Steil, Patrick
patrick.steil@student.kit.edu

Introduction

This report investigates the execution of a parallel Minimum Spanning Tree (MST) algorithm. By leveraging parallelism, our goal is to improve the efficiency of MST computation, allowing the algorithm to adapt seamlessly to the growing size and intricacy of modern graphs. The report offers insights into conventional MST algorithms and outlines the planning and execution of a parallel MST algorithm.

Notation

We denote a *graph* by $G = (V, E)$, representing a tuple comprising a set of vertices V and a set of edges $E \subset V \times V$. We define $|G| \equiv |V|$, as well as $||G|| \equiv |E|$. Additionally, we introduce a *weight function* $\omega : E \to \mathbb{R}$, which associates each edge with a numerical value known as the *weight* of the edge. A *tree* is characterized as a graph devoid of cycles, meaning that between any pair of vertices, there exists at most one simple path. A *forest* is a collection of disjoint trees. In other words, a forest is formed by removing any cycles from a graph. Each connected component of the forest represents a separate tree.When dealing with a specific graph G, the *minimum spanning tree* (MST) problem emerges, seeking a spanning tree —essentially, a tree that encompasses all vertices of G— that minimizes the total sum of edge weights.

Algorithms

The MST problem, solvable in polynomial time, has been a subject of significant research due to its wide-ranging applications in network optimization and connectivity.

Boruvka's Algorithm [2] is an algorithm designed for finding the minimum spanning tree of a graph. The algorithm operates in rounds, where in each round, every connected component in the current forest selects the minimum-weight edge incident to it. This process continues until there is only one connected component left, forming the minimum spanning tree. Boruvka's algorithm is notable for its parallelizable nature, making it well-suited for parallel computing environments.

Kruskal's Algorithm [4] is a classical and widely-used algorithm for finding the minimum spanning tree of a connected, undirected graph. The algorithm follows a greedy approach by iteratively selecting the *lightest* edge, that does not form a cycle with the edges already included in the growing minimum spanning tree. Kruskal's algorithm utilizes a disjoint-set data structure to efficiently check for the presence of cycles during edge selection. Algorithm 1 shows the pseudocode.

Algorithm 1 Kruskal's algorithm

```
1: $T = (V, \emptyset)$                                ▷ implicitly given
2: Sort edges $E$ in non-decreasing order of weight
3: for each edge $(u, v)$ in $E$ do
4:     if $u$ and $v$ are different components of $T$ then
5:         Add $(u, v)$ to $T$
6:         Merge the connected components of $u$ and $v$ in $T$
```

FilterKruskal [5] represents a modification of Kruskal's algorithm, introducing a filtering mechanism during the edge selection process. It addresses the challenge of efficiently handling large graphs by reducing the number of edges considered for inclusion in the minimum spanning tree. Inspired by the partitioning idea in Quicksort [3], FilterKruskal aims to halve the number of edges to be scanned by partitioning them with respect to a pivot element. The algorithm recursively processes the "smaller" edges first, attempting to identify all MST edges without extensive scanning of the "larger" edges. The recursion reaches a base case when the number of remaining edges to be scanned becomes sufficiently small. At this point, Kruskal's algorithm is applied to process these remaining edges. Refer to Algorithm 2 for the pseudocode.

Algorithm 2 FilterKruskal algorithm

```
1: $T = (V, \emptyset)$                                ▷ implicitly given
2: if $|E| \leq$ some threshold then
3:     KRUSKAL'S ALGORITHM($E$, $T$)
4:     return
5: Select a pivot edge $p \in E$
6: Partition $E$ into $E_{\leq}$ and $E_{>}$ with respect to $p$
7: FILTERKRUSKAL ALGORITHM($E_{\leq}$, $T$)
8: if($||T|| = |T| - 1$) return
9: Apply a filter to $E_{>}$
10: FILTERKRUSKAL ALGORITHM($E_{>}$, $T$)
```

Implementation

I implemented the FilterKruskal algorithm, and I began by implementing all sequential algorithms (Kruskal and FilterKruskal).

Sequential. For Kruskal's algorithm, I employed a UnionFind data structure (with path compression and fast union by rank) to manage all components of the growing MST T. Edge sorting was accomplished using the standard `std::sort` algorithm. Notably, in Algorithm 1 (line 3), the loop iterates over every edge. However, it's worth noting that this can be inefficient, as we can stop as soon as we find all $|G| - 1$ edges.

Implementing the FilterKruskal algorithm involved simply translating the pseudocode into valid `C++` code. I set a threshold of $2 \cdot |G|$, so that we use the base case algorithm. The pivot selection utilized the "median of three" method. Partitioning was performed using `std::partition` from the STD library, along with subsequent edge filtering using `std::remove_if`.

To minimize memory reallocation, my implementation performed all operations in place, swapping and overwriting elements in the given edges vector. Two indices (`left` and `right`) were maintained to identify which edges should be considered by each algorithm.

Parallel. Parallelization was easily introduced by enabling all `std` methods to run in parallel using multiple threads, achieved by passing an additional parameter `std::execution::par` to the aforementioned methods.

However, parallel sorting and partitioning steps posed significant bottlenecks. To address this, I adopted the "In-place Parallel Super Scalar Samplesort (IPS4o)" [1] sorting algorithm. This switch substantially reduced running times, particularly when dealing with large graphs on multiple threads. See Figures 1, 2, 3, 4 to see the runtime, speedup, throughput and weak scaling depending on the number of threads used.

Figure 1: Runtime of my implementation.

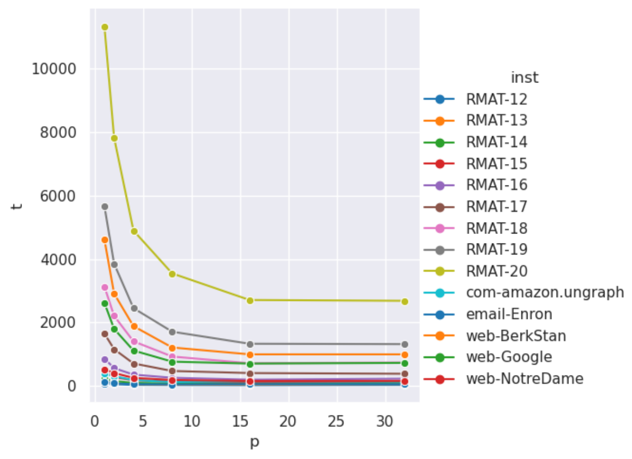


Figure 2: Speedup of my implementation.

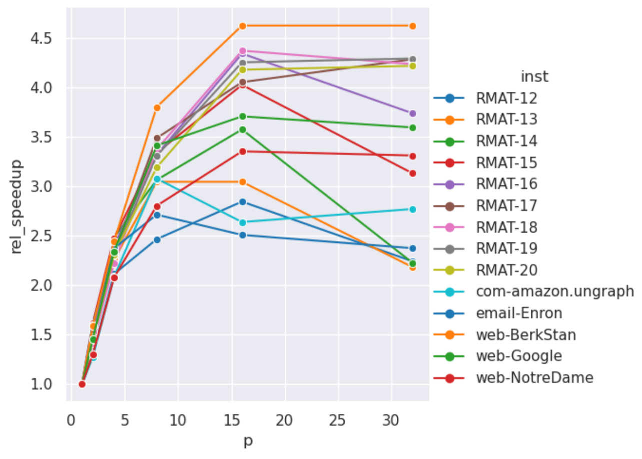
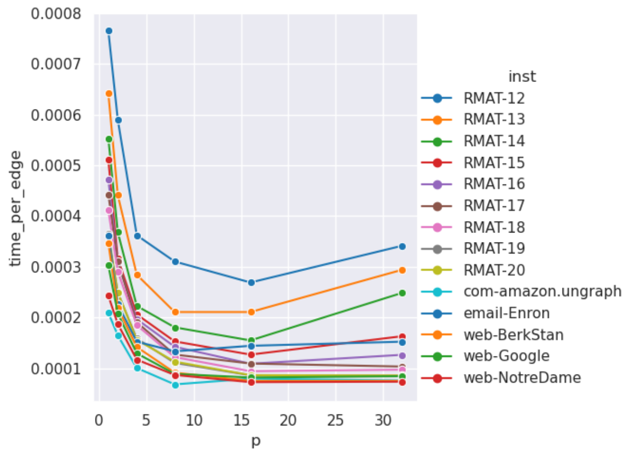


Figure 3: Throughput of my implementation.



References

[1] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.ESA.2017.9`.

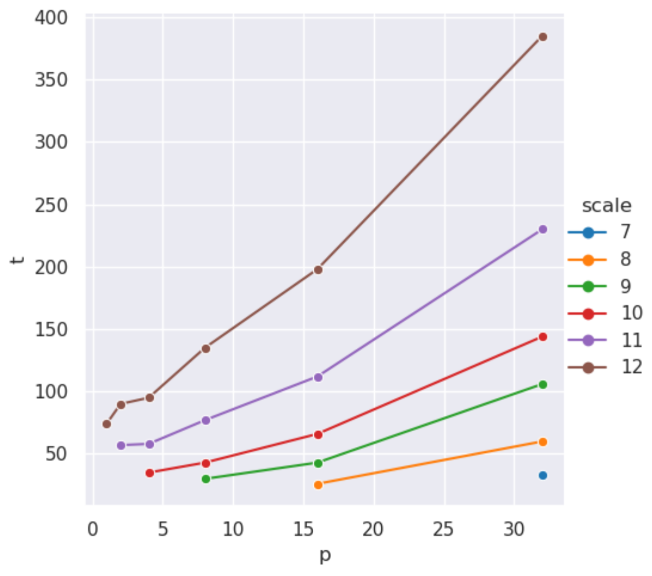
[2] Otakar Boruvka. O jistém problému minimálním. 1926.

[3] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, jul 1961. `doi:10.1145/366622.366644`.

[4] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[5] Vitaly Osipov, Peter Sanders, and Johannes Singler. *The Filter-Kruskal Minimum Spanning Tree Algorithm*, pages 52–61. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9781611972894.5`, `arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611972894.5`, `doi:10.1137/1.9781611972894.5`.

Figure 4: Weak scaling of my implementation.


t
p
scale