DISTRIBUTED LARGE MINIMUM SPANNING TREE COMPUTATION

Abhimanyu Bhadauria, Nodar Ambroladze, Philipp Schaad

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this paper we implement and compare four distributed algorithms for solving the Minimum Spanning Tree (MST) problem: edge partitioning, vertex partitioning and two versions of parallel prim. The main use scenarios for those algorithms are cases where the entirety of the graph does not fit into the main memory of a single machine. The basic assumption for all of them is, that just the graph vertices fit into the memory of each individual compute node. The idea thus is to scale linearly in problem size with the number of added distributed compute nodes running the algorithms.

1. INTRODUCTION

Minimum Spanning Tree (MST) problems can be found in abundance in practical applications like the construction of a cable network (that typically follows the road-network), modeling and understanding social connections on platforms like Facebook™, or dealing with complex web-graphs. As a result of that, many attempts have been made at making MST algorithms faster by parallelizing them, even though it is notoriously hard to gain any significant speedup by parallelizing the MST problem [1]. Numerous algorithms have also been proposed that try to exploit certain patterns / structures or properties of graphs (eg. density) [2, 3]. However MST problems suffer a secondary downside in that real world graphs can sometimes be huge in size. This can mean that graphs may not fit into the main memory of a single machine anymore.

With this work we try to address that problem by proposing a set of distributed algorithms that can work on finding an MST for a large graph on a distributed memory cluster by partitioning the graphs between individual nodes. More specifically we present three algorithms inspired by theoretical analyses and proposals made by Ramaswamy et al. [4].

There already exist a number of distributed MST algorithms [5, 6], but they typically try to address a different problem, like for example individual nodes being unaware of the entire graph topology. They typically still require at least most of the total graph to be present on one node at some time, which makes them unsuitable for huge graphs where memory can be an issue.

2. BACKGROUND

In any undirected graph G, a **spanning tree** is a subgraph that connects all the vertices of G with the minimum possible number of edges, thus forming a tree. An unconnected graph U cannot have a spanning tree, but each connected component of U has one, and all the connected component's spanning trees together form the **spanning forest** of U. A graph can have many different possible spanning trees or forests. But in a weighted graph we can have a **Minimum Spanning Tree**, which is the spanning tree where the sum of edge weights is minimal. The **Minimum Spanning Forest** forms the unconnected equivalent. Figure 1 shows an example of a simple MST.

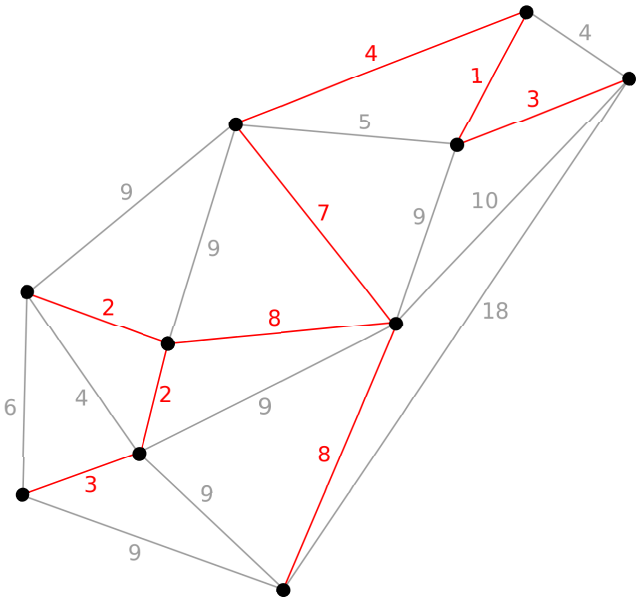

Fig. 1. An example of an MST highlighted in red

Listing 1. Pseudo-Code for the edge partitioning algorithm

```
1  input: Graph G, output: MST M
2  id = rank(); n = n_processes();
3  E = read_every_nth_edge(G, id);
4  V = get_vertices(G);
5  while (true) {
6    M = kruskal_MST(E, V);
7    broadcast(|M|);
8    global_total = combine_broadcasts();
9    if (global_total + |V| < MEM_SIZE) {
10     if (rank == 0) { // root
11       E = gather_all_edges();
12       M = kruskal_MST(E, V);
13       return;
14     } else {
15       send_to_root(M);
16       return;
17     }
18   } else {
19     split_and_distribute(M, n);
20     E = receive_new_edges();
21   }
22 }
```

Listing 2. Pseudo-Code for the vertex partitioning algorithm

```
1  input: Graph G, output: MST M
2  id = rank(); n = n_processes();
3  V_local = [ id · V/n , (id+1) · V/n ]
4  E = read_incident_edges(V_local);
5  M = {∅}
6  for (int i = 0; i < |V| − 1; i++) {
7    e = get_min_edge(E, M);
8    send_to_root(e);
9    if (rank == 0) {
10       e = min(gather_all_edges());
11       M = M ∪ {e}
12       broadcast(e);
13   } else {
14     e = recv_from_root();
15     M = M ∪ {e}
16   }
17 }
```

3. ALGORITHMS

3.1. Edge Partitioning

The edge partitioning algorithm proposed by Ramaswamy et al. [4] works splitting up a graph with $|V|$ vertices and $|E|$ edges on to n processes. Each process reads all $|V|$ vertices and exactly $\frac{|E|}{n}$ edges, before locally computing the MST on the graph induced by that subset of edges. After that, all edges of the local MSTs are gathered on the root process. Ramaswamy et al. [4] state that they now check if all the edges of those combined local MSTs fit into the memory. If so, they compute the final global MST, but redistribute the edges randomly again if they do not yet fit. Such a local MST will have at most $|V| - 1$ edges, meaning that to perform the gathering step on the root process (assuming all machines / nodes are equal in terms of main memory size, and one vertex as well as one edge consume one 'unit of memory') the theoretical minimum memory requirement is $\mathcal{O}(n * |V|))$ per machine. This is independent of whether we redistribute afterwards or not, since the algorithm's design lets the root node alone decide on and perform the redistribution, thus requiring all the local MSTs to fit into its memory after one iteration anyway.

Obviously this still is a very harsh memory constraint, which can be softened at the cost of additional iterations and thus runtime. We do this by changing the redistribution step. After locally computing the MST, each process broadcasts the size of its MST and thus the number of edges it wants to redistribute. Every process then listens for the broadcast of the other ' processes, locally adds up the results to calculate how many edges are remaining in total, and then checks if those (plus the vertices) fit into the memory of the root node. If so, all edges are sent to the root, which computes the final MST. If not, each process splits its set of remaining edges into n equal parts (where n is the number of processes), and then sends those to their designated slave nodes while listening for its own new batches of edges. We then continue to iterate until we can combine on the root node. A pseudocode outline of this procedure is given in Listing 3.

This implies that the theoretical lower bound in memory for our algorithm would be $\mathcal{O}(|V| + 3)$, because we still need to fit all vertices, plus at least two edges. A minimum of 3 edges is needed to make any progress at all in a graph with $|V| > 2$. Of course, the smaller the memory and thus the closer to this lower bound, the more iterations the algorithm will have to perform.

3.2. Vertex Partitioning

We based our vertex partitioning algorithm on an algorithm outlined by Lonar et al. [7], and it works in a very similar way to the sequential prim algorithm: first partition the vertices into n subsets. On each iteration, every process p_i finds the minimum-weighted edge e_i connecting the MST to a vertex in v_i, the partition that was allocated to the processor i, and then sends e_i to the root process. The root process then computes the global minimum of all such edges it

Listing 3. Pseudo-Code for the parallel prim algorithm

```
1  input: Graph G, output: MST M
2  id = rank(); n = n_processes();
3  E = read_every_nth_edge(G, id);
4  V = get_vertices(G);
5  A = Disjoint_Set();
6  for all v ∈ V {
7    A.add_set({v});
8  }
9  M = {∅}
10 while |M| < |V| − 1 {
11   // for each disjoint set calculate
12   // minimum outgoing edge for each
13   // disjoint set
14   E_outgoing = get_min_outgoing_edges();
15   send_to_root(E_outgoing);
16   if (rank == 0) {
17     E_recv = gather_all_outgoing();
18     E_min_edges = get_glob_min_outgoing(E_recv);
19     for e ∈ E_min_edges {
20       M = M ∪ {e};
21       A.Union_Set(e.from, e.to);
22     }
23     Broadcast(E_min_edges);
24   } else {
25     E_min_edges = recv_from_root();
26     for e ∈ E_min_edges {
27       M = M ∪ {e};
28       A.Union_Set(e.from, e.to);
29     }
30   }
31
32 }
```

receives and adds it to the MST before broadcasting the resulting MST. After $|V|-1$ iterations we get the correct MST on each process. The total communication cost is therefore $O(V * n)$ and it's easy to see that this algorithm does not exploit bandwidth very well, and the latency will become a significant bottleneck (especially for sparse graphs), because on each super-step, each process sends only a single edge.

3.3. Parallel Prim

In this algorithm we split the graph exactly the same way as we do in the edge partitioning algorithm. Here, each process maintains disjoint sets of vertices, where in each set the vertices are connected by the edges of the MST that is being constructed: once we decide to add an edge to the MST, we union the two sets that contain vertices incident to this edge.

On each iteration, every process calculates and reports the minimum outgoing edge from each disjoint set of vertices to the root process. After collecting all the reports, the root process computes the global minimum outgoing edges from each disjoint set, adds these edges to the MST and broadcasts said new MST. It's easy to see that algorithm terminates after $O(logV)$ iteration because on each iteration, the number of disjoint sets gets at least halved. The communication cost is therefore

$$n \cdot (\frac{|V|}{2} + \frac{|V|}{4} + \cdots + \frac{|V|}{2^t}) = O(n \cdot V)$$

We implemented two versions of this algorithm. One of which works better for sparse graphs, and the other one is optimized for dense graphs.

4. EVALUATION

Our algorithms try to provide a solution for graphs that do not fit into the main memory of a single compute node. Due to time constraints and how tedious setting up a multi-node compute cluster can be, we simulated this scenario by using a single machine. To achieve accurate simulations we implemented a hard memory limit for our algorithms. Each additional node gets simulated by starting a new process running our algorithm, and each process has a fixed maximum amount of memory it can consume. Should at any point in the computation a node try to consume more memory, the computation fails. This is also true for our sequential algorithms, meaning that if we have a memory limit of 1'000 units (where a unit can be a vertex or an edge) and we have a graph with 500 vertices and 700 edges, the computation with a sequential algorithm will abort.

We implemented fast versions of the most commonly used sequential MST computation algorithms (Kruskal and Prim) and used those as a baseline. We also compared their runtime with well known and widely used implementations like the ones found in the Boost Graph Library (BGL) [8, 9, 10].

As we anticipated, the vertex partitioning algorithm performs very poorly compared to the other parallel algorithms, because of inefficient use of bandwidth.

Figure 3 shows the scaling factor of our algorithms: given a hard limit of 1M memory units on each machine, we are able to solve for a problem of size $|E| = 0.25M$ with a single process, $|E| = 2.5M$ with 4 processes, and $|E| = 25M$ with 64 processes (note that number of processes are not optimal, we rounded it up to closes power of two, because some of our algorithms benefit from having a communication group in a power of two size).

Figure 4 shows the runtimes of our algorithms on a graph depicting the US road network [11]. As expected, the sparse implementation of parallel prim indeed works better on this

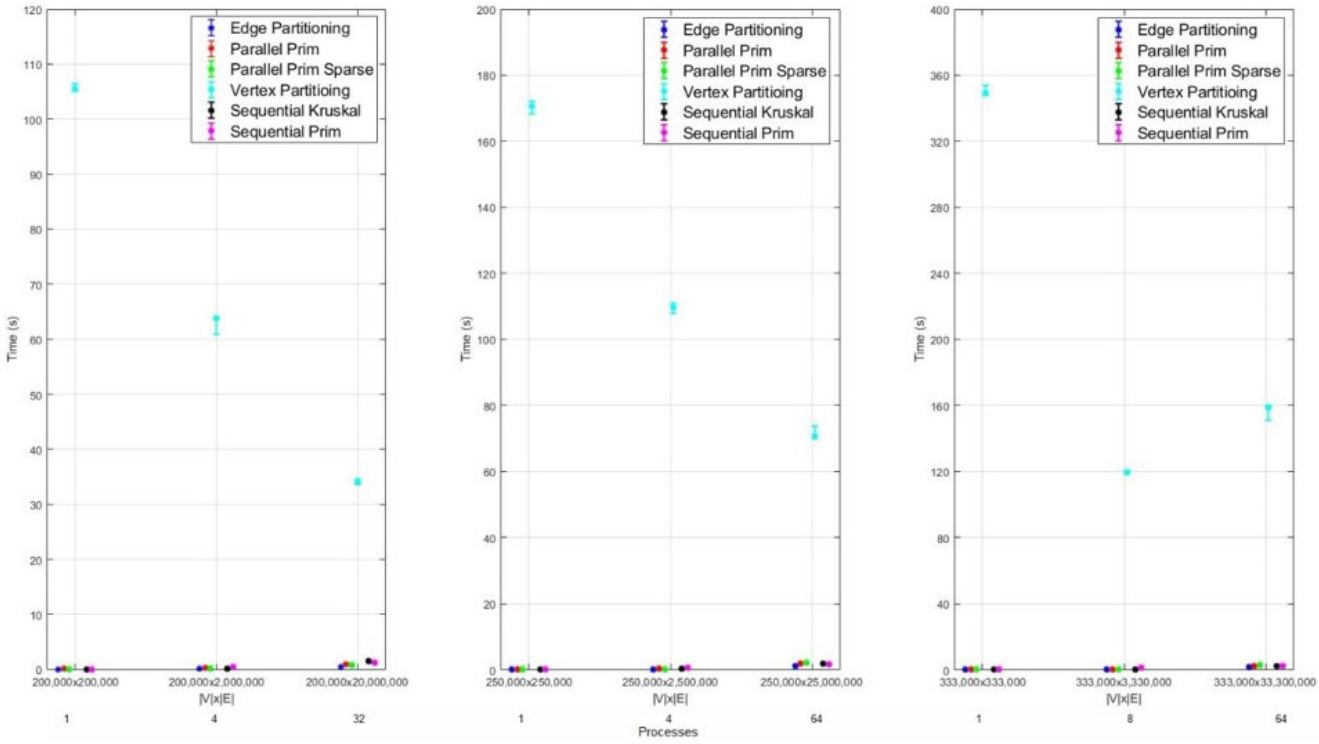

Fig. 2. Performance comparison of all algorithms

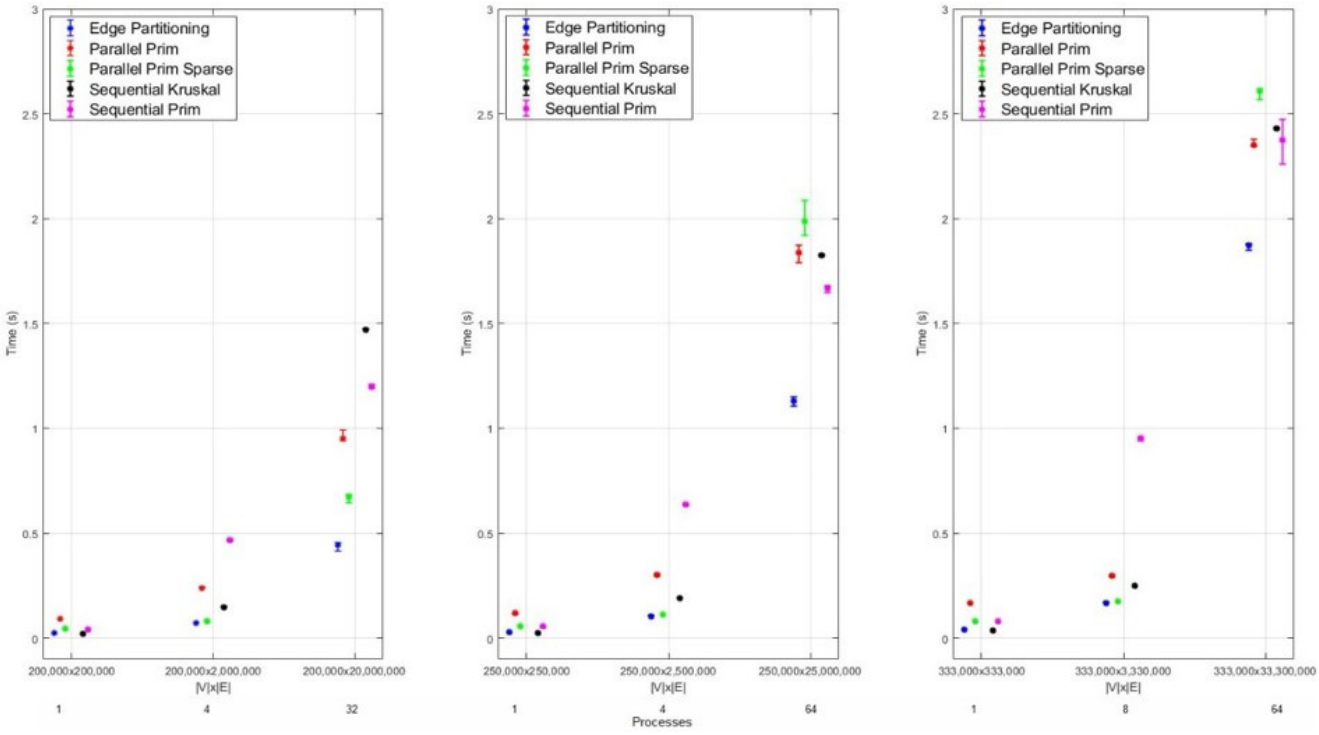

Fig. 3. Closer look at performance excluding vertex partitioning

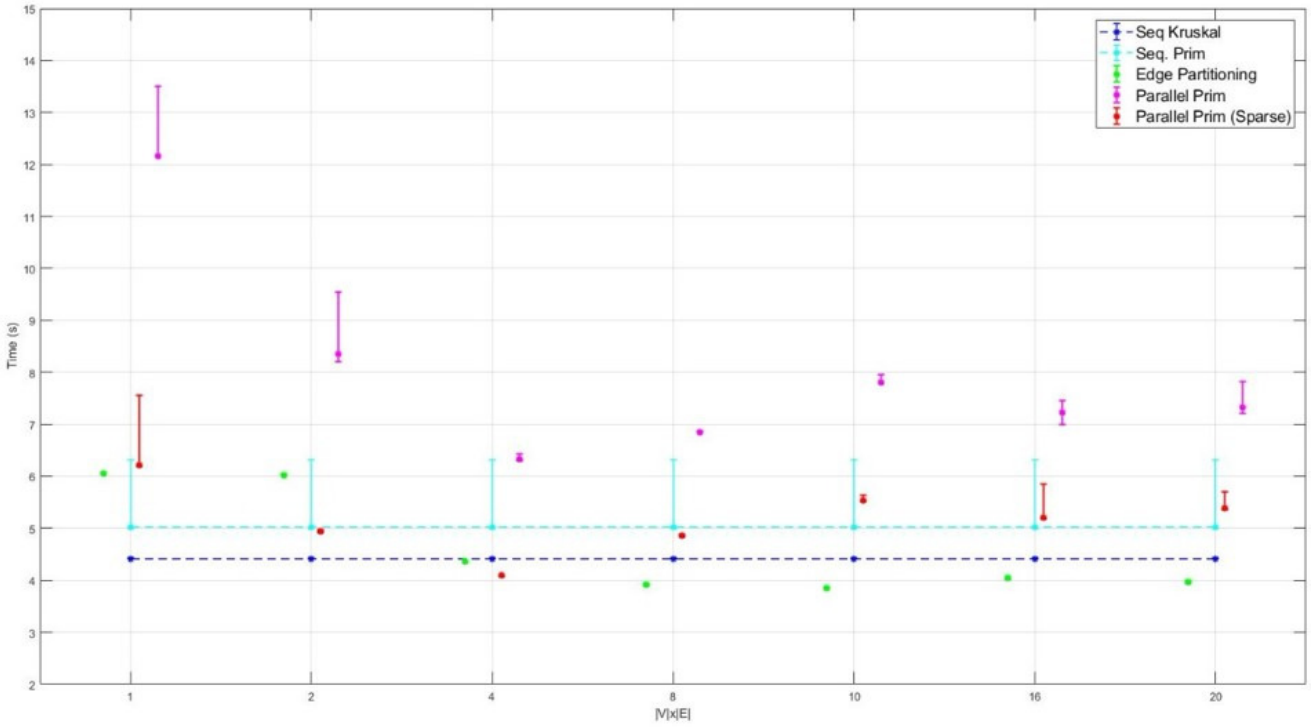

Fig. 4. Algorithm runtimes on US road network

graph than the density optimized version of it. Edge partitioning even works faster than the sequential versions.

5. CONCLUSION

With our goal being scaling in problem size, we don't get any (significant) speedup for the same problem size by adding more processes, but we're able to solve bigger problems and scale linearly in that regard. For a fixed problem size there is a "sweet spot" in the number of processes that solve it optimally. That number depends on the memory constraint for each machine. However, even for the graphs that do fit in one machine, we don't perform any slower than sequential versions.

The fastest of the described algorithms is edge partitioning, which works equally well on sparse and dense graphs, which was expected, as it has better asymptotic complexity both in runtime, and in communication cost compared to the other algorithms. The slowest algorithm on the other hand is vertex partitioning, mostly because of the synchronization overhead and for not using bandwidth very inefficiently.

6. REFERENCES

[1] David A. Bader and Guojing Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps)," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 994 – 1006, 2005.

[2] David A. Bader and Guojing Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," *Journal of Parallel and Distributed Computing*, vol. 66, no. 11, pp. 1366 – 1378, 2006.

[3] F. Dehne and S. Gotz, "Practical parallel algorithms for minimum spanning trees," in *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No.98CB36281)*, Oct 1998, pp. 366–371.

[4] Swaroop I. Ramaswamy and Rohit Patki, "Distributed minimum spanning trees," 2015.

[5] Maleq Khan, Gopal Pandurangan, and VS Anil Kumar, "Distributed algorithms for constructing approximate minimum spanning trees in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 1, pp. 124–139, 2009.

[6] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and systems (TOPLAS)*, vol. 5, no. 1, pp. 66–77, 1983.

[7] Vladmir Loncar, Srdjan Škrbic, and Antun Balaz, "Distributed memory parallel algorithms for minimum spanning trees," in *Proceedings of the World Congress on Engineering*, 2013, vol. 2, pp. 3–5.

[8] "Bgl prim minimum spanning tree," `http://www.boost.org/doc/libs/1_53_0/libs/graph/doc/kruskal_min_spanning_tree.html`, 2017.

[9] "Bgl kruskal minimum spanning tree," `http://www.boost.org/doc/libs/1_40_0/libs/graph/doc/prim_minimum_spanning_tree.html`, 2017.

[10] Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine, et al., *Boost graph library*, Pearson India, 2002.

[11] "Road network of the usa with distances," `http://www.dis.uniroma1.it/challenge9/download.shtml`.