EFFICIENT PARALLELIZATION OF BORŮVKA'S MINIMUM SPANNING TREE ALGORITHM

Samuel Anzalone
BS Computational Science
ansamuel@ethz.ch

Mike Marti
MS Computer Science
mikmarti@ethz.ch

Matteo Kamm
MS Computer Science
matkamm@ethz.ch

Hulda L. Hannesdóttir
MS Computer Science
hhannesdo@ethz.ch

Šimon Hrabec
MS Computer Science
shrabec@ethz.ch

ABSTRACT

Minimum Spanning Tree algorithms are among the most well-known graph algorithms in computer science. They can be used as part of a large variety of algorithms to compute intermediate results and are applied in numerous scientific fields. As such, having implementations that can handle large inputs and compute the result in a reasonable amount of time is of utmost importance. In this paper, we look at a variety of work distribution and merging strategies that can be used to implement Borůvka's MST algorithm. We developed some implementations using a mixture of these strategies. To measure and compare the performance of these implementations, we performed benchmarks on the RACKlette cluster and used the Parallel Boost Graph Library as a baseline. Using these strategies, we were able to develop implementations that are up to 32.6 times faster compared to the baseline.

1. INTRODUCTION

In this section we briefly introduce and explain what Minimum-Spanning-Tree algorithms are used for. In addition, we describe our contribution to the scientific community as well as work related to ours.

Formally, an MST of a given undirected connected graph $G = (V, E)$ with vertices $V = \{0, \dots, n-1\}$ and weighted edges $E \subseteq V \times V$, can be defined as an ayclic subgraph of G which connnects all vertices in V with the least total weight.

Motivation. The MST problem is one of the most studied problems in combinatorial optimization [1]. Although the problem is rather simple, its solutions are often used as part of other algorithms to compute intermediate results. Other scientific fields, such as epidemiology or taxonomy, apply MST algorithms as can be seen in the following list of example problems.

- **Networking** MST algorithms find trees in computer networks that can be used for broadcasting without loops [2].
- **$\frac{3}{2}$-approximate metric TSP** By combining algorithms that find MSTs, matchings and eulerian circuits, one can develop a $\frac{3}{2}$-approximation algorithm solving the metric traveling salesperson problem [3].
- **Molecular Epidemiology** Minimum-Spanning-Trees are used in molecular epidemiology research to estimate relationships among individual strains or isolates [4, 5].
- **Machine Learning** MSTs are used as part of machine learning algorithms. For example, MSTs can reduce the fraction of incorrectly labeled samples when performing brain MRI tissue classification [6].

For most of these use cases, the speed of the MST algorithm is of great importance e.g. to get quick medical results and be able to treat the patient accordingly.

Contribution. In our research, we focus on the MST algorithm proposed by Borůvka [7, 8], which is one of the most prominent MST algorithms as of today. We implement this algorithm using different work distribution and merging strategies. Our implementations use the Message Passing Interface (MPI) for communication. Randomly generated Kronecker graphs are used to evaluate the performance of our implementation. In this paper we present a parallel MST implementation that performs significantly better than one state of the art implementation.

Related Work. As the MST problem is very versatile and can be used in various scientific disciplines, there already is some research on parallelizing MST algorithms. One such paper [9], which is also the main inspiration of this project, evaluates how the performance of Borůvka behaves when using different pointer jumping schemes. They also show that in principle a speedup proportional to the number of processors can be achieved. Other research limits itself on sufficiently dense graphs and presents an algorithm for the bulk synchronous parallel (BSP) model [10]. Furthermore, a parallelization of the MST algorithm using GPUs is presented in [11]. Some implementations that can be found make use of wait-free Union Find data structures [12]. Such data structures speed up the merge step of the Borůvka algorithm. However, they do not aid in

Algorithm 1 Borůvka's algorithm

```
T ← {}
while ∃ more than one connected component do
    for each component c in T do
        e ← FindLightestOutgoingEdge(c)
        T ← T ∪ e
    end for
    MergeComponents
end while
return T
```

distributing the work over multiple computing nodes.

Our research is similar to that of Chung and Condon [9], but differs in the work distribution strategies applied. We do not, however, use the wait-free implementations of those data structures as described in [12].

2. BACKGROUND

In this section we formally introduce the MST problem, as well as Borůvka's algorithm and different parallelization strategies used in our implementations.

Spanning Tree/Forest. A spanning tree (forest) is an acyclic (disconnected) subgraph $T = (V_T, E_T)$ of an undirected graph $G = (V_G, E_G)$, where $V_T = V_G$ and $E_T \subseteq E_G$. It is easy to see, that a spanning tree for a graph G is not necessarily unique, i.e. there can be more than one spanning tree of a graph. For instance, the fully connected graph K_3 has three spanning trees. For simplicity, we will use the term spanning tree throughout this report even though we might be working with spanning forests in case G is not connected.

Minimum-Spanning-Tree Problem. In the MST problem, every edge e of the input graph $G = (V, E)$ has an associated weight. Formally, there is a weight function $w : E \to \mathbb{N}$ that is given as input. The goal is to find a spanning tree with a minimum edge weight sum, i.e. the edge weight of all other spanning trees is at least as large. Just like with a spanning tree, an MST must not necessarily be unique for a given graph G.

Borůvka's Algorithm. Borůvka's algorithm [7, 8] solves the MST problem as described above. Algorithm 1 contains a high-level overview of how the algorithm operates. The algorithm terminates as soon as T is a spanning tree. In case the input graph is not connected, the termination condition has to be slightly adapted to arrive at a forest. The sequential runtime is $\mathcal{O}(m \log n)$, where $m = |E|$ and $n = |V|$ of the input graph $G = (V, E)$.

Union Find Data Structure. A Union Find data structure stores a partition of a set into disjoint subsets in such a way, that finding the corresponding set of an element and merging two sets takes $\mathcal{O}(\alpha(n))$ time [13]. These operations can be used to describe the connected components in the algorithm 1. Note that the runtime achieved here is not on a per-operation basis. Single operations can take longer but the data structure adjusts itself so that successive operations are faster.

Pointer Jumping. Pointer jumping [14] (also referred to as path doubling) is a design technique that allows an algorithm to follow a path using only logarithmic time with respect to the length of the longest path. It does this by redirecting the parent pointers that describe the path, to point to the parent of the parent for each vertex on the path simultaneously. This process is then repeated until every vertex points to the root of the path. This gives a $\mathcal{O}(n \log n)$ runtime, where n describes the amount of vertices on the path. This is superior to simply iteratively fixing each vertex individually, which has a potential worst case runtime of $\mathcal{O}(n^2)$.

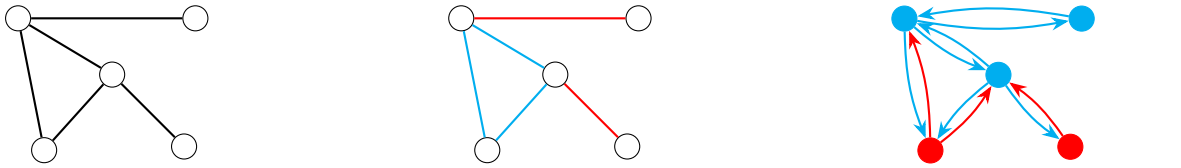
Supervertex Pointer Jumping. Supervertex pointer jumping [9] is an extension of pointer jumping that uses randomness to achieve an expected linear time algorithm. The technique is best explained in the aforementioned paper, but the high level idea is to select a set of vertices to be supervertices, perform pointer jumping on all non-supervertices, until they reach a supervertex or a root, let each supervertex point to the next supervertex and repeat the process recursively on the supervertices. After the recursion, it is sufficient to perform only one more pointer jump for all non-supervertices.

MPI. The message passing interface [15] is a standard for developing applications on parallel computing architectures. It defines library routines for intercore and inter-system communication. Similar to the C++ standard, there is no reference implementation but multiple open source projects implementing the standard, such as MPICH and Open MPI [16].

Kronecker Graphs. Kronecker graphs are a class of graphs constructed from a small base graph by iteratively applying the Kronecker product [17]. The Graph500, a rating of supercomputer systems focusing on graph algorithms, uses a variation of this graph construction for their benchmarks [18].

3. APPROACH

This section contains information about the different implementations developed, as well as other technical details, such as distribution strategies and limitations.



(a) Original input graph (b) Edge distributed input graph (c) Vertex distributed input graph

Fig. 1: Graph distribution example

Distributing the Work. There are many ways how a graph can be distributed among multiple computation units. The two strategies that we considered are distributing the edges and distributing the vertices. A combination of the two would also be conceivable, but was not considered in this project. As these two strategies have different parallelization capabilities, multiple implementations for both of these approaches were developed.

Distribution by Edges. In this approach, the edges of the input graph are distributed evenly among computation units. An illustration of such a distribution can be seen in Fig. 1b. This strategy allows for a uniform workload distribution, as every computation unit has the same amount of edges to process when selecting the minimum weight outgoing edges. On the other hand, the merging of components cannot be parallelized using this approach. This is because during the merging step, all vertices of a component have to unanimously agree on a new leader. But since a vertex is not owned by a computation unit, this cannot be accomplished without additional effort.

Distribution by Vertices. In this approach, the vertices of the input graph are distributed (including their incident edges) among computation units. An example of such a distribution can be seen in Fig. 1c. One major disadvantage of this approach is that each edge gets distributed twice because it is not guaranteed that neighboring vertices are assigned to the same computation unit. Consequently, this strategy requires more time to distribute the graph. Moreover, the workload of each computation unit is unevenly distributed compared to the edge distribution strategy. Imagine a scenario, where certain vertices have a higher degree (amount of incident edges) than others. In that case the computation of the minimum weight outgoing edge of each component takes longer for some computation units whilst others are idle. However, with this approach the merging of components can be parallelized.

Implementations. For this project we developed various C++ implementations, combining different merging approaches with the two aforementioned distribution strategies. The following list contains all the combinations realized:

- Edge Distributed, merging via
  - Union Find
  - Union Find with reduced edges
  - Pointer Jumping
- Vertex Distributed, merging via
  - Union Find
  - Iterative Vertex Fixing
  - Pointer Jumping
  - Supervertex Pointer Jumping

Clearly, this list does not contain every possible combination. Intermediate benchmarking showed that some strategies were not promising enough to be pursued further.

We are not going to give detailed descriptions for all implementations, as their rough idea should be clear with the information given in section 2. The ones that might not be self-explanatory are *Edge Distributed Union Find with reduced edges* and *Vertex Distributed Iterative Vertex Fixing*. The first one uses an additional optimization where edges with both endpoints in the same component get skipped over. As this removal costs additional runtime, we have to be cautious about the impact on performance. The second one uses an iterative fixing of the vertices, briefly mentioned in the pointer jumping paragraph in section 2. This approach has a greater theoretical runtime, but uses far less communication compared to the vertex distributed pointer jumping implementation. The open question here is, whether the lower runtime guarantee is worth the additional communication. This is answered in section 4.3. In addition, the Union Find implementations use the so-called path compression strategy when performing `find()` operations and we merge sets by rank during `union()` calls [19].

Baseline Implementation. To get an idea of how well our implementations perform, we decided to use the Parallel Boost Graph Library (BGL) [20] as our baseline. We considered other baseline implementations but came to the conclusion that the Parallel BGL is the most optimized and well-known library. One promising implementation uses OpenMP which might have led to incomparable results since we use MPI [21].

Kronecker Graph Generator. Initially, we experimented with different kinds of graphs and implemented some generators for them, but ultimately decided to only benchmark Kronecker graphs. Self-written generators can be error-prone and it is difficult to efficiently generate large graphs. Therefore, we decided to use the reference implementation of the Graph500 specification. An alternative to this is the Erdös-Renyi graph generator that is part of the Parallel BGL [20]. To get reproducible benchmark results across all algorithm executions, we seeded the graph generator, so that it always generates the same random graph.

Communication. All implementations use MPI as a communication protocol. Besides send and receive routines used to distribute the graph, we also use scatter, gather and reduction functionalities. To minimize the cost of communication as much as possible, we keep the amount of MPI subroutine calls to a minimum. We achieve this by first locally preparing the data and then sending it in bulk. This is used to e.g. more efficiently distribute the incident edges in all vertex distributed implementations.

Limitations. So far all implementations don't compute the actual MST, but only its edge weight sum. Moreover, the work has to be distributed among all computation units evenly, i.e. the amount of edges or vertices must be divisible by the amount of computation units depending on the distribution strategy. For most implementations, both of these limitations are straightforward to resolve. This is especially true for the MST tree computation, as the lightest edges are distributed among the computation units anyway.

Correctness. To verify the correctness of our algorithms, we perform unit tests using Catch2. On top of that, we perform differential tests with large Kronecker graphs that compare our implementations to the baseline (parallel BGL). To get assurance during development, we use continuous integration that executes the tests and reports potential failure.

4. EXPERIMENTAL RESULTS

In this section we go over the benchmarking setup, as well as the results that we measured, including their interpretation.

4.1. Experimental Setup

The benchmarks were executed on the RACKlette cluster [22], setup and maintained by a group of students from ETH.

Hardware. The RACKlette cluster consists of four identical nodes, each of which has two 64-core AMD EPYC 7742 CPUs running at 2.25GHz with 512GB DDR4 3200MT/s RAM and four Nvidia Tesla V100 GPUs with 32GB GDDR5 VRAM for a total of 512 cores, 2TB of memory and 16 GPUs. The four nodes are connected using a speedy interconnect from Mellanox, namely HDR ConnectX-6 adapters for a 200Gbit/s connection.

Work Allocation. All four cluster nodes were used for the benchmarks. We pinned the CPU cores with the purpose that the amount of used CPU cores was equal on all nodes. In addition, the cores were chosen such that the amount of cache overlap is minimized. This allocation strategy should lead to more consistent measurements, as otherwise benchmarks with a low MPI core count could have an unfair advantage due to less network communication.

Software. All benchmarks were conducted using CentOS Linux release 7.9.2009 (Core) with Linux kernel 3.10.0, the MPI implementation mvapich2 version 2.3.4 and Parallel BGL version 1.78.0. The code was compiled with GCC version 10.3.0 using the compilation flags `-mavx -march=native -O3`. Our results are obtained using the performance measurement tool LIKWID [23]. LIKWID uses hardware counters for its measurements and can be invoked by simply marking the code to be benchmarked.

4.2. Performed Benchmarks

As described in section 3, all benchmarks were performed using Kronecker graphs. These graphs have 16 times more edges than vertices (as per Graph500 specification [18]). This is to ensure an average degree of 32. The implementation *Edge Distributed Pointer Jumping* resulted in bad performance. Since we are not certain which part of the algorithm causes this, we decided to not include it in our reported measurements.

Baseline Comparison. To compare our implementations with the baseline, we performed a strong scaling comparison using graphs with 2^{19} vertices. The number of MPI processes were the powers of 2: $2^0, \ldots, 2^6$. The baseline implementation did not allow us to go any further than that because the execution took too long. We believe this to be an implementation issue within the parallel BGL.

Implementation Comparison. To compare our implementations with each other, we performed a strong and weak scaling comparison. The strong scaling comparison was performed using graphs with 2^{20} vertices. The number of MPI processes were the powers of 2: $2^0, \ldots, 2^9$. The weak scaling comparison was performed using the configurations seen on the x-axis of Fig. 4.

Detailed Runtime Analysis. To get a better understanding of which parts of the algorithms require the most runtime, we performed fine-grained measure-

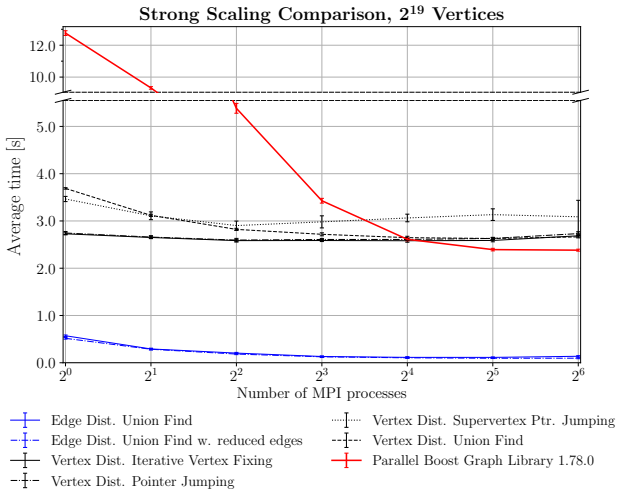


Fig. 2: Strong scaling comparison with the baseline, 2^{19} vertices

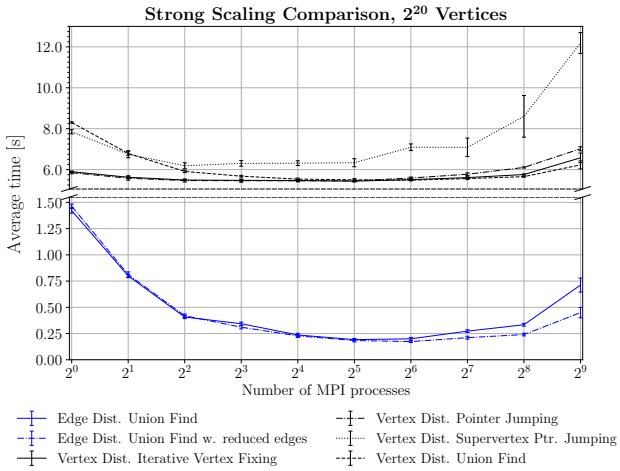


Fig. 3: Strong scaling comparison without the baseline, 2^{20} vertices

MPI processes	Best impl.	Parallel BGL
1	0.51679	12.76185
2	0.28577	9.31996
4	0.18393	5.38284
8	0.12564	3.42770
16	0.10462	2.61154
32	0.09486	2.39382
64	0.09639	2.38184

Table 1: Runtime results in seconds from strong scaling comparison, 2^{19} vertices

ments. We differentiated between graph distribution, lightest edge selection and merging. This benchmark was performed using 64 MPI processes and a graph with 2^{19} vertices.

Summarizing Results. As we are working with runtimes, the arithmetic mean is the obvious choice for summarization. The results for one algorithm execution were summarized by taking the maximum runtime of any MPI process. To get a summary over all algorithm iterations, an arithmetic mean was computed.

Repetitions. We collected measurements until the 90% confidence interval was within 5% of our reported means. This was achieved by performing 10 repetitions in all benchmarks, where the first execution is discarded due to warmup.

4.3. Results

This section contains the results of the performed benchmarks, including an analysis.

Baseline Comparison. The results of the strong scaling comparison for the baseline comparison are presented in Fig. 2. All our implementations exceed the performance of Parallel BGL when work is distributed on 8 or fewer MPI processes. For more than 8 MPI processes, only the edge distributed algorithms outperform the baseline. We verified that these results are statistically significant by making sure that the 90% confidence intervals do not overlap. An interesting observation is that Parallel BGL performs exceptionally poorly for low MPI core counts, whilst our implementations are more consistent. Table 1 contains the absolute runtimes of our best implementation, *Edge Distributed Union Find with reduced edges*, and the baseline. The best speedup of $9.31996s/0.28577s \approx 32.6$ was achieved when using 2 processes.

Implementation Comparison. Fig. 3 contains the results of our strong scaling comparison. As can be seen, the edge distributed implementations outperform the vertex distributed implementations substantially. All vertex distribution implementation perform similarly apart from the supervertices pointer jumping one which performs worse in comparison as the number of MPI processes increases. The reason for this is explained in the fine-grained analysis. It is also worth mentioning, that the standard deviation for all implementations is minuscule, except the one of the supervertex pointer jumping. This is most likely due to the randomness of that merging strategy. Another interesting point is that the edge distributed implementations do not scale after reaching 16 MPI processes. As the ratio between vertices and edges is exactly $1:16$, it is not surprising to see diminishing returns past 16 MPI processes. This is because the lightest edge computation communicates proportionally to the amount of vertices of the input graph. After the 16 process threshold, the communication outweighs the per-process work, result-

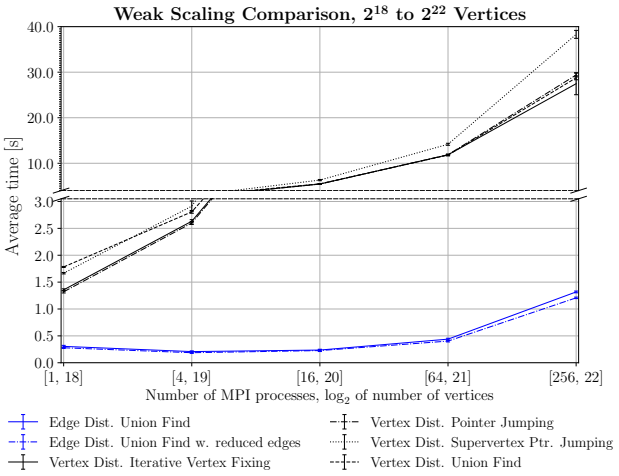


Fig. 4: Weak scaling comparison, 2^{18} – 2^{22} vertices

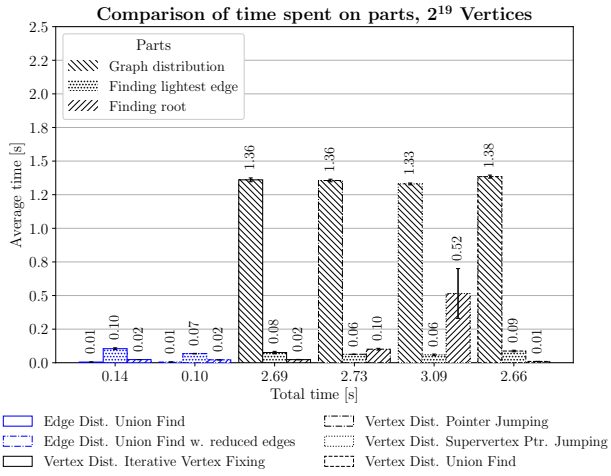


Fig. 5: Comparison of time spent on parts, 2^{19} vertices, 64 MPI processes

ing in worse performance for increasing MPI processes. More dense graphs might benefit from running on more MPI processes. The weak scaling comparison of Fig. 4 shows a similar picture. In addition, the weak scaling comparison indicates that the edge distributed implementations will perform better with a growing number of processes and increasing number of vertices.

Detailed Runtime Analysis. Fig. 5 gives a more detailed view and compares the runtime of the algorithm parts. The work distribution is an immense bottleneck for the vertex distributed implementations, whereas it only makes up a small part of the total runtime of the edge distributed implementations. An explanation of this could be, that the edge distribution implementations use an `MPI_Scatter`, potentially implemented as a tree. In the vertex distribution implementations, the work is sequentially distributed by the MPI process `0`. The lightest edge selection performs very similarly across all implementations. On the other hand, the Union Find merging strategy outperforms all others, except for the iterative vertex fixing approach. Interestingly, this approach outperforms the other two pointer jumping strategies, even though it has a worse theoretical runtime. This result indicates that the additional synchronization required is not worth the tighter theoretical runtime bound. The same holds for the supervertex pointer jumping, the strategy requiring the most synchronizations, which performed poorly in the strong scaling comparison. In addition, the plot shows that the large standard deviation of the *Vertex Distributed Supervertex Pointer Jumping* implementation comes from the merge step. This strengthens the claim made in the previous paragraph.

5. CONCLUSIONS

As is evident from the benchmark evaluation in section 4.3, we outperform the Parallel BGL by a wide margin. The best approach to distribute the work is the edge distribution strategy. The merging strategy used by the algorithm does not have a significant impact on the performance. However, the *Supervertex Pointer Jumping* performs the worst. In conclusion, we can say that we achieved our goal and found approaches that perform better than one state of the art implementation. The Parallel BGL is, of course, more sophisticated and supports generic graph data structures as well as other algorithms. The results we found are similar to those of Chung and Condon [9] in that supervertex pointer jumping performs worse than pointer jumping on structured and random graphs.

Future Work. Besides the obvious step of finding new strategies and implementing more combinations, a possible way on how to improve the current implementations is by resolving the limitations described in section 3. In particular, storing the actual MST and supporting all MPI configurations independent of the input graph. Moreover, depending on graph properties, such as the connectivity or the average degree, different strategies could be applied at run-time. This would require further, more-detailed measurements. Other work could also try to benchmark different graph types, for instance Erdös-Renyi graphs, and figure out which graph properties influence the runtimes. Moreover, it would be interesting to see if a parallelized merging strategy using a wait-free Union Find data structure [12] performs better than our current implementations.

6. REFERENCES

[1] Ronald L Graham and Pavol Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985.

[2] Yogen K. Dalal and Robert M. Metcalfe, "Reverse path forwarding of broadcast packets," *Commun. ACM*, vol. 21, no. 12, pp. 1040–1048, dec 1978.

[3] Nicos Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Tech. Rep., Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[4] Enea Spada, Luciano Sagliocca, John Sourdis, Anna Rosa Garbuglia, Vincenzo Poggi, Carmela De Fusco, and Alfonso Mele, "Use of the minimum spanning tree model for molecular epidemiological investigation of a nosocomial outbreak of hepatitis c virus infection," *Journal of clinical microbiology*, vol. 42, no. 9, pp. 4230–4236, 2004.

[5] Stephen J Salipante and Barry G Hall, "Inadequacies of minimum spanning trees in molecular epidemiology," *Journal of clinical microbiology*, vol. 49, no. 10, pp. 3568–3575, 2011.

[6] Chris A Cocosco, Alex P Zijdenbos, and Alan C Evans, "A fully automatic and robust brain mri tissue classification method," *Medical image analysis*, vol. 7, no. 4, pp. 513–527, 2003.

[7] Otakar Borůvka, "O jistém problému minimálním," 1926.

[8] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová, "Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history," *Discrete mathematics*, vol. 233, no. 1-3, pp. 3–36, 2001.

[9] Sun Chung and Anne Condon, "Parallel implementation of boruvka's minimum spanning tree algorithm," in *Proceedings of International Conference on Parallel Processing*. IEEE, 1996, pp. 302–308.

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

[11] Jucele França de Alencar Vasconcellos, Edson Norberto Cáceres, Henrique Mongelli, and Siang Wun Song, "A parallel algorithm for minimum spanning tree on gpu," in *2017 International symposium on computer architecture and high performance computing workshops (SBAC-PADW)*. IEEE, 2017, pp. 67–72.

[12] Richard J. Anderson and Heather Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1991, STOC '91, p. 370–380, Association for Computing Machinery.

[13] Robert Endre Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, no. 2, pp. 215–225, apr 1975.

[14] Joseph JéJé, "An introduction to parallel algorithms," *Reading, MA: Addison-Wesley*, vol. 10, pp. 133889, 1992.

[15] Lyndon Clarke, Ian Glendinning, and Rolf Hempel, "The mpi message passing interface standard," in *Programming environments for massively parallel distributed systems*, pp. 213–218. Springer, 1994.

[16] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al., "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[17] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani, "Kronecker graphs: an approach to modeling networks.," *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[18] Graph 500, "Graph 500 Benchmark Specification," `https://graph500.org/?page_id=12`.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition, 2009.

[20] Nick Edmonds, Douglas Gregor, and Andrew Lumsdaine, "Parallel boost graph library," `https://www.boost.org/doc/libs/1_61_0/libs/graph_parallel/doc/html/index.html`.

[21] Kathleen Fuh and Shreya Vemuri, "Parallelizing borůvka's algorithm," `https://kfuh1.github.io/15418-project/`.

[22] Manuel Burger, "RACKlette - HPC," `https://racklette.ethz.ch/`.

[23] Jan Treibig, Georg Hager, and Gerhard Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.