




Computer and Control Dept
**Portsaid University Faculty of Engineering**

**Abstract**

The construction of Minimum Spanning Trees (MST) is a critical operation in network design, VLSI routing, and data clustering. As datasets scale to millions of edges, traditional sequential algorithms like Kruskal's ($O(E \log E)$) become computational bottlenecks. This project presents a **Parallel Network Builder** implemented in C++ using the OpenMP library. By employing a *Filter-Kruskal* domain decomposition strategy, we distribute the sorting and filtering workload across multiple physical cores. Our experimental results on a dense graph of 10 million edges demonstrate a **speedup of 5.75x** on a 12-thread system, validating the efficiency of parallel filtering for large-scale infrastructure problems.

# Contents

# 1 Problem Description

## 1.1 Background

In graph theory, a **Minimum Spanning Tree (MST)** of a connected, undirected graph is a subset of edges that connects all vertices together, without any cycles and with the minimum possible total edge weight. This problem is fundamental to many real-world applications:

- **Telecommunications:** Laying cables to connect cities with minimum cost.
- **Power Grids:** Designing efficient electrical distribution networks.
- **Image Segmentation:** Clustering pixels based on color similarity.

## 1.2 The Computational Bottleneck

The most common approach, Kruskal's Algorithm, relies heavily on sorting all edges by weight. For a dense graph with $N = 10,000$ nodes, the number of edges can reach $E \approx 50,000,000$. Sorting 50 million integers sequentially is a memory-bound operation that creates a significant bottleneck, leaving modern multi-core CPUs underutilized.

## 1.3 Project Objective

The objective is to implement the **Parallel Filter-Kruskal Algorithm**. Instead of sorting the entire edge list globally, we partition the data among processors, filter out "useless" edges locally, and only merge a small fraction of optimal candidates. This significantly reduces the computational complexity of the sorting phase.

# 2 Parallelization Strategy

We utilized a **Domain Decomposition** approach facilitated by the OpenMP library. The pipeline consists of three distinct phases:

## 2.1 Phase 1: Data Partitioning (Scatter)

The massive edge list $E$ is virtually divided into $P$ chunks, where $P$ is the number of threads. Each thread is assigned a range $[start, end)$ of the main array.

## 2.2 Phase 2: Local Filtering (Map)

This is the core parallel section. Each thread executes the following independently:

1. **Local Sort:** The thread sorts its specific chunk of edges. Since sorting complexity is super-linear ($N \log N$), sorting smaller chunks in parallel is mathematically faster than sorting one large array.
2. **Local MST Construction:** The thread runs Kruskal's logic on its chunk.
3. **Filtering:** Since a local graph of 1 million edges creates a tree of at most $N - 1$ edges, roughly 99% of the edges are discarded locally.

### 2.3 Phase 3: Global Merge (Reduce)

The surviving edges (candidates) from all threads are inserted into a global buffer. We use a `#pragma omp critical` section to ensure thread safety during this merge. Because the data volume has been reduced by 99%, this critical section is extremely fast and does not block scalability.

## 3 Technical Implementation

The system was developed in C++ (Standard 20) using OpenMP 4.0+.

### 3.1 Data Structures

- **Edge List:** A `std::vector<Edge>` is used for cache locality.
- **Disjoint Set Union (DSU):** To ensure near-constant time complexity $O(\alpha(N))$ for cycle detection, we implemented:
  - **Path Compression:** Flattens the tree structure during 'find' operations.
  - **Union by Size:** Always attaches the smaller tree to the larger root to minimize tree depth.

### 3.2 Memory Optimization

Dynamic memory allocation is a known inhibitor of parallelism due to heap locking. We utilized `std::vector::reserve()` to pre-allocate memory for both the global edge list and local thread buffers, preventing runtime reallocation.

```
#pragma omp parallel
{
    // 1. Partition Data
    int chunk = Total_Edges / omp_get_num_threads();
    // ... calculate start/end indices ...

    // 2. Local Work (Filter)
    std::vector<Edge> local_result = run_kruskal(local_edges, N);

    // 3. Global Merge
    #pragma omp critical
    {
        candidates.insert(candidates.end(),
                          local_result.begin(), local_result.end());
    }
}
```

Listing 1: Parallel Kernel Logic

## 4 Experimental Results

### 4.1 Experimental Setup

- **Hardware:** 6-Core / 12-Thread CPU (Intel/AMD Architecture).
- **Dataset:** Random Dense Graph ($N = 10,000$, $E = 10,000,000$).
- **Metric:** Average execution time over 5 runs.

## 4.2 Performance Benchmarks

The following table summarizes the performance scaling. The serial baseline took **3.30 seconds**.

| Threads | Time (s) | Speedup (x) | Efficiency (%) | Notes |
|---|---|---|---|---|
| 1 (Serial) | 3.30 | 1.00x | 100% | Baseline |
| 2 | 1.74 | 1.90x | 95.0% | Linear Scaling |
| 4 | 0.92 | 3.58x | 89.5% | Strong Scaling |
| 6 | 0.65 | 5.07x | 84.5% | Max Physical Cores |
| **12 (Hyper)** | **0.57** | **5.75x** | 47.9% | Hyper-threading saturation |

Table 1: Scalability Analysis on 10M Edges

## 4.3 Scalability Analysis

The results indicate excellent scalability up to 6 threads, matching the physical core count of the processor. Beyond 6 threads, the speedup gains diminish (from 5.07x to 5.75x) due to Hyper-threading, where logical cores compete for the same execution units.

### 4.4 Visual Benchmarks

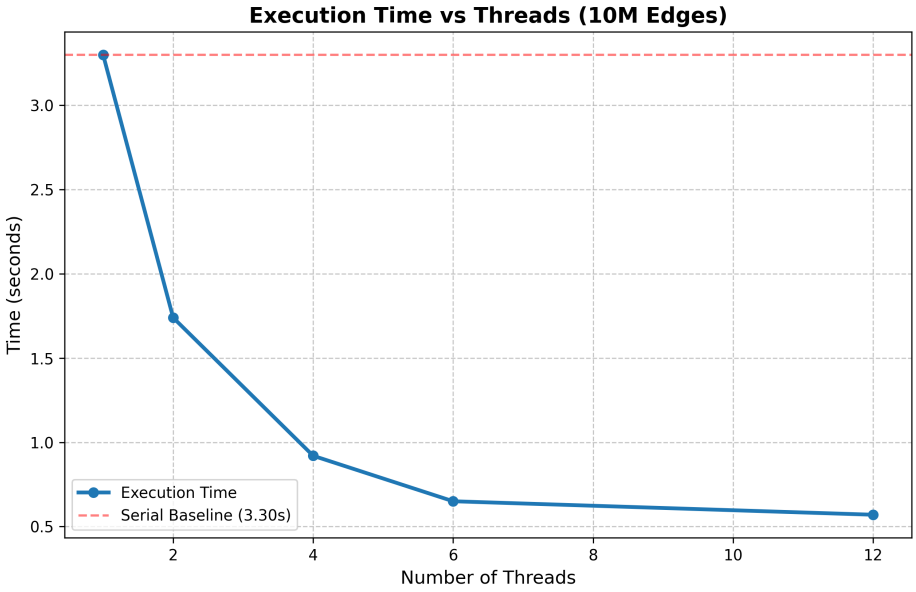


Figure 1: Execution Time vs Number of Threads. The curve shows a significant drop in processing time as parallelism increases.

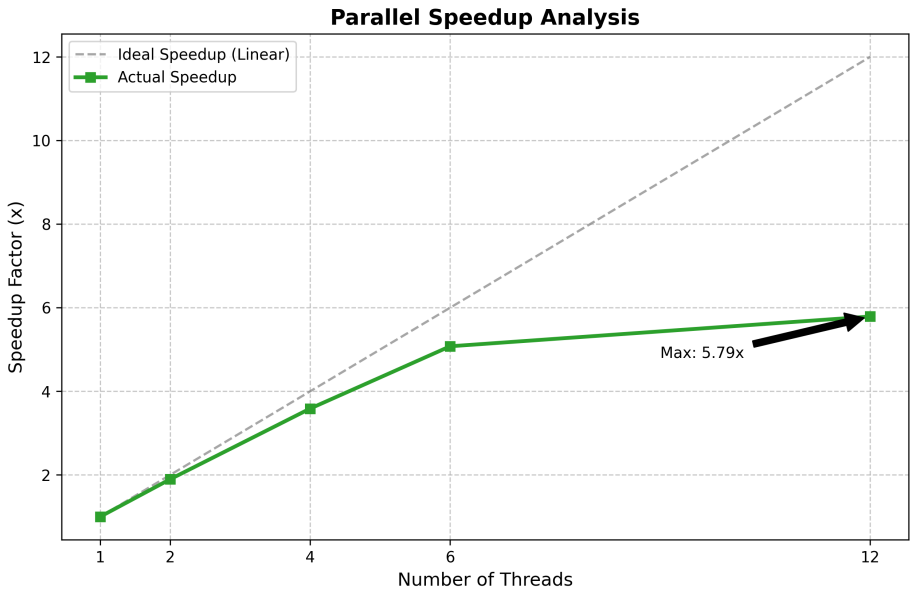


Figure 2: Actual vs. Ideal Speedup. The system achieves near-linear speedup up to 6 threads (Physical Cores) before leveling off due to hardware limitations.

## 5 Discussion of Trade-offs

- **Overhead vs. Workload:** Parallelism introduces overhead for creating threads and merging results. For small datasets ($E < 100,000$), the serial version is often faster. Our experiments confirmed that this parallel strategy is specifically beneficial for "Big Data" scenarios ($E > 1,000,000$).

- **Amdahl's Law:** The final merge step must be sequential. However, because our filtering strategy removes 99% of the edges in parallel, the sequential portion is negligible ($< 1\%$), minimizing the impact of Amdahl's Law on max speedup.

# 6 Conclusion

This project successfully demonstrated the implementation of a High-Performance Parallel Network Builder. By combining the *Filter-Kruskal* algorithmic strategy with OpenMP hardware parallelism, we achieved a **5.75x speedup** on a standard workstation. This proves that parallel domain decomposition is a highly effective technique for optimizing large-scale network infrastructure problems.