Master's Thesis

A Practical Scalable Shared-Memory Parallel Algorithm for Computing Minimum Spanning Trees

Wei Zhou

Submission Date: March 30, 2017

Supervisors: Prof. Dr. rer. nat. Peter Sanders
Prof. Guy E. Blelloch

Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 30. März 2017

Wei Zhou

Abstract

The present thesis briefly reviews the history of the Minimum Spanning Tree (MST) Problem and a number of known algorithms to solve it. Furthermore, a new simple, elegant and practical algorithm based on Borůvka's algorithm for parallel MST computation on shared-memory machines is developed. The algorithm utilizes a parallel primitive called *priority write* that is easily and efficiently implementable with atomic compare-and-swap (CAS) instructions. The parallelism in the algorithm is coarse-grained and no explicit locks other than in the implementations of usual parallel primitives are needed. Experiments show that the algorithm is efficient on both synthetic and real-world graphs and is invulnerable to adversarial inputs. In our tests, it performs faster than or as fast as state-of-the-art implementations, which often perform poorly or even sequentially on particular classes of graphs. The new algorithm offers good performance even with few processors and therefore can be used as a sole universal implementation.

Acknowledgments

I feel fortunate and grateful for receiving joint supervision from Prof. Peter Sanders at the Karlsruhe Institute of Technology (KIT) and Prof. Guy E. Blelloch at the Carnegie Mellon University (CMU) for this thesis. I would like to thank the interACT program and the IGEL scholarship at KIT for their financial and organizational support for my stay at CMU, where most of the work in this thesis was finished. I am also sincerely grateful to Yan Gu at CMU and Dr. Julian Shun at the University of California, Berkeley for their help and all the valuable discussions.

Contents

1 Introduction 1

2 Previous Work 3
2.1 Classical Algorithms . . . 3
2.1.1 Borůvka's Algorithm . . . 3
2.1.2 Prim's Algorithm . . . 7
2.1.3 Kruskal's Algorithm . . . 8
2.2 Modern Algorithms . . . 9
2.2.1 Yao's Algorithm . . . 10
2.2.2 Cheriton-Tarjan Algorithm . . . 10
2.2.3 Variants of Kruskal's Algorithm . . . 11
2.2.4 Other Algorithms . . . 12
2.3 Parallel algorithms . . . 14
2.3.1 Preliminaries . . . 14
2.3.2 Borůvka's algorithm . . . 17
2.3.3 Prim's algorithm . . . 18
2.3.4 Kruskal's algorithm . . . 18

3 The New Algorithm 22
3.1 Priority Write . . . 22
3.2 Compaction . . . 24
3.3 The Algorithm . . . 25

4 Experimental Results 29
4.1 Benchmark Configuration . . . 29
4.2 Benchmark Results . . . 33
4.2.1 Graphics and Tables for the Experiments . . . 33
4.2.2 Analysis for Sequential Algorithms . . . 46
4.2.3 Analysis for Parallel Algorithms . . . 47

5 Conclusions and Outlooks 49

References 50

List of Figures

1 A graph and its MSF. . . . 1
2 A graph and the pseudo-forest formed during a Borůvka step. . . . 6
3 Path compression . . . 9
4 Union-by-size. . . . 9
5 Benchmark results for `randLocal_20M` graph. . . . 33
6 Benchmark results for `rMat_20M` graph. . . . 34
7 Benchmark results for `2Dgrid_20M` graph. . . . 34

8 Benchmark results for 3Dgrid_20M graph. . . . 35
9 Benchmark results for stars_20M graph. . . . 35
10 Benchmark results for chain_20M graph. . . . 36
11 Benchmark results for delaunay_20M graph. . . . 36
12 Benchmark results for delaunay_20M-n graph. . . . 37
13 Benchmark results for delaunay_20M-2n graph. . . . 37
14 Benchmark results for delaunay3d_10M graph. . . . 38
15 Benchmark results for delaunay3d_10M-2n graph. . . . 38
16 Benchmark results for delaunay3d_10M-4n graph. . . . 39
17 Benchmark results for uniform_20M_20M graph. . . . 39
18 Benchmark results for uniform_2M_20M graph. . . . 40
19 Benchmark results for uniform_200K_20M graph. . . . 40
20 Benchmark results for uniform_20K_20M graph. . . . 41
21 Benchmark results for nlpkkt240 graph. . . . 41
22 Benchmark results for USA graph. . . . 42
23 Benchmark results for livejournal graph. . . . 42

List of Tables

2.1 MST algorithms on PRAM. . . . 21
4.1 System specifications. . . . 29
4.2 Sizes of the test graphs and the numbers of MST edges. . . . 32
4.3 Running times of sequential algorithms. . . . 43
4.4 Running times of parallel algorithms. . . . 44
4.5 Relative and absolute speedups of parallel algorithms. . . . 45

List of Algorithms

1 Borůvka Step 5
2 Borůvka's Algorithm 6
3 Priority Write 22
4 Compare-and-Swap 23
5 Priority Write with Compare-and-Swap 23
6 Pointer Jumping 24
7 New Parallel MST/MSF Algorithm 26

1 Introduction

Given a connected weighted undirected graph $G = (V, E)$ where V and E are the set of vertices and edges, respectively, and an edge weight (or cost) function $w : E \to \mathbb{R}^+$, the *Minimum Spanning Tree (MST)* of G is defined as a *connected* subgraph $T = (V, E_0)$ such that $E_0 \subseteq E$ and the total weight of the chosen edges, $\sum_{e \in E_0} w(e)$, is minimized. T must be a tree because fewer edges will disconnect the graph while more edges will introduce cycles in T and removing any edge on a cycle will reduce the total weight while keeping T connected. Some authors explicitly require acyclicness in the definition of MST.

The definition can be extended to unconnected G. In that case, we define the *Minimum Spanning Forest (MSF)* to be the set of the MSTs of all the connected components of G. We may also allow non-positive weights in the definition of MST by explicitly requiring T to be a tree with the lowest total weight. A graph with marked MSF edges is given in Figure 1.

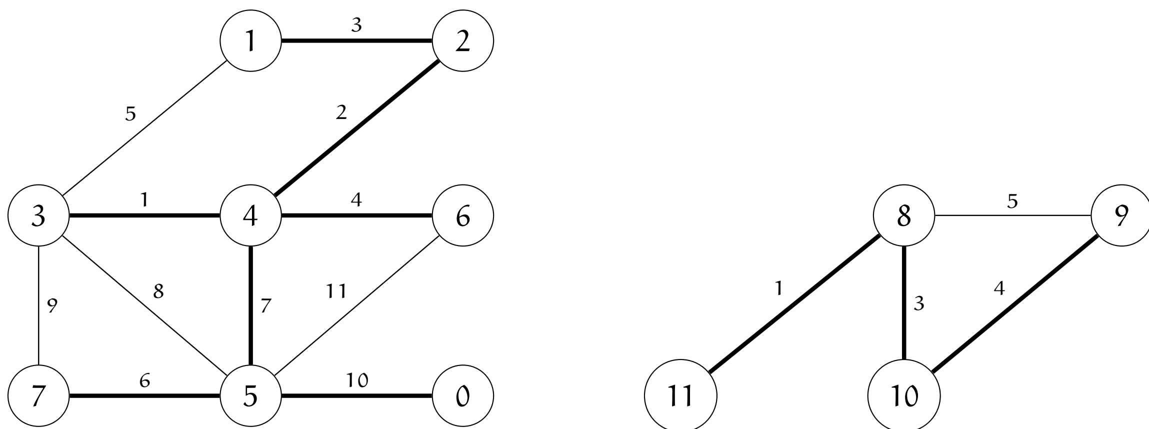


Figure 1: A graph and its MSF. The boldface edges are in the MSF.

The *Minimum Spanning Tree (Forest) Problem* is to find an MST (MSF) for a given graph $G = (V, E)$. Note that there can be more than one MST (MSF) for a graph with the same minimum total weight. Any single one is a valid solution to the problem.

Minimum spanning trees have many theoretical and practical applications. The most direct and obvious one is network design, e.g. road network or electrical grid construction. The goal of the former is to connect all cities with roads and we would like to minimize the cost for the construction. MST problem also occurs as the critical part of other algorithms. Notable examples include an approximation algorithm to the NP-Hard *Metric Traveling Salesman Problem (Metric TSP)*[1] that is no worse than twice of the optimal cost (see e.g. [1]). The MST problem is also closely related to the concept of *matroids* [1].

Largely due to physical limitation, the clock frequency of a CPU and its computing power cannot rise arbitrarily. In contrast, however, the data volume of modern information processing is growing faster than ever. This motivates the study of parallel computation.

[1] The problem is defined as follows: given n points with a distance function $d(\cdot, \cdot)$ satisfying the triangle inequality, i.e. $d(a, c) \leqslant d(a, b) + d(b, c)$ for all a, b and c, find a tour with the minimum total traveled distance that visits every point exactly once and then leads back to the starting point.

In this thesis, we develop an efficient coarse-grained parallel algorithm for the MST problem for shared-memory architectures. The algorithm is applicable on multi-core computers that are ubiquitous nowadays. Experimental results indicate that, at least for the MST problem, even consumer-grade PCs can handle huge graphs efficiently.

The thesis is outlined as follows. The next section, Section 2, introduces some fundamental properties of MSTs that enable their efficient computation and then briefly presents some known algorithms, both classical and modern ones, serial and parallel ones, for MST computation. Section 3 focuses on the main result of the thesis, a new parallel algorithm for the MST problem based on Borůvka's algorithm introduced in Section 2. The new algorithm is evaluated on synthetic and real-world graphs in Section 4 together with some of the known algorithms in Section 2. Section 5 concludes the thesis with a summary and some outlook on future research.

Throughout this thesis, we use $n = |V|$ and $m = |E|$ to denote the number of vertices and that of edges, respectively. Vertices are uniquely numbered and identified from 0 to $n-1$ and edges from 0 to $m-1$. An edge is represented by a tuple $e = (u, v, w)$ where u and v are the two endpoints and w the weight. We use $e.u$, $e.v$ and $e.w$ to denote the three components of the tuple of edge e. We also assume that the edge weights are distinct. Ties can be broken by any deterministic property of the edges, e.g. unique edge indices. For graphs with distinct edge weights, the MST is also unique. This is easily proved by contradiction using the Cycle property described in Section 2. Unless otherwise noted, "graphs" in this thesis are assumed to be connected, undirected and weighted. The thesis focuses on MST instead of MSF. This does not incur loss of generality because all of the presented algorithms can be easily adapted for MSF computation without raising their asymptotic time complexities. For sequential algorithms, one can first perform a breadth-first search and run the MST algorithms on each connected component. Most parallel MST algorithms can handle unconnected graphs natively because of the way they work. We can otherwise always resort to a parallel algorithm to the connectivity problem[2] which is intuitively an easier problem (see e.g. [2]). Since most of the algorithms discussed in this thesis can trivially be implemented with $\mathcal{O}(m+n)$ space which is asymptotically optimal[3], space complexity is generally omitted throughout the thesis unless noteworthy.

[2] The connectivity problem is the problem of identifying all connected components of a given graph.
[3] This holds as long as the input is also taken into account.

2 Previous Work

The Minimum Spanning Tree Problem is one of the fundamental problems of algorithmics and therefore has been extensively studied. In this section we briefly introduce some of the known algorithms for solving the MST problem. We first look at some classic serial algorithms and then later parallel ones.

MSTs have various properties that can be exploited to compute them. The following are some of the most frequently used ones. All of those properties have simple proofs by contradiction which are omitted here (see e.g. [1, 3]).

Property 2.1 (Cycle property). *If G contains a cycle C and there is only one edge e on C whose weight is* **strictly** *higher than edges on C, then e cannot be in an MST.*

Definition 2.2 (cut, cut edge). *A* **cut** *C of a graph G is a partitioning of its vertices into two disjoint sets V_0 and V_1. A* **cut edge** *with respect to C is an edge that connects a vertex in V_0 and another in V_1.*

Property 2.3 (Cut property). *If a cut edge e with respect to cut $C = \{V_0, V_1\}$ is* **strictly** *lighter than any other cut edges with respect to C, then e belongs to an MST. In the case where all edges have distinct weights, e belongs to* **the** *MST.*

Property 2.4 (uniqueness). *If all edges in graph G have distinct weights, then G has a unique MST.*

2.1 Classical Algorithms

2.1.1 Borůvka's Algorithm

The earliest explicit formulation of the MST problem and an efficient MST algorithm is believed to be due to Borůvka in 1926 [4, 5, 6]. His algorithm has been re-discovered several times during the next decades by others, e.g. Sollin [6], and therefore bears many names, most notably "Borůvka's algorithm" and "Sollin's algorithm". We call it "Borůvka's algorithm" throughout the thesis.

Borůvka's algorithm runs in phases. It maintains the set of found connected components, denoted S, during the process. It starts with n trivial components, each containing a single vertex. We then iteratively reduce the number of components in the set by so-called *Borůvka steps* in each phase until there is only one connected component in the set.

A Borůvka step basically tries to find *safe edges* and add them to the MST by exploiting the Cut property. It does so by finding the minimum outgoing edge[4] for every connected component in the set S. By the Cut property (by choosing the cut $C = (T, V \setminus T)$ for each component $T \in S$), those edges must be in the MST. After adding those edges to MST, it joins the respective connected components of the two endpoints of every found MST edge.

It is easy to prove that no more than $\mathcal{O}(\log n)$[5] Borůvka steps are needed. That is because the number of vertices in the *smallest* connected component at least doubles after each step and a connected component cannot contain more than n vertices. The total time complexity of Borůvka's algorithm is therefore $\mathcal{O}(\log n \cdot T_{\text{borstep}}(m, n))$, where $T_{\text{borstep}}(m, n)$ depends on the

[4] Though the graph is undirected, we sometimes assign conceptual directions to the edges. No actual modification of the edges is performed.

[5] $\log n$ is always base-2 logarithm and $\ln n$ is e-based throughout this thesis. Logarithms of other bases are given explicitly.

concrete implementation of Borůvka steps. Borůvka steps can be implemented in $\mathcal{O}(m)$ time so that the total running time of Borůvka's algorithm is $\mathcal{O}(m \log n)$. One such implementation is described below.

The implementation assumes an edge-list representation[6] of graphs and connected components are implicitly defined by recording the *representative* of the connected component of every vertex. Let $R[u]$, $u \in V$, denote the representative of the connected component in which vertex u lies. The implementation works in three steps: *find-min*, *grafting* and then *shortcutting*.

Find-Min Step. The find-min step enumerates all edges that connect two different connected components and updates the current lightest outgoing edge for both connected components. By using R values to check whether both endpoints are in the same component, it is easy to see find-min works in $\mathcal{O}(m)$ time.

Grafting Step. After finding the minimum edges, they are added to the MST. The grafting step merges the two connected components joined by any such minimum edge $e = (u, v, w)$ by "grafting" one component to the other, i.e. setting $R[i] \leftarrow j$ where $i = R[u]$ and $j = R[v]$ are the representatives of the components of u and v, respectively. Note that after this step, the meaning of R changes slightly so that the true representative of the component containing a non-representative vertex u is found by following the *path* defined by those R values, i.e. by iteratively setting $u \leftarrow R[u]$ until it stops changing. However, this may never stop when there is a cycle. This issue is discussed later. This grafting step takes $\mathcal{O}(n)$ time because every connected component has no more than one minimum outgoing edge and we only do an assignment for each such edge.

Shortcutting Step. The final shortcutting step then fully "shortcuts" all such paths by finding the true representative of i's component and then setting the R values for all vertices along this path to this representative. This step takes $\mathcal{O}(n)$ time because each transition from i to $R[i]$ will only be processed at most twice — once while trying to find the true representative and once while shortcutting along the path — and there are only $\mathcal{O}(n)$ such transitions.

The total running time of this Borůvka step implementation is therefore $\mathcal{O}(m)$, implying an $\mathcal{O}(m \log n)$-time implementation of Borůvka's algorithm. Algorithm 1 details this implementation of Borůvka step and Borůvka's algorithm is given in Algorithm 2.

As mentioned before, there is a caveat in the grafting step: an edge could be chosen as the minimum edge by the components of both its endpoints. It is thus important not to add an edge to the MST twice in line 8 of Algorithm 1 on the one hand. On the other hand, careful analysis reveals that the relation "$\rightarrow$" where "$A \rightarrow B$" means the minimum edge for connected component A goes to component B, forms a *pseudo-forest*, meaning there is no cycle except for roots, where every root also points back via the same minimum edge, forming cycles of length 2 (see Figure 2). This situation results in an infinite loop in the shortcutting step. This can be solved by breaking the symmetry during the grafting: before setting $R[i] \leftarrow j$ we check if the minimum edge for the current (i's) component is also the minimum for j's and, if so, we only graft i to j when $i > j$. This decision is arbitrary as long as it is deterministic — we may also choose to graft when $i < j$. The solution is reflected in line 13.

There are other popular implementations of Borůvka's algorithm assuming different graph representations. The edge-list representation is given in detail because it is particularly suitable for the new algorithm in this thesis and enables a more elegant parallelization. A asymptotically faster variant of Borůvka's algorithm by Yao [7] is given in Section 2.2.1.

[6] In this representation, the edge tuples are stored in a simple array of length m, denoted as $E[0..m-1]$.

Algorithm 1: Borůvka Step

```
Data: Graph G = (V, E)
      Array R[0..n − 1], where R[u] is the representative of the connected component in
      which vertex u lies
Result: Updated array of representatives

1  begin
2      best[i] ← sentinel, i ∈ V    /* assuming sentinel is an edge of weight ∞ */
       /* find-min step                                                          */
3      foreach e ∈ E where R[u] ≠ R[v] do
4          if e.w < best[R[e.u]].w then
5              best[R[e.u]] ← e
6          if e.w < best[R[e.v]].w then
7              best[R[e.v]] ← e
       /* best[i] is now the lightest edge leaving i's component if i is a
          representative and there are still valid edges connecting i's
          component and another; otherwise best[i] = sentinel.                   */
8      Add those minimum edges to the MST (e.g. by marking these edges).
       /* grafting step                                                          */
9      foreach i ∈ V do
10         if best[i] ≠ sentinel then                  /* i is a representative */
11             u ← the endpoint (of edge best[i]) in the other component
12             j ← R[u]
13             if best[i] = best[j] and i < j then
14                 Do nothing                   /* break symmetry; R[i] stays i */
15             else
16                 R[i] ← j                                   /* graft i to j */
       /* shortcutting step                                                      */
17     foreach i ∈ V do
18         r ← i
19         while r ≠ R[r] do
20             r ← R[r]
           /* r is now the representative of i's component                       */
21         j ← i
22         while R[j] ≠ r do
23             (R[j], j) ← (r, R[j])              /* simultaneous assignments */
```

Algorithm 2: Borůvka's Algorithm

```
  Input:  Graph G = (V, E)
  Output: The MST of G

1 begin
2 |  R[i] ← i, i ∈ V                                          /* initialization */
3 |  cc ← n    /* number of connected components implicitly defined by R */
4 |  while cc > 1 do
5 |  |  Invoke Borůvka step (Algorithm 1)
6 |  |  cc ← |{i ∈ V | i = R[i]}|          /* every component has a representative */
7 |  return MST edges found during Borůvka steps
```

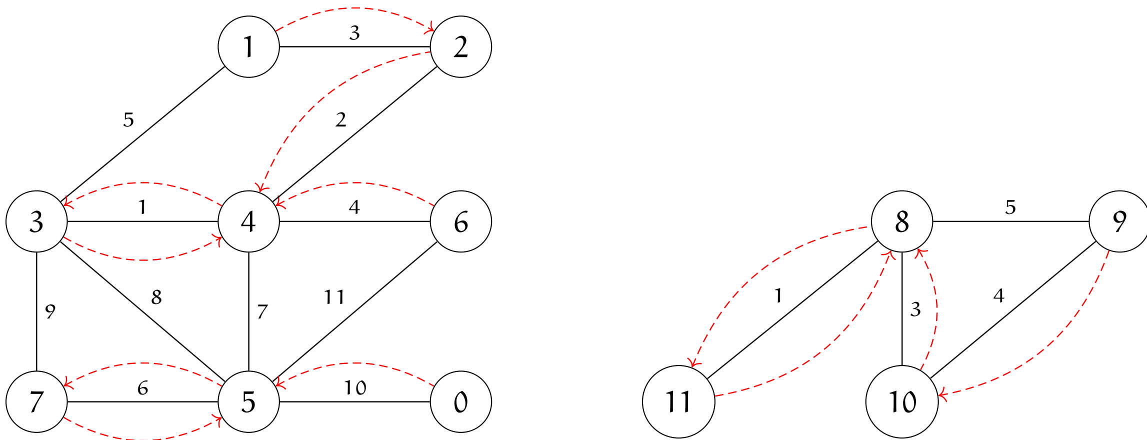


Figure 2: A graph and the pseudo-forest formed during a Borůvka step. The dashed arrows define the pseudo-forest. Note that three 2-cycles are formed. All of them must be broken to prevent infinite loops in the shortcutting step.

2.1.2 Prim's Algorithm

Prim's algorithm [8] (independently by Dijkstra [9] at around the same time), a three-decade later re-discovery of Jarník's algorithm [6], is another classical algorithm on the MST problem. It, together with Kruskal's algorithm discussed later, has received better coverage in standard texts on algorithms than Borůvka's algorithm, perhaps because the most recent re-discovery of Borůvka's algorithm by Sollin (that aroused renewed interest therein [6]) appeared a couple of years later than these two.

The algorithm works by choosing an arbitrary vertex as the starting MST and then growing the tree gradually by finding the nearest vertex to the current tree which has not been added to it. The correctness of the algorithm follows from the Cut property.

The running time of Prim's algorithm depends on how fast the nearest vertex can be found. For a dense graph ($m = \Theta(n^2)$), it suffices to maintain the minimum distance between each vertex and the MST in an array D (initialized with ∞s). This way, the next vertex can be found by a loop in time $\mathcal{O}(n)$. After adding the vertex, the minimum distances stored in array D are updated for its neighbors, which again takes $\mathcal{O}(n)$ time. The algorithm terminates after $n-1$ steps and thus has a time complexity $\mathcal{O}(n^2)$, which is asymptotically optimal.

The running time can be improved for sparse graphs with priority queues. A priority queue is an abstract data type that supports `insert`, `delete`, `find-min`, `delete-min` and `decrease-key` operations. `find-min` finds the smallest element in the priority queue and `delete-min` removes it. `decrease-key` takes a pointer to a present element in the priority queue and decreases its key by a given (non-negative) difference. General `delete` operation can be implemented by a `decrease-key` with a sufficiently large difference followed by a `delete-min`.

If we maintain the vertices in a priority queue with their respective minimum distances as the keys, finding the nearest vertex reduces to finding (and deleting) the minimum in the priority queue. After adding the minimum edge and the nearest vertex u to the MST, we update the distances for u's neighbors as before or add a neighbor if it has never been present in the priority queue. Because we only need to update vertex v's distance if the weight between u and v is smaller than the current distance of v, the update can be implemented by a `decrease-key` operation. Thus we need $(n-1)$ `delete-min` operations, $(n-1)$ `inserts` and $\mathcal{O}(m)$ `decrease-keys`. The running time of Prim's algorithm then fully depends on the concrete priority queue implementation. For binary heaps (see e.g. [1]), all three operations have complexity $\mathcal{O}(\log n)$ where n is the number of elements in the heap, giving an $\mathcal{O}((m+n)\log n)$ implementation of Prim's algorithm. Advanced priority queues like Fibonacci heaps [10] and thin heaps [11] offer constant amortized time complexity for all standard operations except for deletions which take $\mathcal{O}(\log n)$ time, enabling an $\mathcal{O}(m + n\log n)$ time implementation of Prim's algorithm, though the constant factor hidden in the complexity limits their usefulness [12, 13]. On the contrary, a data structure named pairing heap [14] proves to be fast also in practice [12, 13], though its `decrease-key` operation is now known to be $\Omega(\log\log n)$ [15], i.e. *not* optimal after a decade of being conjectured so. A modified version of pairing heaps achieving this lower bound is also known [16].

An interesting aspect of Prim's algorithm with binary heaps is its expected time complexity when the edge weights are random. [17] proves that the expected running time in this case is $\mathcal{O}(m + n\log n\log(1 + m/n)) = \mathcal{O}(m + n\log n\log\log n)$[7] even when an adversary gets to choose

[7] The reduction to the latter is proved by discussing the two cases $m \geqslant n\log n\log\log n$ and $m < n\log n\log\log n$.

the graph topology as long as the weights are random. This justifies the efficiency of this simple implementation demonstrated in practice [12, 13].

2.1.3 Kruskal's Algorithm

Another standard algorithm for computing MSTs in sequential settings is Kruskal's algorithm [18]. It maintains the set of connected components starting with n trivial ones just as in Borůvka's algorithm. However, it does not locally choose the minimum edge for each component, but globally finds the minimum edge connecting two different components and then joins them in one step. The procedure is executed until the number of connected components is reduced to one. The most prevailing way to implement the searching procedure is to first sort all edges by their weights in ascending order and inspect every edge in the sorted order to see whether both endpoints are in the same component. If they are, the edge is skipped; otherwise both components are joined and the edge is marked to be in the MST.

If we abstract the needed operations from the algorithm, we in fact need the so called *disjoint set* or *union-find* abstract data type that supports

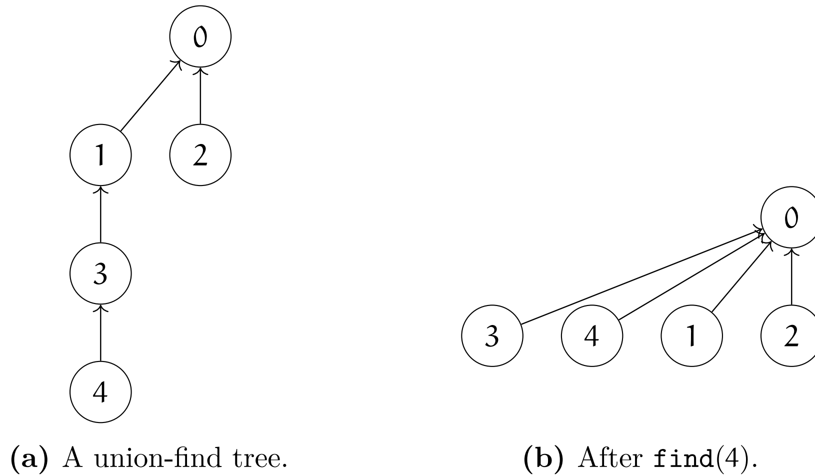
- `make-set`(x): making a singleton set containing x,
- `find`(x): finding the representative of the set in which x lies, and
- `unite`(x, y): joining the sets containing two given elements x and y.

The most suitable union-find data structure for Kruskal's algorithm is similar to the R array used in Borůvka's algorithm. Each set is represented as a tree that is implicitly defined by those R values where $R[i]$ is the parent of i in its tree. The representative of i's set is found by following R until we reach the top (where $R[i] = i$). Merging is done by grafting the root of one tree to that of the other.

This representation looks simple but inefficient. However, it can be proven that, when two techniques called *path compression* and *union-by-size* are employed, the total time for executing any sequence of operations containing m `find`s and n `make-set`s (thus at most $(n-1)$ `unite`s) is $\mathcal{O}(n + m\alpha(m+n, n))$, where $\alpha(m, n)$ is a very slow-growing function called the *inverse Ackermann function* [8] that is no larger than 4 for all practical inputs [19]. Path compression is almost what we did in the shortcutting step in Borůvka's algorithm: for every `find`(x) operation, after finding the root r, we set the parent of every element y on the path from x to r also to r, i.e. $R[y] \leftarrow r$. Intuitively, this makes future `find` operations on those elements much cheaper without increasing the complexity of the present one. Union-by-size adds an attribute to every root representing the size (number of elements) in its tree. When doing an `unite` operation, we only graft the tree of smaller size to the larger one, breaking ties arbitrarily, and update the size attribute of the new root. The size field of the smaller tree will never be used again thereafter and can be discarded if necessary. Both techniques are depicted in Figure 3 and Figure 4. They are very easy to implement yet the analysis is highly non-trivial and out of the scope of this thesis. [19] offers a complete analysis and more techniques that can be used to replace path compression and/or union-by-size and are asymptotically equally fast. Furthermore, it has been also proven in [19] that the time bound is *tight* in a sense: any implementation of the union-find data structure needs $\Omega(n + m\alpha(m+n, n))$ time in the worst case to execute a

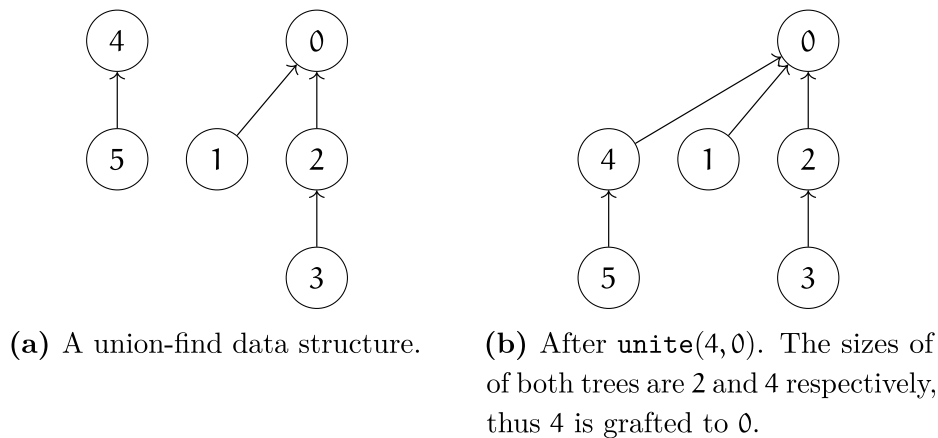
[8]Formally, $\alpha(m, n) := \min\{i \geqslant 1, i \in \mathbb{Z} \mid A(i, \lfloor \frac{m}{n} \rfloor) \geqslant \log_2 n\}$ where the *Ackermann function* is defined to be $A(1, j) = 2^j$ for $j \geqslant 1$, $A(i, 1) = A(i-1, 2)$ for $i \geqslant 2$, and $A(i, j) = A(i-1, A(i, j-1))$ for $i, j \geqslant 2$. The latter has an explosively fast growth, which is why the growth of its inverse is extremely slow.

sequence containing m `find`s and n `make-set`s for *pointer machines*[9] under certain technical assumption. For $m \geqslant n$, which is the case in Kruskal's algorithm, a simpler lower and upper bound of $\Theta(m\alpha(m, n))$ can be proven.



(a) A union-find tree. **(b)** After `find`(4).

Figure 3: Path compression



(a) A union-find data structure. **(b)** After `unite`(4, 0). The sizes of of both trees are 2 and 4 respectively, thus 4 is grafted to 0.

Figure 4: Union-by-size.

Kruskal's algorithm can be efficiently implemented using this union-find data structure. Sorting the edges costs $\mathcal{O}(m \log m)$ time. After that, we have to inspect every edge in the worst case, involving two `find` operations per edge, and possibly join both components which are $(n - 1)$ `unite` operations in total. Thus we need $\mathcal{O}(m\alpha(m, n))$ time in the worst case. In summary, Kruskal's algorithm has a time complexity of $\mathcal{O}(m \log m)$ or, if the edges are already given in sorted order, $\mathcal{O}(m\alpha(m, n))$.

2.2 Modern Algorithms

On top of the discussed classical algorithms, many more algorithms that are theoretically or practically more efficient have been proposed. In this section we give a brief introduction to some of them. We start with variants of these classical algorithms and then look at some of the more recent and advanced ones.

[9]Informally, pointer machines are a model of computation where no direct memory addressing is allowed. Memory cells must be found by following pointers and not by arithmetic.

2.2.1 Yao's Algorithm

Borůvka's algorithm makes local decisions during each Borůvka step. If we can speed up these local decisions, we can then accelerate Borůvka step. A natural direction to try is to avoid inspecting every edge during each Borůvka step somehow. It should be possible to implement Borůvka step in less than $\Theta(m)$ time if we can manage this because other steps in a Borůvka step only takes $\mathcal{O}(n)$ time in contrast to $\mathcal{O}(m)$.

Yao's algorithm [7] is the first MST algorithm that achieves an $\mathcal{O}(m \log\log n)$ time bound in the worst case. It assumes an adjacency array[10] (or list) representation of graphs. The outbound edges for every vertex are first partitioned into $k = \log n$ equal-sized groups such that every edge in a later group is at least as heavy as all edges in earlier groups for the same vertex. Edges within the same group are *not* sorted. This can be done in $\mathcal{O}(m \log k) = \mathcal{O}(m \log\log n)$ time if we recursively apply the classical linear selection algorithm (see e.g. [1]). The goal of the partitioning is to make sure later groups do not have to be checked before all earlier ones are fully exhausted. Therefore only one group, i.e. only $\frac{1}{k}$ of all edges for each vertex have to be checked to determine the minimum edge for this vertex. We remember the current group ID for every vertex across the Borůvka steps. After determining the minimum edge for a vertex, we mark it or (lazily) remove it. If exhausting the whole group did not yield any usable edge, we increment the group ID for the vertex and search again in the next group. The total cost for *failed* searches of the groups for an unused edge is bounded by the number of edges across all Borůvka steps and is therefore $\mathcal{O}(m)$. We still have $\mathcal{O}(\log n)$ stages, so the total time for all stages is $\mathcal{O}\left(m + \log n \cdot \sum_{i=1}^{n} \left\lceil \frac{|\text{neighbors}(i)|}{k} \right\rceil\right) = \mathcal{O}\left(m + \log n \cdot (\frac{2m}{k} + n)\right)$ which is $\mathcal{O}(m + n\log n)$ for $k = \log n$. For $m = \Omega(n \log n)$ this is a linear time algorithm. In order to eliminate the $n \log n$ term for the case $m = \mathcal{O}(n \log n)$, we first run normal Borůvka steps (with $k = 1$, i.e. no partitioning) for $\log\log n$ stages to reduce the number of vertices by a factor of $\log n$. This preprocessing takes $\mathcal{O}(m \log\log n)$ time, which is also the running time of the whole algorithm. Yao's algorithm is often considered impractical because it involves the linear selection algorithm which has a high constant factor [20].

2.2.2 Cheriton-Tarjan Algorithm

Another $\mathcal{O}(m \log\log n)$ algorithm for MST is due to Cheriton and Tarjan [21]. The algorithm also maintains a set of connected components implicitly represented by the representatives of every component, just as in Borůvka's algorithm. For every component it manages a *meldable* priority queue, i.e. a priority queue that supports efficient merging of two queues, of all edges having exactly one endpoint inside the component. Edge weight is the key of the priority queues.

The algorithm works in a *round-robin* fashion. At the beginning we have n trivial connected components in a (normal, first-in first-out) *queue*. At each step, the algorithm pops the first element from the queue and then the minimum edge from its priority queue. We find the other endpoint of the edge (the one that is not in this component) and its corresponding component and remove it from the queue. We then merge the two connected components and their associated priority queues. Finally the merged component is appended to the queue. The

[10] An adjacency array representation has an edge list sorted by their starting vertex (thus every undirected edge is stored twice in the list), and an array of length n representing the index of the first outgoing edge for every source index. This way, we can efficiently find all neighbors of any given vertex.

steps are repeated until we have only one connected component in the queue. This connected component represents the MST of the original graph.

Components can be represented by the same union-find data structure as in Kruskal's algorithm. Therefore the merging of two connected components can be efficiently implemented. It remains to find a priority queue implementation that supports efficient merging. Cheriton and Tarjan chose a variant of so-called *leftist heaps* invented by Crane [22]. A leftist *tree* is a binary tree such that, for every node in the tree, the right path from the node (the path along the pointers of the right children until there is none) is the shortest among all paths from the node to the bottom. Another way to characterize leftist trees is to define a function $\mathrm{rank}(x)$ for every node x as

$$\mathrm{rank}(x) = \begin{cases} 0, & \text{if } x \text{ is an external node} \\ 1 + \min\{\mathrm{rank}(\mathrm{left}(x)), \mathrm{rank}(\mathrm{right}(x))\}, & \text{otherwise} \end{cases}$$

where $\mathrm{left}(x)$ and $\mathrm{right}(x)$ are the left and right child of x, respectively. A leftist tree is defined as a binary tree satisfying $\mathrm{rank}(\mathrm{left}(x)) \geqslant \mathrm{rank}(\mathrm{right}(x))$ for every internal node x, hence its name. A leftist *heap* is then a leftist tree satisfying the heap order as in binary heaps. In addition to `insert` and `delete-min`, leftist heaps also support `merge` operations that merge two leftist heaps in $\mathcal{O}(\log n)$ time, where n is the total number of elements in both trees, in contrast to ordinary binary heaps, for which it would take $\mathcal{O}(n)$ time. The variant used in Cheriton-Tarjan algorithm uses *lazy deletion* and *lazy merging* to achieve the claimed $\mathcal{O}(m \log\log n)$ time bound. The running time of this algorithm can be improved slightly further to $\mathcal{O}\left(m \log\log_{2+m/n} n\right)$ by doing some cleanup operations that remove duplicate or self edges (edges that connect vertices in the same connected component) at appropriate intervals. Further details can be found in [21] and [20]. This algorithm is also deemed impractical because the small speedup over classical algorithms does not overweigh the constant factors due to the use of more complicated pointer based data structures.

2.2.3 Variants of Kruskal's Algorithm

The bottleneck of Kruskal's algorithm resides in the sorting step. Intuitively, the number of connected components is often reduced to one (thus the MST has already been found) long before every edge is inspected. A result from the theory of random graphs states that the expected number of edges that need to be checked is about $\frac{1}{2} n \ln n$ for large enough n [23], much fewer than m for dense graphs.

A method to incorporate this observation is to build a priority queue on the edges instead of sorting and then use `delete-min` to get the next lightest edge until the MST has been built. Building a priority queue takes $\mathcal{O}(m)$ time for many implementations, e.g. binary heaps (see e.g. [1]). Each subsequent `delete-min` needs $\mathcal{O}(\log m)$ time. The worst-case time complexity is therefore $\mathcal{O}(m + m \log m + m\alpha(m, n)) = \mathcal{O}(m \log n)$ just as the original algorithm, but the average case is now only $\mathcal{O}(m + n \ln n(\log m + \alpha(m, n))) = \mathcal{O}\left(m + n \log^2 n\right)$. This implementation is called *Kruskal's algorithm with demand-sorting* by some authors [12].

Another algorithm demonstrating this early-stopping idea works like Quick-sort. If the number of edges is small enough (e.g. $m = \Theta(n)$), we run a normal Kruskal's algorithm with union-find. Otherwise, we choose a pivot edge (uniformly at random or the median) and partition the edges into two sets: $E_1 = \{e \mid e.w \leqslant \mathrm{pivot}.w\}$ and $E_2 = \{e \mid e.w > \mathrm{pivot}.w\}$. We

then recurse with only the edges in E_1. After returning from the recursion, we check if the MST has already been built and only recurse with edges in E_2 if it has not. During the whole algorithm, a global union-find data structure is used. This algorithm, sometimes called the *Quick-Kruskal algorithm* [24], has the same average time complexity as Kruskal's algorithm with demand-sorting, i.e. $\mathcal{O}\left(m + n \log^2 n\right)$ [25].

The above two algorithms with early-stopping only work well for random graphs with random weights as early stopping is of no use even if the MST contain a single heavy edge. If the topology is fixed and can be chosen by an adversary, random weights alone cannot guarantee the above average time complexity [24]. An example given in [24] is a "lollipop graph" that consists of a random graph and a tail (a chain) connected to one node. Obviously, all of the edges on the tail must be in the MST, and the probability that an edge is in the heavier half of all edges is $\frac{1}{2}$, implying about half of the edges on the tail are in the heavier half, rendering the early-stopping ineffective.

A simple remedy of the Quick-Kruskal algorithm called the *Filter-Kruskal algorithm* is given in [24]. It adds a *filtering step* which removes the edges that have endpoints in the same components after recursing processing the lighter half of the edges, before recursing with the heavier half. This filtering is done with help of the global union-find data structure. Experiments show that his heuristic is indeed much more robust than early-stopping alone. From a theoretical point of view, they proved that the expected running time of Filter-Kruskal algorithm is $\mathcal{O}\left(m + n \log n \log \frac{m}{n}\right) = \mathcal{O}\left(m + n \log n \log \log n\right)$ for *arbitrary graph* with random weights. Note the same bound is also achieved by Prim's algorithm with binary heaps under the same condition as mentioned before in Section 2.1.2.

The filtering idea can be used without recursive partitioning. We simply do a *one-time* partitioning with Quick-select (see e.g. [1]) to find the (e.g. $\Theta(n)$) lightest edges, run Kruskal's algorithm (or any other MST algorithm that *return* a union-find data structure) with those edges, do a filtering on the heavier edges, and invoke Kruskal's algorithm on the remaining heavier edges. This is what the parallel implementation of Kruskal's algorithm in the *Problem Based Benchmark Suite (PBBS)* [26] does. More details on that algorithm is given in Section 2.3.4.

2.2.4 Other Algorithms

In the last few decades, a couple of algorithms that are asymptotically faster than Yao's and Cheriton and Tarjan's $\mathcal{O}(m \log \log n)$ have been proposed. Most of them are highly complicated and thus deemed impractical, or are only applicable to graphs with certain special properties. We only give an incomplete listing here.

Fredman and Tarjan [10] gave an $\mathcal{O}(m \log^* n)$ time algorithm for the MST problem that invokes Prim's algorithm implemented with Fibonacci heaps iteratively, where the *iterated logarithm* $\log^* n := \min\{i \mid \underbrace{\log \log \cdots \log}_{i \text{ times}} n \leqslant 1\}$ and $\log^* n \leqslant 5$ for all practical input. The rationale of the algorithm is to limit the number of elements that coexist in the priority queue to reduce the $n \log n$ term in the time complexity of Prim's algorithm (which is $\mathcal{O}(m + n \log n)$). The algorithm runs in passes, each of which executes Prim's algorithm until the number of elements in the priority queue exceeds a certain threshold k or after it just added a vertex into the priority queue that has been marked by a previous Prim instance, at which point it starts a new instance of Prim's algorithm in another vertex. The pass ends when every vertex belongs to the tree of an instance of Prim's algorithm. Every tree found by a Prim instance

is contracted into a super-vertex. The i-th pass can be implemented in time $\mathcal{O}(m + n_i \log k)$ where n_i is the number of vertices before the pass. If we set $k = k_i = 2^{\frac{2m}{n_i}}$, that would be $\mathcal{O}\left(m + n_i \log(2^{\frac{2m}{n_i}})\right) = \mathcal{O}\left(m + n_i \cdot \frac{2m}{n_i}\right) = \mathcal{O}(m)$. Furthermore, it can be shown that no more than $\frac{2m}{k_i}$ super-vertices remain after pass i, because the total degree of every Prim tree after the pass is at least k_i and the total degrees of the whole graph is $2m$. Therefore we have $k_{i+1} = 2^{\frac{2m}{n_{i+1}}} \geqslant 2^{\frac{2m}{2m/k_i}} = 2^{k_i}$. Thus the sequence of k_i increases *tetrationally*, indicating the number of passes is $\mathcal{O}(\log^* n)$, proving a total time complexity $\mathcal{O}(m \log^* n)$ for the algorithm. This algorithm is then (very) slightly improved by Gabow et al. [27] to $\mathcal{O}(m \log \log^* n)$ shortly thereafter.

The bound for deterministic algorithms for the MST problem was again lowered by Chazelle [28] to $\mathcal{O}(m\alpha(m, n))$ by utilizing an interesting data structure called *soft heaps* also invented by him [29]. A soft heap is a meldable priority queue implementation that may *corrupt* elements stored in it by increasing their keys. Soft heaps support `delete-min` and `merge` in constant amortized time and `insert` in $\mathcal{O}\left(\log \frac{1}{\epsilon}\right)$, satisfying the additional property that no more than ϵn elements in the data structure are corrupted at any time, where $0 < \epsilon \leqslant \frac{1}{2}$ is the *error rate*. The errors are introduced to break the information-theoretic lower bound on priority queues because otherwise insertions and deletions could be used to do comparison-based sorting in $o(n \log n)$ time. A similar algorithm with the same bound has also been independently proposed by Pettie [30].

Pettie and Ramachandran [31] ultimately have broken the bound again by providing an asymptotically *optimal* deterministic comparison-based algorithm for the MST problem on pointer machines. An intriguing fact about the algorithm is that although the running time of their algorithm is proven to be matching the decision tree lower bound for pointer machines, this bound itself is not known. The best upper and lower bound of the running time of the algorithm to date are $\mathcal{O}(m\alpha(m, n))$ (achieved by Chazelle [28] as mentioned above) and $\Omega(m)$, respectively. They have also proved that their algorithm works in linear time with high probability for random graphs even if the lower bound should later be shown to be superlinear. Roughly, their algorithm first generates optimal *decision trees* for graphs of no more than $r = \log \log \log n$ vertices in $\mathcal{O}(n)$ time. Then it partitions the graph into subgraphs of about r vertices in $\mathcal{O}(m)$ time. After that, the MST for each subgraph is calculated by using the optimal decision trees, for which we do not know the exact time bound. The found MSTs are contracted into individual super-vertices in linear time. The remaining graph is dense because the number of vertices is reduced to $\frac{n}{r}$. For such a graph, the MST can be found in linear time by invoking previous algorithms (e.g. Fredman and Tarjan's $\mathcal{O}(m \log^* n)$ time algorithm). Therefore the total running time is dominated by the application of the decision trees and other steps take linear time in total.

Note that the lower bound for the MST problem, should it prove to be superlinear, does not necessarily hold for models other than pointer machines. For example, on the so-called *trans-dichotomous model*, a unit-cost RAM model where a word cannot hold *unreasonably* much data, the MST can be found in deterministic linear time for integral edge weights [32]. Deterministic linear time can also be achieved for special graphs. For example, the MST problem for *planar* graphs can be solved in $\mathcal{O}(n)$ time and for dense graphs (where $m = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$) in $\mathcal{O}(m)$ time [21]. The limit on the density can be relaxed to $m = \Omega(n \log \log \log n)$ while retaining a linear time bound for all asymptotically faster algorithms described in this section [31]. This fact is exploited in the optimal algorithm described above.

If we have access to a stream of perfectly random bits with uniform distribution, the MST

problem can be solved in expected linear time by an algorithm given by Karger et al. [33]. Their algorithm relies on a linear *verification* algorithm, an algorithm that verifies the minimality of a spanning tree in linear time and returns witnesses for edges that do not belong in the true MST. One such verification algorithm on word RAM model is given by King [34] and one for pointer machines by Buchsbaum et al. [35]. With the latter, this MST algorithm can be completely implemented on pointer machines. The MST algorithm first executes Borůvka step twice to reduce the number of vertices by a factor of 4. It then chooses a random sample of all edges by including each edge with $\frac{1}{2}$ probability and finds the MSF F of the chosen subgraph recursively. The heavy edges with respect to the found MSF F are then filtered out with help of the verification algorithm. This is an application of the Cycle property. If we denote the remaining light edges as L, the algorithm finally returns the MST of $F \cup L$ with a recursive call. The key observation that leads to the claimed running time is that L has an expected size of only $\frac{n}{2}$. Therefore the expected running time is $T(m,n) = T(\frac{m}{2}, \frac{n}{4}) + O(n+m) + T(\frac{n}{2} + \frac{n}{4}, \frac{n}{4}) = O(n+m)$ where $T(m,n)$ denotes the running time of the algorithm on a graph with m edges and n vertices.

Katriel et al. [36] focused on a practical modification of the randomized linear time algorithm for dense graphs by ignoring the Borůvka steps, selecting a sample of edges of size $\sqrt{mn}$ instead of $\frac{m}{2}$, utilizing Prim's algorithm instead of recursion and reducing the verification step to range minimum queries (RMQ) that make use of a byproduct of the first Prim instance, namely the order in which vertices are added into the MST. The resulting algorithm has expected time complexity $\mathcal{O}(m + n\log n + \sqrt{mn})$ if Prim's algorithm is implemented with Fibonacci heaps. They reported favorable performance with pairing heaps.

There are more algorithms that are based on the Cycle property. For example, the so-called *reverse-delete* algorithm that Kruskal described in the same paper as Kruskal's algorithm [18] works by looking at the edges in *descending* order by weight and delete an edge if it does *not disconnect* the graph. As a practical implementation would need an efficient algorithm for dynamic connectivity supporting edge deletion which is highly non-trivial, this algorithm is not usually used. An implementation by Thorup [37] achieves a running time of $\mathcal{O}(m\log n(\log\log n)^3)$.

2.3 Parallel algorithms

This section discusses some parallel algorithms for the MST problem. For that, we first give some fundamental definitions and constructs in parallel computing.

2.3.1 Preliminaries

Models of Computation The most-used computation models in shared-memory parallel computation are the Parallel Random-Access Machine (PRAM) models, an analogy to the RAM model for sequential algorithms. On a PRAM, we have a set of p processors identified by unique indices called processor IDs. The number of processors may depend on the problem size. All processors have access to a global shared memory and have their own private registers and memory that others cannot access. All processors work synchronously and each can perform a standard arithmetic or logic operation on a memory cell within one clock cycle. There are multiple PRAM variants that differ in the way they handle concurrent memory accesses to the same memory cell within the same clock cycle: **Exclusive-Read Exclusive-Write**

(EREW): No concurrent reads or writes to the same memory location are allowed; **Concurrent-Read Exclusive-Write (CREW):** Processors may read a memory cell concurrently and will receive the same value, but no concurrent writes by multiple processors are allowed; and **Concurrent-Read Concurrent-Write (CRCW):** Processors may read or write to the same memory location at the same time. Technically it is also possible to define an Exclusive-Read Concurrent-Write (ERCW) PRAM, but this is rarely used because write access is generally assumed to imply read access. In the case of CRCW PRAM, it is important to define the result of concurrent writes. Again here are several variants given in the increasing order of ability: **CRCW-Common** where all processors writing to the same cell in the same clock cycle must write the same value and this value is stored into the cell; **CRCW-Arbitrary** where processors may write different values and an arbitrary (random) processor would succeed in writing its value into the cell while the values other processors want to write are ignored; **CRCW-Priority** where the processor with the smallest ID (thus highest *priority*) succeeds; and **CRCW-Reduction** where an associative operator (a reduction) somehow combines the written values the processors attempt to write into a single one and writes it into the cell, e.g. taking the sum, minimum or maximum. Although the latter two seem much stronger than CRCW-Common, which in turn looks stronger than EREW PRAM, it is possible to simulate a concurrent memory access of CRCW-Reduction or CRCW-Priority PRAM on an EREW PRAM within only $\mathcal{O}(\log p)$ parallel time by exploiting the fact that sorting n elements can be done on EREW PRAM in $\mathcal{O}(\log n)$ time (see [38]).

Within PRAM models, we can define quantities that characterize the performance of algorithms. Basically we have the *depth* which is the number of clock cycles that have elapsed before all processors (hence the program or algorithm) terminate, and *work* which is the sum of clock cycles of every processor in which the processor performed operations (in contrast to being idle). The main advantages of using depth and work to specify the complexity of a PRAM algorithm is the simplicity to translate them into the running time of the algorithm on a real-world computer via *Brent's theorem*. Brent's theorem, also called the Work-Time Scheduling Principle, states that we can simulate a PRAM algorithm of time $T(n)$ and work $W(n)$ with unlimited number of processors on p processors within $T_p(n) = \mathcal{O}\left(\frac{W(n)}{p} + T(n)\right)$ time on the same PRAM model [38].

In order to describe parallel algorithms, the following well-studied fundamental operations are defined and used as primitives later for describing parallel algorithms.

Parallel Map. Perhaps the simplest parallel construct is the *map* function. $\mathtt{map}(f, L)$ applies a function f to every element in the sequence L and returns a sequence containing the results. Formally, $\mathtt{map}(f, L) := \{f(x) \mid x \in L\}$. `map` is normally required to return the sequence of new values in the same order as their corresponding values in L. If the function f is *pure*, i.e. it does not have any side-effect and its value only depends on the argument, $f(x)$ can be evaluated for every $x \in L$ in parallel. Due to the simplicity to parallelize such a `map` operation, it is sometimes called "embarrassingly parallel". If function f does not have a return value, but does something to the environment based on its argument like writing to an array, `map` is also called parallel *foreach* or parallel *for-loop*. `map` is present in most functional programming languages, though it is not always executed in parallel.

Parallel Reduction (Fold). Another frequently used primitive is *reduction* or *fold*. For a pure associative binary function f and a sequence L, `reduce` applies f to the first two elements in L, and then successively applies f to the last function value and the next element of L. Formally,

its functionality is defined as $\texttt{reduce}(f, L[0..n-1]) := f(\texttt{reduce}(f, L[0..n-2]), L[n-1])$ and $\texttt{reduce}(f, L[0..1]) := f(L[0], L[1])$. Often, we also define the result of `reduce` on an one-element list to be the list itself. Though any associative f can be used, the most frequently used ones by far are `min`, `max` and `sum`. The associativity of f comes into play when we try to implement `reduce` in sub-linear time. Since $\texttt{reduce}(f, L[0..n-1]) = f(\texttt{reduce}(f, L[0..\frac{n}{2}]), \texttt{reduce}(f, L[\frac{n}{2}+1..n-1]))$ by associativity and both arguments of the outer application of f are independently computable, `reduce` can be implemented in depth $\mathcal{O}(\log n)$ on EREW PRAM. By Brent's theorem, we can simulate the algorithm in $\mathcal{O}\left(\frac{n}{p} + \log n\right)$ time on a machine with p processors. If $p < \log n$, we can achieve a depth of $\mathcal{O}\left(\frac{n}{p} + p\right)$ easily by breaking the list into p part and letting each processor compute the sum of one sublist. A single processor computes adds the sums of the sublists together to produce the final result.

Parallel Prefix-Sum (Scan). Prefix-sum, also called *scan*, is a stronger primitive than reduction and is present in many parallel algorithms. It also forms the foundation of other parallel primitives. This operation computes all prefix sums of a given sequence. Two versions of prefix-sum exist, namely inclusive and exclusive prefix-sums. They differ in whether $L[i]$ is included in the prefix-sum of position i. Prefix-sum is stronger than reduction because the last value of an inclusive prefix-sum is exactly the reduction. At first glance, it is not obvious that prefix-sum can be computed efficiently in parallel because its definition seems inherently sequential. However, it can also be computed in $\mathcal{O}(\log n)$ time too by a two-phase algorithm sometimes called *Blelloch scan* [39]. Prefix-sums can also be generalized to any associative operators besides addition.

Parallel Filtering. With help of prefix-sum, we can implement another useful primitive: filtering. $\texttt{filter}(\texttt{pred}, L)$ gives a sequence containing the elements of L for which the predicate `pred` gives true in their original relative order, i.e. $\texttt{filter}(\texttt{pred}, L) := \{x \in L \mid \texttt{pred}(x) = \texttt{true}\}$. For an efficient implementation of `filter`, we first use a parallel `map` to compute an array $\texttt{flags}$ of 0 or 1 where $\texttt{flags}[i] = 1$ if and only if $\texttt{pred}(L[i])$ is true. An exclusive prefix-sum of $\texttt{flags}$ is calculated. This gives the count of elements before $L[i]$ where the predicate gives true, and is thus the final position of $L[i]$ in the resulting sequence if $\texttt{pred}(L[i])$ is true. A final parallel for-loop copies these values to the corresponding position of the target array. A special case of filtering, $\texttt{pack}(\texttt{flags}, L)$, simply stores elements of L whose $\texttt{flag}$ values are 1 into another array. If we also want to get those elements with flags of 0, we may use a `partition` or `split` which is slightly more efficient than two filtering operations because it can reuse some intermediate computations.

Parallel Sorting. As in sequential computation, sorting is also a fundamental building block of parallel algorithms. Many efficient algorithms exist for PRAM (see e.g. [38]). On real-world multi-core CPUs, people seem to prefer sample sort which proves to be efficient [40]. The algorithm chooses a sample of the original array, sorts the sample, chooses $m-1$ equidistant values as *splitters* and partitions the original array into m disjoint buckets with those splitters. Buckets are recursively sorted and then concatenated to form the sorted array. We omit the details here.

The most natural choice of classical algorithm for parallelization is Borůvka's algorithm because of the way it makes decisions: every Borůvka step finds the minimum outgoing edge for a component *locally*, thus can be executed for all components in parallel. That is probably why

Borůvka's algorithm received far more attention in the era of parallel computation than Prim's and Kruskal's in the literature. We first look at some implementations of Borůvka's algorithm and then a few of Prim's and Kruskal's algorithm.

2.3.2 Borůvka's algorithm

Bader and Cong [41] gave several parallel implementations of Borůvka's algorithm and are among the first that reported reasonable speedup on shared-memory multiprocessors. All of these implementations do a true graph compaction (also known as contraction) after grafting in a Borůvka step, i.e. contract every component to a super-vertex, instead of the implicit one given in Algorithm 1. They use different representations of graphs and thus have slightly different implementations of Borůvka steps.

Variant 1. One implementation (called Bor-EL) uses an edge-list representation as in Algorithm 2 but every edge is stored twice in the list (one for each direction). The contraction step is simply sorting the edge list with the super-vertex (or representative as in Algorithm 2) of the first endpoint as the primary, that of the second endpoint as the secondary and the edge weight as the tertiary key. Self-loops within the same component and multi-edges between components are removed with a subsequent parallel prefix-sum.

Variant 2. A second implementation (Bor-AL) uses adjacency array as the graph representation where every edge is again in both lists of outgoing edges of its endpoints. The contraction step is accomplished by sorting the array of vertices by their representatives, which is equivalent to sorting all the edges by the representatives of their first endpoints, and then concurrently sort the outgoing edges for every vertex by the representatives of the second endpoints with a sequential sorting algorithm (insertion sort for short lists and merge sort for longer ones). Redundant edges are then removed with prefix sums.

Variant 3. A last implementation of Borůvka's algorithm which is perhaps more interesting uses a *flexible adjacency lists (array)* representation of graphs (denoted Bor-FAL). This implementation reduces the cost for compaction by allowing every vertex to hold a *list of adjacency lists (arrays)*. After grafting, every vertex plugs its list of adjacency arrays to that of its representative and edges themselves do not have to be moved around. The find-min step has to check for redundant edges because self and multi-edges are not removed with this representation.

A fundamental problem with all these implementations is that their find-min steps are done concurrently for all vertices but *sequentially* for each vertex. That means there can be massive load imbalance if the degrees of the vertices differ too much, as is the case for graphs containing star-shaped subgraphs. Furthermore, the first two implementations uses the expensive parallel sort to build the contracted graph. The Bor-FAL implementation does not have this problem, but may degenerate to the cache-unfriendly adjacency-list representation if vertex degrees are evenly distributed.

Another implementation of Borůvka's algorithm using adjacency-array representation is given by da Silva Sousa et al. [42]. In the contraction step, the implementation uses atomic incrementing instructions to count the new number of outgoing edges to get the index of the first edge for each vertex. Subsequently it copies edges between to the new edge array, again using atomic increments. Self-edges within a component are removed but multi-edges between components are not. This implementation also suffers from the load-imbalance problem in the

find-min step and in the contraction step due to atomic increments.

A very recent work of Cong and Tanase [43] aims to reduce the cost for memory accesses and improve locality in Borůvka's algorithm, generalizing ideas proposed in [44]. Their implementation (independently) exploits some ideas similar to the present thesis. We feel this thesis is still justified because the new algorithm proposed here has other new ideas and the design was finalized prior to the publication of their work. When describing the new algorithm we will give reference to their work that demonstrates the same or similar idea.

2.3.3 Prim's algorithm

Prim's algorithm is apparently inherently sequential and hard to scalably parallelize. Known algorithms run several instances of Prim's algorithm starting from different vertices simultaneously and stop to do some kind of merging when two trees touch. Such algorithms can be fast on many graphs, but are vulnerable to adversarial or non-random input in which case the algorithms can be forced to run sequentially. One implementation by Bader and Cong [41] that combines Prim's algorithm with Borůvka's is given here.

The algorithm works by running multiple instances of Prim's algorithm with binary heaps simultaneously with different starting vertices. Every Prim instance performs sequentially as normal and *colors* the found vertices to its unique color when they are first added into its heap. It runs until the heap becomes empty or it extracts a vertex from its heap that is colored by another processor or has any neighbor that is.

After all Prim instances are stopped, we add the found trees to the MST. For every vertex that is not removed from the heap by any Prim instance, we also add its shortest outgoing edge to the MST. Then we shrink the components induced by the MST edges to super-vertices in parallel. Super-vertices with no outbound edges are removed because they represent connected components of the original graph.

After the shrinking we start over again with multiple Prim instances. The whole process is repeated until the number of remaining super-vertices goes below a threshold, at which point we find its MST by executing Prim's algorithm sequentially.

Since every processor running a Prim instance may visit different number of vertices during an iteration, load-imbalance may occur. This is remedied with a work-stealing technique. They also noted that the algorithm may make no progress during an iteration for very special graphs. This can be solved if vertices are shuffled in advance. A definite worst-case for this algorithm is star-shaped graph where almost no progress can be made in the Prim instances and they loop through all neighbors of a vertex by a single processor, essentially sequentializing the algorithm. This problem is not easily solvable by work-stealing in their algorithm as stated.

2.3.4 Kruskal's algorithm

Of the two main phases of Kruskal's algorithm, sorting is very well parallelizable (see e.g. [38]). In the contrary, the second part is hard to scalably parallelize because whether to accept or to reject an edge depends on the decisions made earlier. Known implementations can be forced to run sequentially by an adversary and will do more work than the sequential implementation. However, a good speedup on graphs that arise in practice is still achievable. One such practical implementation is given by Blelloch et al. [26] and included in the PBBS.

Blelloch et al.'s algorithm utilizes a technique called *deterministic reservation*. First, the following structures and operations are introduced as building blocks of the algorithm:

Priority write. For a memory cell x, a priority write, denoted x.pwrite(v), sets the value in x to v if v is smaller than the original value stored in x (thus having a higher *priority*). If multiple processors perform this operation on the same memory cell simultaneously, the memory cell will contain the smaller of the original value stored in the cell and the smallest value of all those write operations. On modern shared-memory architectures, this operation can be implemented with the atomic compare-and-swap (CAS) instruction within a loop. More on that is given later in Section 3.

Priority reserve. A data type (called a reservation station) is introduced which supports three operations: priority reserve (x.reserve(p)), check (x.check(p)) and check-and-release (x.checkR(p)), where p is conceptually the priority of the operation. x.reserve(p) reserves the memory location x with priority p. This reservation fails (or will be canceled) if other reservations with higher priority have reserved (or should reserve later) the same location. The function returns whether the reservation was successful. x.check(p) checks if x is reserved with priority p. x.checkR(p) checks if x is reserved with priority p and cancels the reservation if so. Cancellation is done by storing a sentinel priority $\perp$ to the cell to denote the cell is now not reserved. This value is also used to initialize the data type.

We now describe their algorithm. Edges are first sorted into nondecreasing order using a parallel sample sort. An array of reservation stations is initialized with $\perp$. A union-find data structure for the n vertices is initialized to contain singletons. The algorithm takes a *prefix* of the array of edges, and for an edge between u and v of index i in the sorted order, it tries to reserve stations u and v with priority i if u and v do not belong to the same connected component according to the current union-find structure. This can be done with a parallel for-loop. After all reservations are submitted, a *commit step* performs a parallel loop over that prefix to check if the respective edge successfully reserved at least *one of* u and v. If so, it releases the reservation on the other station, marks the edge as an MST edge, and merges the components of u and v as in the normal Kruskal's algorithm by linking *the smaller of* u *and* v *to the larger*. This ensures no cycle is formed. Note that we lose the benefit of union-by-size by doing this. After that, a prefix-sum is used to move the unsettled (neither discarded nor successfully committed) edges together to the front of the remaining unprocessed edges and a new iteration is started with another prefix of the same length of the unsettled edges and some new edges.

Another optimization mentioned before can also be used. Namely we only choose a small number (say $\frac{4}{3}n$) of the lightest edges and run the algorithm on this reduced list of edges. After that we filter out the heavy edges with respect to the current spanning forest and run the algorithm on the remaining edges. They reported a relative speedup of 18 and absolute speedup of 10 (against an optimized serial implementation of Kruskal's algorithm with the same filtering optimization) on a machine with 32 cores (64 threads with hyper-threading). This algorithm is used as the main rival of the new algorithm introduced in Section 3.

Another parallel algorithm based on Kruskal's algorithm is the Filter-Kruskal algorithm described in Section 2.2.3. The parallelism resides in the base case where $m = \mathcal{O}(n)$, where a parallel sorting algorithm can be used, the partitioning and the filtering. In fact, the base case can be any other parallel MST algorithm as long as it maintains the global union-find data structure in the Filter-Kruskal algorithm. This algorithm with the above parallel Kruskal's

algorithm of PBBS as the base case is also included in the experiments in Section 4.

Katsigiannis et al. [45] tried to speed up Kruskal's algorithm using helper threads that discard heavy edges on cycles while the main threads executes the normal Kruskal's algorithm. The edges are still considered by the main thread in non-decreasing order. Helper threads examine the edges that have not yet been checked by the main thread and mark discarded the edges forming cycles in the present forest with help of the union-find data structure. The main thread first consults an boolean array that the helper threads write to to see whether the edge is already discarded before checking for cycle itself. Because multiple helper threads are running, one may hope that many edges are already discarded when the main thread comes to those edges. Though they reported a speedup of 5 on 8 cores on some synthetic graphs, it can be seen this algorithm does not scale well as the number of threads goes up because of low utilization of the helper threads. In fact, it even demonstrated *slow-down* when the number of helper threads increases to some (not very high) point.

Apart from parallel implementations of originally sequential algorithms, there are also algorithms designed to be parallel, especially on PRAM models. We give an incomplete summary of the results in Table 2.1. Many of these PRAM algorithms share strong similarities with the classical sequential algorithms, especially Borůvka's algorithm.

A notable example is due to Awerbuch and Shiloach [46]. The algorithm exploits a variant of the CRCW-Priority PRAM model where the priority of processors are not defined by their IDs, but by another fixed attribute. The algorithm needs one processor for each edge and one for each vertex. That priority attribute of the processors representing edges is edge weight. Each iteration of the algorithm is also roughly divided into find-min, grafting and breaking symmetry, and shortcutting where the find-min is implicitly performed by exploiting the concurrent write capability. The difference from Algorithm 1 is that the shortcutting step only does one linking step for each vertex instead of a full compression of the trees. The number of iterations can still be proven to be $\mathcal{O}(\log n)$. Zaroliagis [47] modifies this algorithm by incorporating the partitioning idea in Yao's algorithm and achieves the same depth within the CRCW-Common PRAM model. The algorithm is more complicated and calls the above algorithm to reduce the number of vertices for sparse ($m < n \log^2 n$) graphs. It also makes use of a number of other algorithms for different tasks, notably a algorithm that simulates CRCW-Priority PRAM on CRCW-Common PRAM.

Both algorithms and most of the other algorithms in Table 2.1 do not permit easy efficient implementation on real-world machines because simulating CRCW PRAM on computers is generally not practical, and the partitioning would incur large constant factor anyway. The EREW ones are often complicated. In retrospect, the ideas in the present thesis (priority writes) might be applicable to implement these algorithms on real computers. This could be a direction for future work.

There are also algorithms designed for other architectures. For example, [48, 49, 50, 51, 42] aim at GPUs and [52, 53, 54] at distributed-memory architectures. These are out of the scope of this thesis.

Algorithm	Depth – Work	Note[※]
Chin et al. [55], 1982	$\mathcal{O}\left(\log^2 n\right) - \mathcal{O}\left(n^2\right)$	EREW
Cole and Vishkin [56], 1986	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left((m+n)\log\log\log n\right)$	CRCW-P
Awerbuch and Shiloach [46], 1987	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left((m+n)\log n\right)$	CRCW-P
Karger [57], 1992	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left(m+n^{1+\epsilon}\log n\right)$	EREW
Johnson and Metaxas [58], 1992	$\mathcal{O}\left(\log^{3/2} n\right) - \mathcal{O}\left((m+n)\log^{3/2} n\right)$	EREW
Cole et al. [59], 1994	$\mathcal{O}\left(2^{\log^* n}\log n\right) - \mathcal{O}\left(m+n\right)$	CRCW-A, randomized
Cole et al. [60], 1996	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left(m+n\right)$	CRCW-A, randomized
Poon and Ramachandran [61], 1997	$\mathcal{O}\left(2^{\log^* n}\log n\log\log n\right) - \mathcal{O}\left(m+n\right)$	EREW, randomized
Zaroliagis [47], 1997	$\mathcal{O}\left(\log^2 n\right) - \mathcal{O}\left((m+n)\log n\right)$	EREW
(same as above)	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left((m+n)\log n\right)$	CRCW-C
Pettie and Ramachandran [62], 1999[¤]	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left(m+n\right)$	EREW, randomized
Chong et al. [63], 2001	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left((m+n)\log n\right)$	EREW
Chong et al. [64], 2003	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left((m+n)\sqrt{\log n}\right)$	EREW
(same as above)	$\mathcal{O}\left(\log n\right) - \mathcal{O}\left((m+n)\log\log n\right)$	CRCW-A

Table 2.1: MST algorithms on PRAM.

[※] CRCW-P stands for CRCW-Priority, CRCW-A for CRCW-Arbitrary, CRCW-C for CRCW-Common.

[¤] A preliminary version of the work appeared in 1999. The referenced version was published in 2002.

3 The New Algorithm

In this section we describe a new parallel algorithm based on Borůvka's algorithm. The main steps of the algorithm do not differ from the original Borůvka's algorithm: we still have an outer loop that keeps invoking Borůvka step until we have a single connected component. The Borůvka step still has three main steps, i.e. find-min, grafting and shortcutting as in Algorithm 1. We first describe the necessary building blocks of the algorithm and then describe the new algorithm incorporating these building blocks.

3.1 Priority Write

A main ingredient of the new algorithm is *priority write* or *priority update*. This primitive is used in Blelloch et al.'s parallel implementation of Kruskal's algorithm and briefly described in Section 2.3.4. Priority write of the value v to a memory cell identified by x (thus x is a pointer or address) is denoted `x.pwrite(v)` or `pwrite(x, v)`. The operation replaces the value at the destination of x with v if v is smaller than the original value stored there. More formally, the operation is functionally equivalent to Algorithm 3:

Algorithm 3: Priority Write

```
1 Function pwrite(x: pointer to value, v: value)
2     if *x > v then    /* "*x" is the value stored in the destination of x */
3         *x ← v
```

Problems arise when this operation is used in parallel settings. When multiple processors do this operation simultaneously, data race will occur. Take the following processor interleaving as an example where $*x$ is initialized to 5 and Processor 1 and 2 call `pwrite(x, 3)` and `pwrite(x, 1)`, respectively:

```
// *x is initialized to 5.
Processor 1: if 5 > 3 then              // true
Processor 2: if 5 > 1 then              // true
Processor 2:    *x <- 1                 // *x is now 1
Processor 1:    *x <- 3                 // *x is now 3
// *x = 3 at the end
```

Here $*x$ is incorrectly set to 3 after both `pwrites`. We therefore require `pwrite` to be *atomic*. On modern shared-memory architectures, `pwrite` is not readily available but can be easily implemented with atomic compare-and-swap (CAS) instructions. A CAS operation is functionally equivalent to Algorithm 4, but guaranteed to be atomic by the hardware (see

e.g. [65]):

Algorithm 4: Compare-and-Swap

```
1 Function cas(x: pointer to value, old: value, new: value): boolean
2     if *x = old then
3         *x ← new
4         return true                                   /* success */
5     else
6         return false                                  /* failure */
```

With help of CAS instruction, the priority write operation can be implemented free of data race as follows in Algorithm 5:

Algorithm 5: Priority Write with Compare-and-Swap

```
1 Function pwrite(x: pointer to value, v: value)
2     do
3         old ← *x
4     while v < old and cas(x, old, v) = false   /* short-circuiting in effect */
```

This piece of code repeatedly reads the current value of $*x$ to a variable old, checks if the intended new value v is still smaller than old, and does a CAS to store v into $*x$ if it is. The loop ends when at some point the newly read value old is no larger than the intended new value v or a CAS has succeeded. Note that the second condition is checked only if the first is true because of short-circuiting. From now on, pwrite denotes this version of priority write.

Priority writes may be defined for any value type with a total ordering. Due to hardware limitation on atomic compare-and-swap instruction, however, the size of supportable value types is normally restricted to that the largest supported primitive integral type, e.g. 16 bytes on current mainstream 64-bit processors (see e.g. [65, 66]).

Priority writes have been first introduced in [26] and extensively analyzed by Shun et al. [67]. They may seem very inefficient at first glance due to the presence of the loop and CAS instruction. More careful reasoning reveals that the CAS (and a potential write) only occurs when the present value in $*x$ is larger than the intended new value and the loop is broken immediately when this is not the case. Therefore the performance of the primitive depends on how often a new value is written by the CAS instruction by all processors. Intuitively, this does not happen all that often because a written value will prevent many future writes. In fact, if every of p processors attempts to call $\texttt{pwrite}(x, v_i)$ where v_i is drawn uniformly at random from a range or v_i is the i-th value in a random permutation of values, the expected number of writes is only $\ln p + \mathcal{O}(1)$. That is because a random permutation of length p only has $H_p := \sum_{i=1}^{p} \frac{1}{i} = \ln p + \mathcal{O}(1)$ *prefix-minima* in average with high probability, where H_p is the p-th harmonic number [68]. This is also the expected running time if p processors write to a single location with random values at the same time. The worst case happens when the values are written in decreasing order. This is extremely unlikely because that would mean the CAS supporting hardware somehow orders all writes in the hardware queue in descending order. Because it is the operating system that schedules concurrent processes invoking pwrite, and perfect synchronization, which does not exist in real world, would be needed for an adversary to forge such a situation, we may safely assume the expected running time is $\mathcal{O}(\ln p)$ regardless of the input. If we use p processors to make n pwrites and we assume the n operations are divided into batches of size p as is roughly the

case with parallel for-loop, each of the batch takes $\mathcal{O}(\log p)$ time. Thus the expected total time for all operations is $\mathcal{O}\left(\frac{n}{p}\log p\right)$. Shun et al. [67] gave a result of $\mathcal{O}\left(\frac{n}{p} + c\log n + cp\right)$ where c is a constant characterizing the cost of the ensuring cache-coherence.

The situation becomes more obscure when p processors execute n `pwrite`s to m different locations. If the `pwrite`s are made to random locations, we can expect the contention on a single cell to reduce. The exact running time seems to be open at this time. Shun et al. [67] gave a bound of $\mathcal{O}\left(\frac{n}{p} + cm\log\frac{n}{m} + (cp)^2\right)$. This result seems unfavorable, especially because of the second and third terms which do not decrease or even rise as p goes up. That is probably because their result makes a pessimistic assumption that a successful CAS by a processor results in a full invalidation of the cache of all other processors incurring time cost c. This is pessimistic because other processors do not have to be aware of the write unless they want to write to the same location. Despite the imperfect theoretical bound, they demonstrated good performance in practice, as we will also see later in the context of the new MST algorithm.

Note that there is a similarity between priority writes and the PRAM model with concurrent writes by priority (CRCW-Priority). The difference is that the argument defines the priority of a `pwrite` operation and not the processor ID which is the case with CRCW-Priority. Therefore `pwrite` is intuitively more flexible. On the other hand, CRCW-Priority can simulate `pwrite`, too, by first sorting the sequence of operations by their priorities and let the processor with the ID corresponding to the rank of the arguments perform the write. This, however, incurs extra cost due to the sorting.

3.2 Compaction

The new algorithm is based on Borůvka's algorithm, therefore it also has a shortcutting step that compacts the found connected components into super-vertices. We discuss this compaction step in this section.

Definition 3.1 (compaction)**.** *Formally, the* **compaction** *operation receives the array of parents* $R[0..n-1]$ *as input and is supposed to shortcut the trees implicitly defined by the values so that at the end of the compaction process, every vertex* i *is either a root (characterized by* $R[i] = i$*) or is the child of a root (*$R[R[i]] = R[i]$*).*

This can be solved in parallel on CREW PRAM with a pointer jumping (or path doubling) technique within $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n\log n)$ work [38] as shown in Algorithm 6.

Algorithm 6: Pointer Jumping

```
Input: Parent array R[0..n−1]
Output: Compacted parent array: a node is either a root or a child of a root

1 foreach 0 ⩽ i < n do in parallel
2     while R[i] ≠ R[R[i]] do          /* if i's parent is not i's grandparent */
3         R[i] ← R[R[i]]                          /* graft i to its grandparent */
```

Note that for parallel loops on PRAM, program code within the loop body is run by all processors simultaneously with perfect synchronization. That means all processors run line 2 at the same time and, after all processors are done checking the condition, all processors for which the condition holds run line 3 simultaneously. The synchronization is crucial for the correctness

of the theoretical time bound $\mathcal{O}(\log n)$. Indeed, consider a chain: if we let all processors *wait* until the processor representing the leaf node finally reaches the root, the time is already $\Omega(n)$.

There are more efficient algorithm for this compaction problem that only needs $\mathcal{O}(n)$ operations in total (see e.g. [38]), but it turns out that this step only takes a marginally small proportion of the time of our MST algorithm, so we do not go further optimizing it. In fact, we may even run line 2 and line 3 *without synchronization* in practice because the operating system normally schedules processes more or less evenly so that every process benefits from the shortcutting work of other processes thanks to cache-coherency protocols. This way, we can also relieve ourselves of costly locks.

3.3 The Algorithm

On a high level, the new algorithm described here does not differ much from the sequential Borůvka's algorithm given in Algorithm 2. The difference resides in how the Borůvka step is implemented. Assuming the same edge-list representation, we run through every edge in a parallel loop in the Borůvka step to determine the best edge for every current super-vertex, namely the representative of its current connected component. This is done by performing a priority write to each of the endpoints of every edge with the weight and ID as the argument. By the definition of priority writes, we will have the best edges for every connected component. After that we graft trees along the best edges, breaking symmetry just like in the sequential version, but this time in parallel. After that we perform a full compaction. A new set of active super-vertices is found with a filter operation. To improve locality of memory accesses, we change the endpoints of every edge to their corresponding representatives. Self-edges are filtered out at the end of the Borůvka step.

Now we describe our MST algorithm using the edge-list representation in detail. We say a tuple $(a_1, a_2, \ldots, a_k)$ is smaller than another tuple $(b_1, b_2, \ldots, b_k)$ if there is an i, $1 \leqslant i \leqslant k$, so that $\forall_{1 \leqslant j < i} : a_j = b_j$ and $a_i < b_i$. This also known as the *lexicographical order*. Two edges can be compared according to the lexicographical order of their corresponding $(\mathtt{weight}, \mathtt{index})$ pair, where `index` is the position of the edge in the original edge list. Operator "$\xleftarrow{\text{par}}$" stands for parallel assignment, i.e. all available processors divide the intended assignments evenly and execute their own portion in parallel. The full algorithm in given in Algorithm 7. Note that before every Borůvka step, V always only holds vertices that are representatives of their components.

As we can see, the algorithm does not differ much from the sequential implementation in Algorithm 1 and Algorithm 2. The first main difference is the use of `pwrite` in the find-min step in line 7 and line 8. Conceptually, those `pwrite`s try to store the edge into the locations for both its endpoints in `best` array. This solves the load-imbalance issue with known implementations which loop through all vertices in a parallel but process all edges for a vertex sequentially. Since CAS instructions and thus `pwrite`s have limit on the size of the operands, we actually write the pair $(\mathtt{weight}, \mathtt{index})$ of the edge with `pwrite`. This is well suitable for double-precision or 64-bit integral weights (both are 8 bytes) and 4 to 8-byte indices since most of current 64-bit architectures support CAS instructions on 16-byte operands. This is also better than only storing the `index` because we save two random memory accesses when comparing an edge with the two present in the `best` array (for both endpoints).

The grafting step almost stays the same except that it is now parallel. If $\mathtt{best}[i] = \mathit{sentinel}$

Algorithm 7: New Parallel MST/MSF Algorithm

```
   Input: Graph G = (V, E)
   Output: The MST of G

1  begin
2      R[i] <-par- i, i ∈ V                                /* initialization */
3      MST ← ∅                                     /* all the MST/MSF edges */
4      while |V| > 1 and |E| > 0 do
           /* loop invariant: V only contains root vertices.              */
           /* loop invariant: E contains no self-loops.                   */

5          best[i] <-par- sentinel, i ∈ V     /* sentinel is an edge of weight ∞ */
           /* find-min step                                               */
6          foreach e ∈ E do in parallel
7              best[e.u].pwrite(e)
8              best[e.v].pwrite(e)
           /* best[i] is now the lightest edge leaving i's component;
              best[i] = sentinel if there is none.                        */

           /* grafting step                                               */
9          foreach i ∈ V do in parallel
10             index ← index of i in the current V
11             if best[i] = sentinel then      /* no edge found for component i */
12                 R[i] ← −1                      /* inactivate that component */
13                 mst_edge[index] ← sentinel
14             else
15                 j ← the other endpoint of edge best[i]
16                 if best[i] = best[j] and i < j then
17                     Do nothing with R[i]       /* break symmetry; R[i] stays i */
18                     mst_edge[index] ← sentinel
19                 else
20                     R[i] ← j                                    /* graft i to j */
21                     mst_edge[index] ← best[i]

22         MST ← MST + {mst_edge[j] ≠ sentinel | 0 ⩽ j < |V|}         /* filtering */

           /* shortcutting step                                           */
23         Perform pointer jumping (Algorithm 6) on R for vertices in V to compact trees.

           /* relabeling step                                             */
24         E ← {(R[e.u], R[e.v], e.w) | e ∈ E}

           /* filtering step                                              */
25         E ← {e ∈ E | e.u ≠ e.v}                                    /* filtering */
26         V ← {i ∈ V | R[i] = i}                /* new set of vertices; filtering */
27     return MST
```

for some vertex i (recall that i is always a representative), that means the component of i did not get any edge in the find-min step. Thus we *inactivate* the component by setting the parent of i to a negative value in line 12. If we graft a component to another, we also mark the edge we just used to be in the MST (line 21). Subsequently, all those marked edges are added to the set of MST edges in line 22 with a parallel filtering.

As in the sequential case, we perform a shortcutting step to make every vertex either a root or a child of a root in line 23. Note we should only do this for roots vertices that are not inactivated in line 12 (i.e. those with $R[i] \geqslant 0$).

In order to further improve locality in the find-min step, we *relabel* the endpoints of the edges by changing the endpoints of every edge to their parents in line 24. This way, the find-min step in the next iteration does not have to consult the R array to determine the representative of the endpoints for an edge, saving two random memory accesses. Cong and Tanase [43] have independently come up with the same idea and applied to their variant of Borůvka's algorithm very recently.

Finally, a filtering step takes place to remove self-edges, reducing the length of the edge list and saving time in the find-min step. The vertex set is also updated to the new set of roots (thus removing exhausted components and components that are grafted to other components). The loop invariant is restored this way and wo start the loop over unless the edge set becomes empty or only a single vertex remains.

Note that this algorithm also computes the minimum spanning forest if the original graph is not connected. That is because a component is inactivated in line 12 if it is exhausted, and it will be removed from the vertex set, so it will not prevent further progress of the algorithm.

An important remark is that the parallelism in this algorithm is very coarse-grained: we are only using data parallel primitives and no locks or other manual synchronization are needed within each such construct. This is favorable because locks are too expensive to guarantee good performance.

Since the algorithm is essentially Borůvka's algorithm, we still have $\mathcal{O}(\log n)$ iterations where the number of active vertices at least halves after every iteration. In every iteration, the grafting takes $\mathcal{O}\left(\frac{n_i}{p} + \min(p, \log n_i)\right)$ time where n_i is the number of vertices at the beginning of the i-th iteration and p is the number of available processors. Shortcutting takes $\mathcal{O}\left(\frac{n_i \log n_i}{p}\right)$ time if we implement pointer jumping with synchronization. In the practice, however, we do not have to (and want to) use expensive locks to ensure synchronization, as discussed in Section 3.2. The parallel filtering and parallel for-loops take time $\mathcal{O}\left(\frac{m_i}{p} + \min(p, \log m_i)\right)$ in total, where m_i is the number of edges at the beginning of the iteration. The running time of the find-min step is still open as discussed in 3.1, but we conjecture it to be expected $\mathcal{O}\left(\frac{m_i \log m_i}{p}\right)$. In practice it runs very fast and is invulnerable to adversarial inputs. Summing up all terms, noticing $n_{i+1} \leqslant \frac{n_i}{2}$, the MSF algorithm is conjectured to finish in expected $\mathcal{O}\left(\frac{n \log n + m \log m \log n}{p}\right)$ time regardless of input, assuming $n \gg p$. The $\frac{m \log m \log n}{p}$ term looks too large and does not seem to match the performance in practice. It is of theoretical interest to prove the bound or even a better one, but in practice, it appears that we can safely use `pwrite` without much thoughts on performance. Note that if the algorithm is run with a single thread, it degenerates to normal Borůvka's algorithm. The running time in that case becomes deterministically $\mathcal{O}(m \log n)$ because each Borůvka step does no more than $\mathcal{O}(m)$ work. This makes the algorithm favorable for situations where we do not always have the full computing resources of the system.

Other optimizations can be applied to the algorithm to improve the performance even further for some graphs. One such optimization has been introduced in Section 2.3.4, namely we can run the algorithm on the lightest portion of all the edges, run a filtering on the remaining edges with help of the acquired information during the first run and run the algorithm for a second time on the edges surviving the filtering. Finding the lightest edges can be done with ideas similar to a parallel sample sort: we take a sample of all edges, sort them with a sample sort, and take the respective value in the sample according to the portion we want, and do a partitioning/splitting with that value as the pivot. The lightest edges are thereby moved to the beginning of the array. Note that we have to *reactivate* the inactivated components after the first run by setting $\mathtt{R[i]} \leftarrow \mathtt{i}$ for all $\mathtt{i}$ with negative $\mathtt{R[i]}$ because they are inactivated only because we exhausted edges *in the lightest portion*. Cong and Tanase [43] have gone even further by partitioning the edges into many buckets (instead of two) where edges in earlier buckets are lighter than all edges in later buckets and always doing a filtering on the edges in a bucket before running their MST algorithm on it.

4 Experimental Results

We have introduced the new algorithm for computing minimum spanning trees in Section 3. In this section, we compare the new algorithm with some of the existing sequential and parallel algorithms introduced in Section 2 on synthetic and real-world graphs and present experimental results that demonstrate the practical performance of our new algorithm.

4.1 Benchmark Configuration

Experiments are conducted on a workstation with the following technical characteristics:

Name	Value
OS	Ubuntu 14.04.5 64-bit
CPU	Intel(R) Xeon(R) CPU E7-8867 v4 @ 2.40GHz
Sockets	4
Cores per socket	18
Threads per core	2
Cache line size	64 bytes
Cache alignment	64-byte boundary
L2 cache size	256 KB/core
L3 cache size	45 MB/socket
RAM	1 TB

Table 4.1: System specifications.

The test programs and benchmark are based the *Problem Based Benchmark Suite (PBBS)* [69]. It is a framework that offers various data parallel primitives and can be used to compare different solutions to the same problem by their performance. Correctness can be checked against a reference implementation in a black-box manner.

The following implementations are included in the benchmark:

`seq_pbbs_kruskal`. A sequential implementation of Kruskal's algorithm provided in PBBS. It uses union-find with path-compression and union-by-size to maintain the current components.

`seq_pbbs_filtering_kruskal`. Same as above, but it uses the filtering technique described in Section 2.3.4 and Section 3, i.e. we first uses a partitioning algorithm to get the lightest edges, run Kruskal's algorithm on it, filter out self-edges with the current union-find data structure, and execute once more Kruskal's algorithm on the remaining edges.

`par_pbbs_kruskal`. The parallel Kruskal's algorithm provided in PBBS, introduced in Section 2.3.4, without filtering.

`par_pbbs_filtering_kruskal`. Same as above but with filtering.

`par_filter_kruskal`. An implementation of the recursive Filter-Kruskal algorithm described in [24] and briefly introduced in Section 2.2.3. The parallel Kruskal's algorithm with filtering from PBBS serves as the base case of the algorithm and the needed partition and filter primitives are provided by PBBS.

`seq_prim_binary`. An implementation of Prim's algorithm with binary heaps.

seq_prim_pairing. An implementation of Prim's algorithm with pairing heaps. The pairing heaps are from the Policy-Based Data Structures included in the GNU C++ compiler [70].

seq_boruvka. An implementation of Borůvka's algorithm given in Algorithm 2.

par_boruvka_d. A variant of Borůvka's described in Section 2.3.2 and [42]. Small changes were made to make it suitable for the PBBS framework including support for double-precision weights and graph format conversion. Time for conversion is not counted towards its running time.

par_new_boruvka. An implementation of the new Borůvka's algorithm presented in this thesis.

par_new_filtering_boruvka. Same as above but with filtering.

All source codes are written in C++ and are compiled using GNU C++ compiler Version 5.4.1 with relevant compilation flags `-O3 -march=native`. Parallelization is generally achieved with the Cilk++ [71] except for `par_boruvka_d`, which uses OpenMP and partly Intel Threading Building Blocks (TBB) for parallelization.

Various synthetic and real-world graphs are used in the benchmarks. These are listed below:

randLocal_20M. A random local graph with 20 million vertices and degree 5 for each vertex. The 5 edges for each vertex is chosen uniformly at random. This is generated with the graph generation utility provided in PBBS.

rMat_20M. A graph generated with the *Recursive Matrix (R-MAT)* algorithm proposed in [72]. The algorithm models real-world graphs like social networks nicely and produces ones with small diameter and power-law degree distributions. This is generated with the graph generation utility provided in PBBS.

2Dgrid_20M. A regular square 2-D grid with 20 million vertices. The side length, i.e. the number of vertices on a side, is therefore $\sqrt{20 \text{ million}}$. The vertices on the borders are also adjacent to their counterparts on the other side of the grid. This is generated with the graph generation utility provided in PBBS.

3Dgrid_20M. A regular 3-D grid with 20 million vertices with side length $\sqrt[3]{20 \text{ million}}$. The same edge-wrapping as above is also present. This is generated with the graph generation utility provided in PBBS.

stars_20M. A graph consisting of stars with 20 million edges. This is generated with the graph generation utility provided in PBBS.

chain_20M. A graph containing a single chain of increasing 20 million vertex IDs and edge weights. This is generated with the graph generation utility provided in PBBS.

delaunay_20M. A graph generated by randomly scattering 20 million points onto the unit square and building the Delaunay triangulation of the point set. The number of edges is roughly 60 million because Delaunay triangulations are in a sense planar graphs with the maximum number of edges and connected simple planar graphs never have more than $3n-6$ edges (see e.g. [73]). The edge weights are the Euclidean distances of the endpoints. This graph is generated with MathWorks Matlab [74].

delaunay_20M-n. Same as above, but n (which is 20 million) random edges of the triangulation are removed. This is to model a planar graph of "half fullness".

delaunay_20M-2n. Same as above but with $2n$ edges removed. The resulting graph

only has about n edges.

delaunay3d_10M. Similar to delaunay_20M-n, but now 10 million points instead of 20 are drawn from a unit *cube* and the edges are those formed by a three-dimensional Delaunay triangulation.

delaunay3d_10M-2n. Same as above, with $2n$ edges removed.

delaunay3d_10M-4n. Same as above, with $4n$ edges removed. Note the resulting graph still has about 40 million edges since the graph is not planar anymore and hence does not satisfy the same $3n - 6$ upper bound as in the two-dimensional case.

uniform_20M_20M. A graph uniformly drawn from the universe of all graphs of 20 million vertices and 20 million edges. The generation is done with the readily available functions RandomGraph and UniformGraphDistribution of Wolfram Mathematica [75].

uniform_2M_20M. Same as above but with 2 million vertices and 20 million edges, so that the density of the graph slightly increases.

uniform_200K_20M. Same as above but with only 200 thousand vertices.

uniform_20K_20M. Same as above but with only 20 thousand vertices.

nlpkkt240. A graph from the SuiteSparse Matrix Collection, formerly The University of Florida sparse matrix collection [76].

USA. A real-world road network of the USA, provided by the 9th DIMACS Implementation Challenge [77]. Weights are the physical distances.

livejournal. A graph taken from the Stanford Network Analysis Platform (SNAP) [78] representing the friendship network of the LiveJournal social network.

Unless noted otherwise, the edges weights in the graphs are uniformly random. Table 4.2 summarizes the used graphs, their sizes and the final MST (MSF) edges.

All time measurements are calculated by executing the respective implementations ten times and taking the average of the running times after removing the minimum and the maximum.

Name	Vertices	Edges	MST edges
randLocal_20M	20 000 000	100 000 000	19 999 999
rMat_20M	33 554 432	100 000 000	29 355 409
2Dgrid_20M	19 998 784	39 997 568	19 998 783
3Dgrid_20M	19 902 511	59 707 533	19 902 510
stars_20M	20 000 004	20 000 000	20 000 000
chain_20M	20 000 000	19 999 999	19 999 999
delaunay_20M	20 000 000	59 999 950	19 999 999
delaunay_20M-n	20 000 000	39 999 943	19 932 186
delaunay_20M-2n	20 000 000	19 999 961	17 214 746
delaunay3d_10M	10 000 000	77 586 968	9 999 999
delaunay3d_10M-2n	10 000 000	57 587 804	9 999 991
delaunay3d_10M-4n	10 000 000	37 593 541	9 997 509
uniform_20M_20M	20 000 000	20 000 000	16 762 252
uniform_2M_20M	2 000 000	20 000 000	1 999 999
uniform_200K_20M	200 000	20 000 000	199 999
uniform_20K_20M	20 000	20 000 000	19 999
nlpkkt240	27 993 601	746 478 752	27 993 599
USA	23 947 347	28 854 312	23 947 346
livejournal	4 036 538	34 681 189	3 997 961

Table 4.2: Sizes of the graphs included in the benchmark and the number of MST edges in these graphs. If the number of MST edges is smaller than |V| - 1, the value is then the number of MSF edges.

4.2 Benchmark Results

The results of the benchmark are presented in this subsection. We first list the graphics of the benchmarks and then provide a detailed analysis. The x-axis of the graphics is always the number of used threads, and y the speedup over the fastest sequential implementation for that particular graph. Note that the y-axis has an unusual scaling to enhance the contrast below the $y = 1$ line.

For a more precise reference, the running times for 1 and 144 threads are also tabulated after the graphics in Table 4.3 and Table 4.4. The relative and absolute speedups are given in Table 4.5.

4.2.1 Graphics and Tables for the Experiments

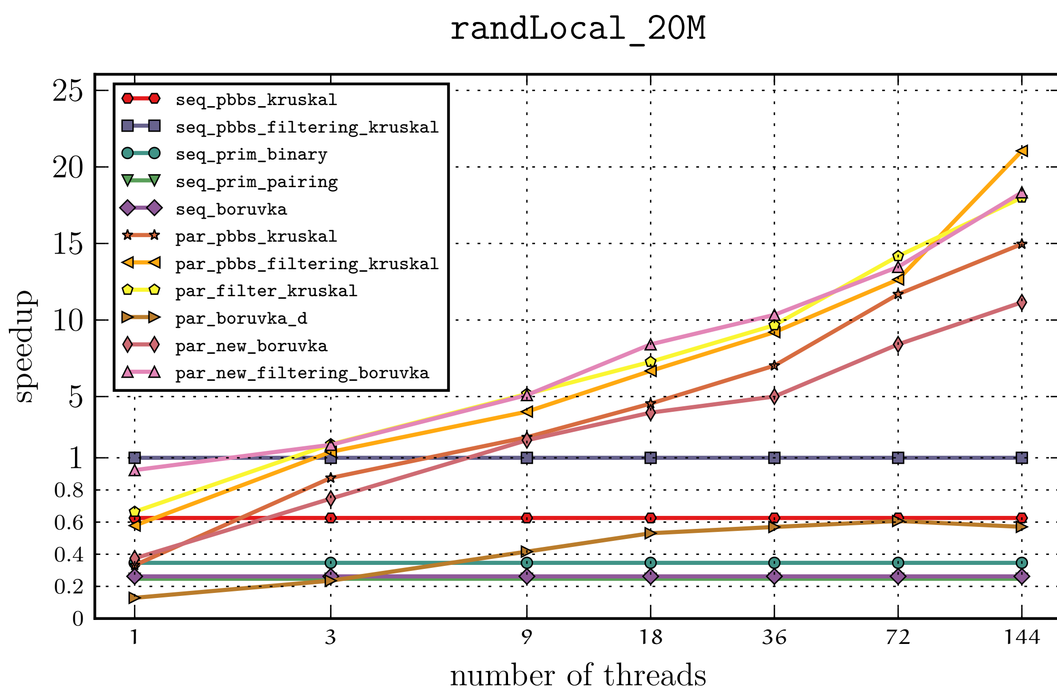


Figure 5: Benchmark results for `randLocal_20M` graph. The lines for `seq_boruvka` and `seq_prim_pairing` overlap.

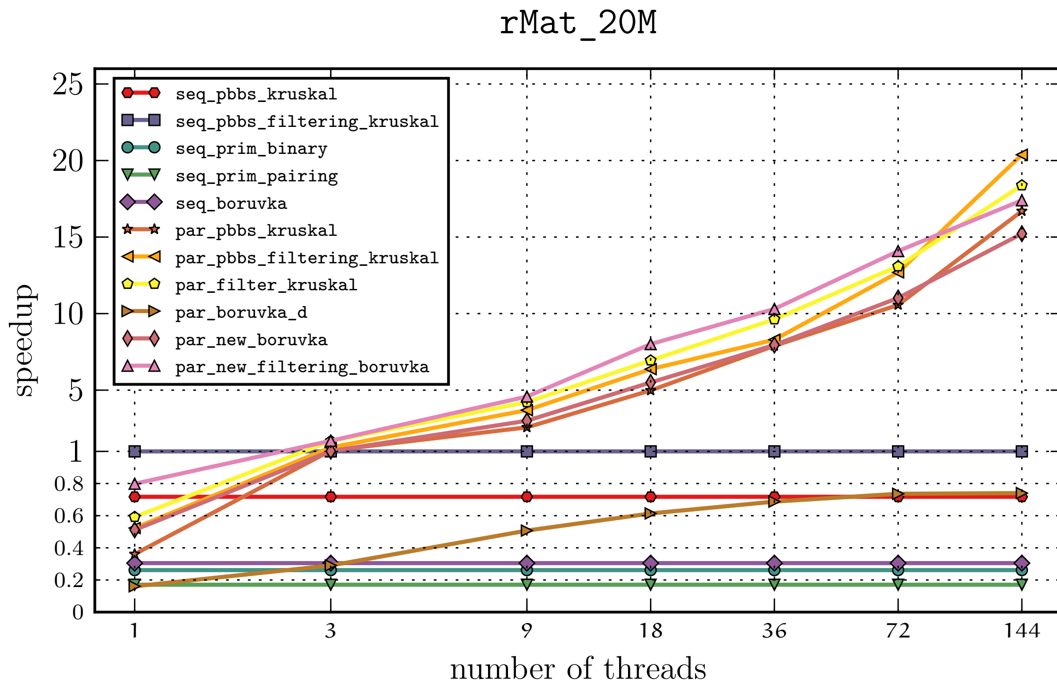


Figure 6: Benchmark results for `rMat_20M` graph.

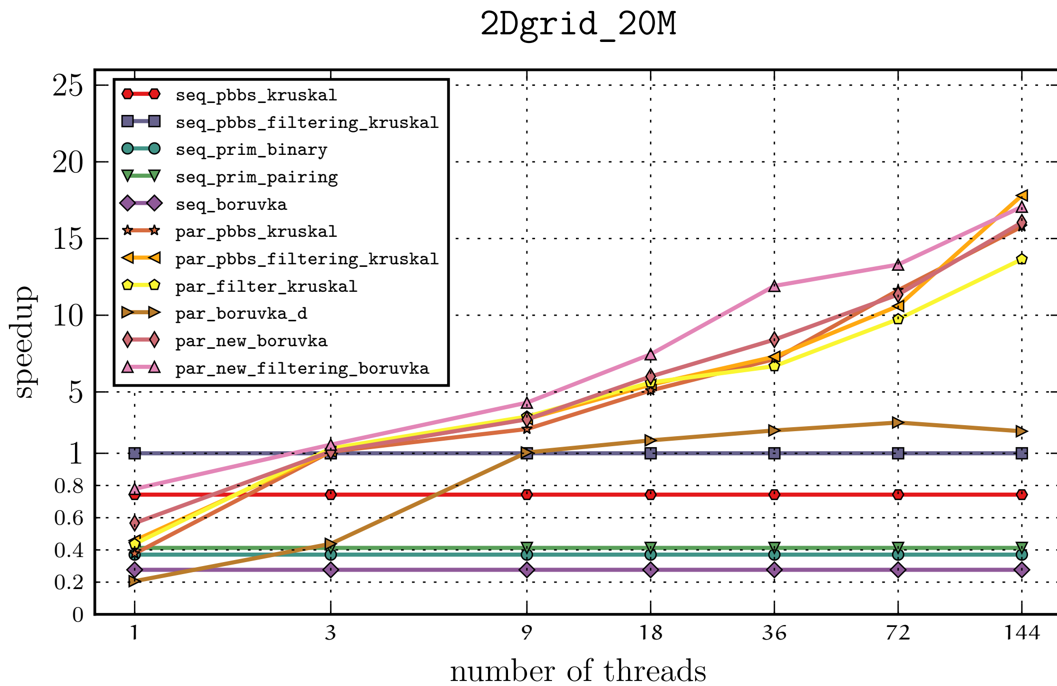


Figure 7: Benchmark results for `2Dgrid_20M` graph.

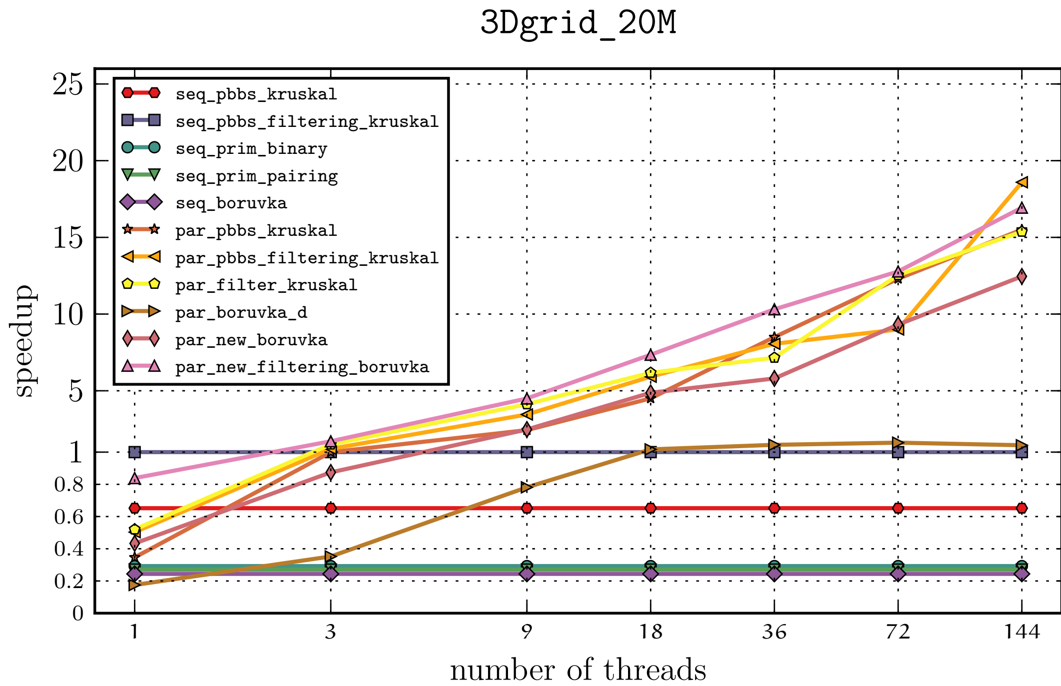


Figure 8: Benchmark results for `3Dgrid_20M` graph. The lines for `seq_boruvka`, `seq_prim_binary` and `seq_prim_pairing` are closely next to each other.

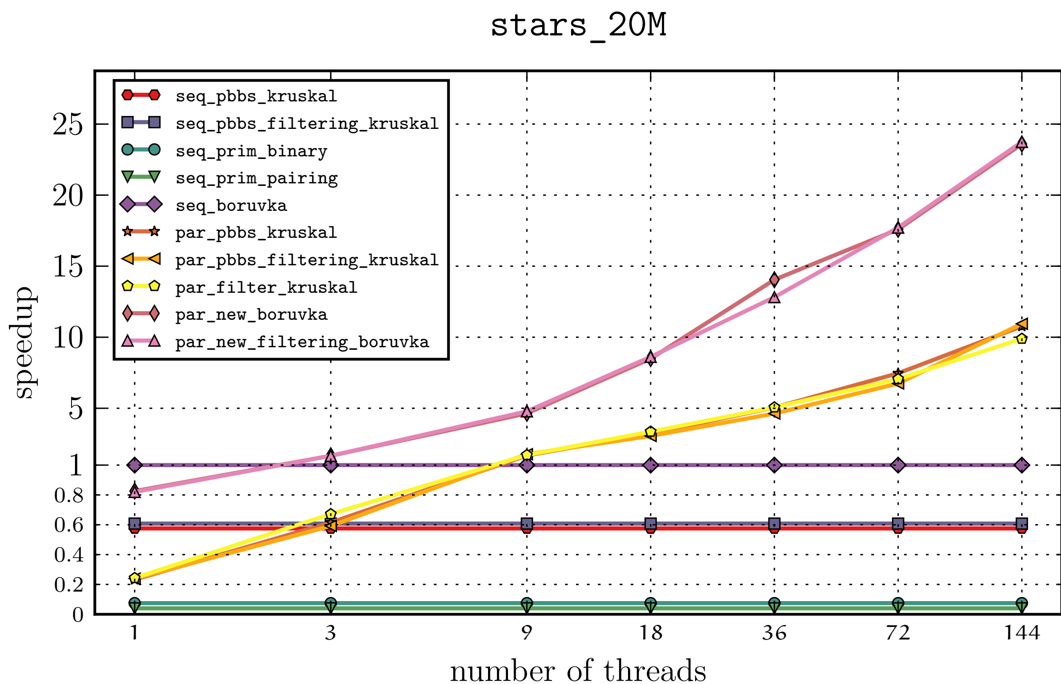


Figure 9: Benchmark results for `stars_20M` graph. Prim's algorithm performs very poorly on this graph because it needs $\mathcal{O}(n \log n)$ time to execute n `inserts` or `decrease-keys`. `Par_boruvka_d` is excluded because the code the authors provide online [42] seems to contain a bug and did not terminate for stars.

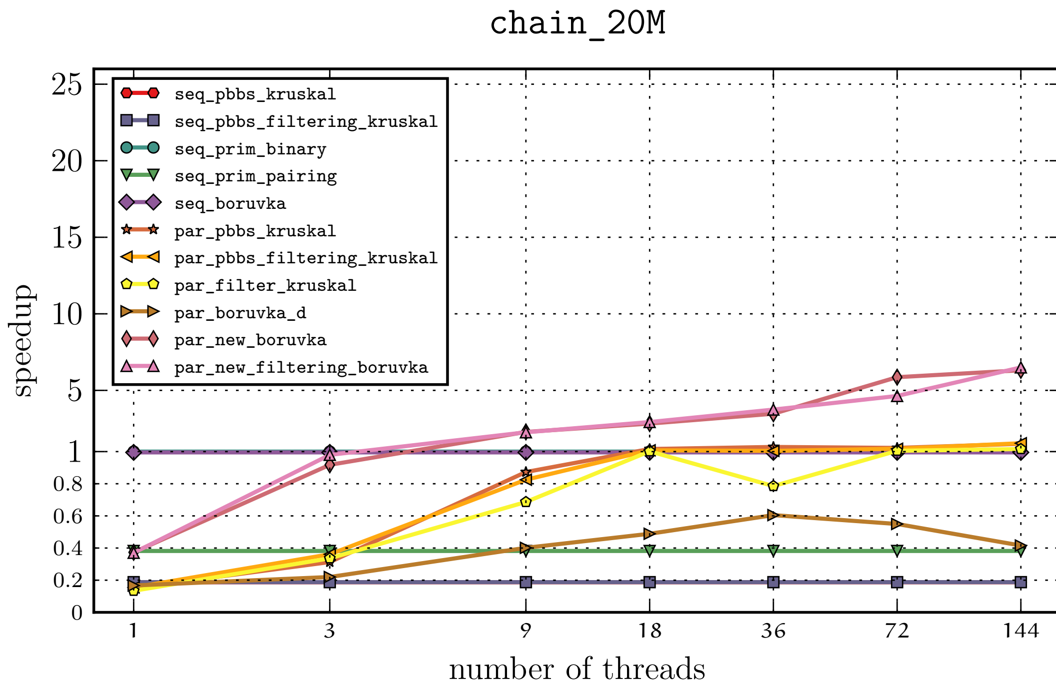


Figure 10: Benchmark results for `chain_20M` graph. Kruskal's algorithm works badly here. The lines for `seq_boruvka` and `seq_prim_binary` overlap and these for `seq_pbbs_kruskal` and `seq_pbbs_filtering_kruskal` overlap.

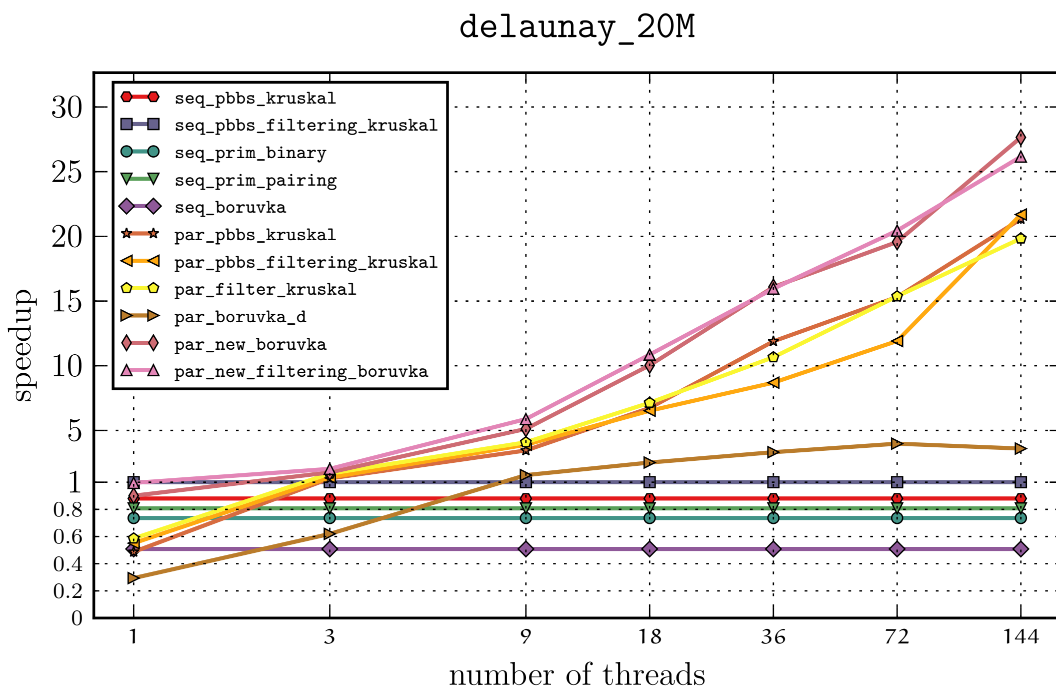


Figure 11: Benchmark results for `delaunay_20M` graph.

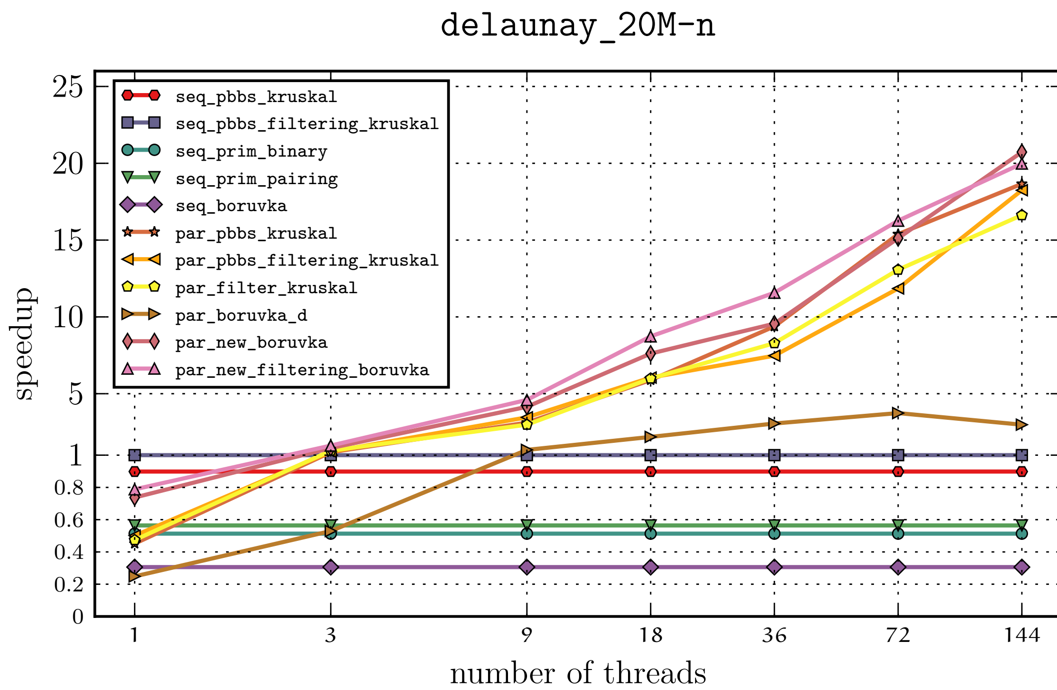


Figure 12: Benchmark results for `delaunay_20M-n` graph.

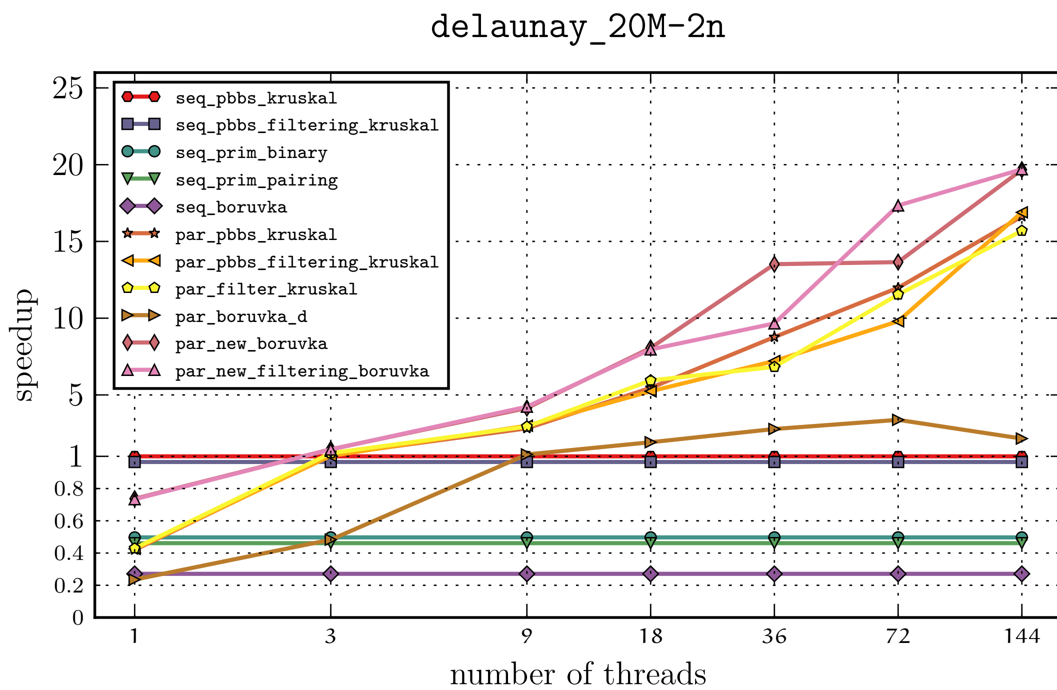


Figure 13: Benchmark results for `delaunay_20M-2n` graph. Kruskal's algorithm, with or without filtering, is the fastest sequential algorithm.

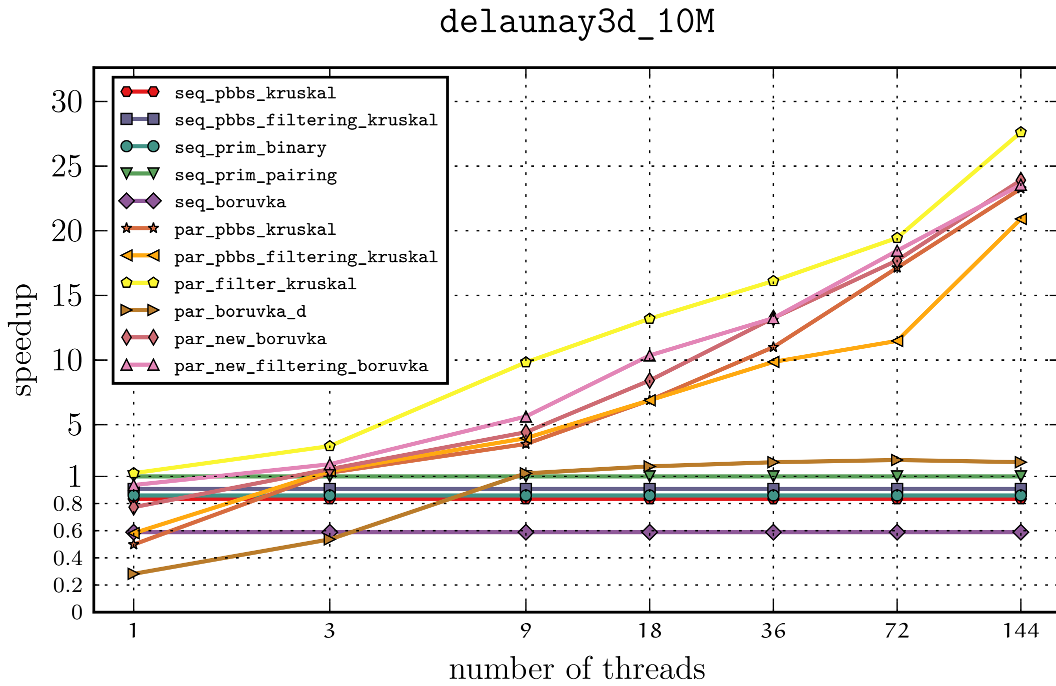


Figure 14: Benchmark results for `delaunay3d_10M` graph.

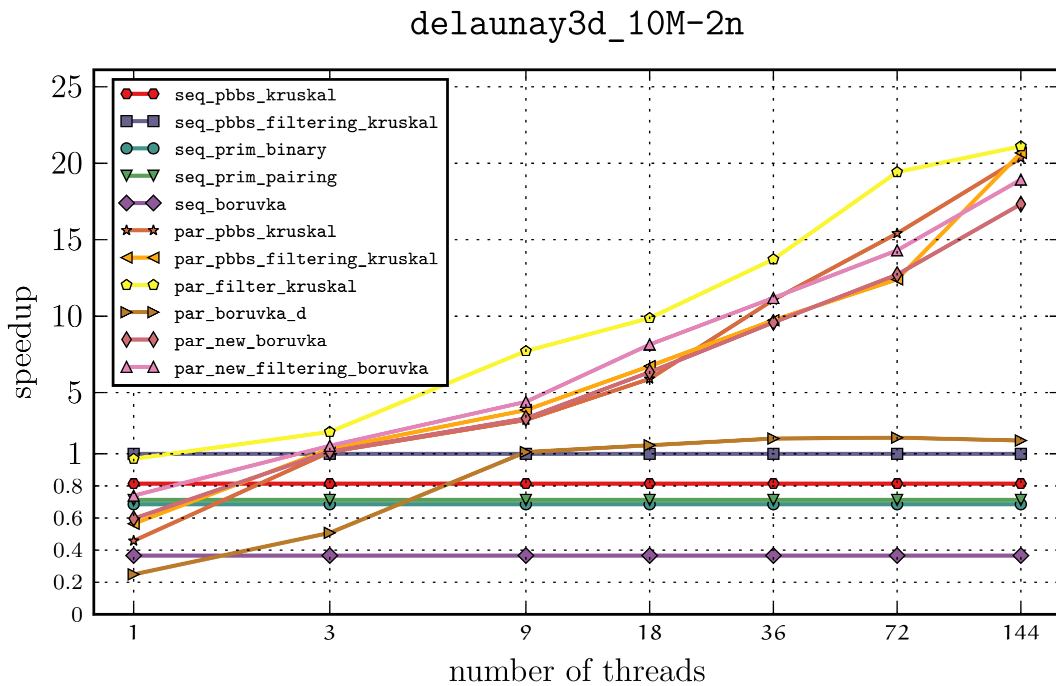


Figure 15: Benchmark results for `delaunay3d_10M-2n` graph.

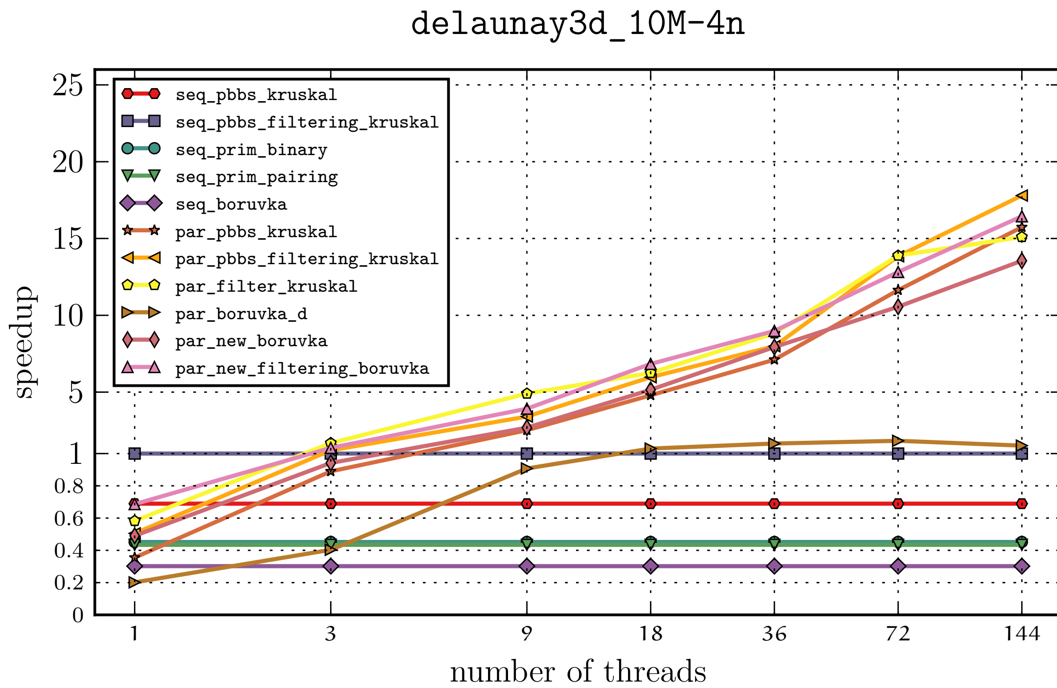


Figure 16: Benchmark results for `delaunay3d_10M-4n` graph.

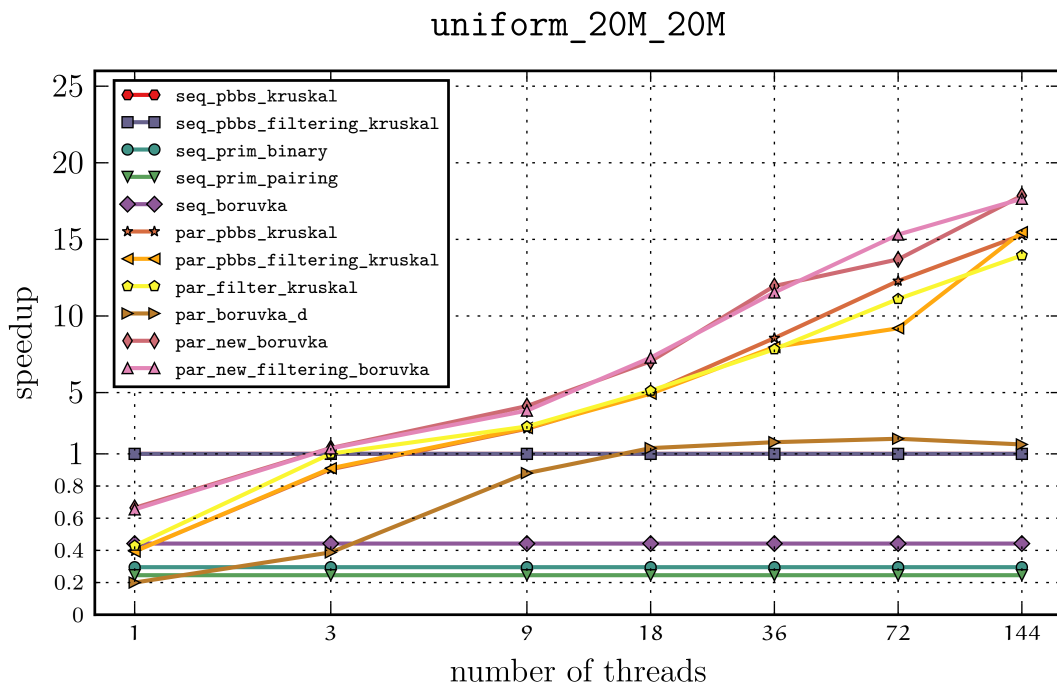


Figure 17: Benchmark results for `uniform_20M_20M` graph. Kruskal's algorithm with and without filtering share the same straight line as the fastest sequential algorithms.

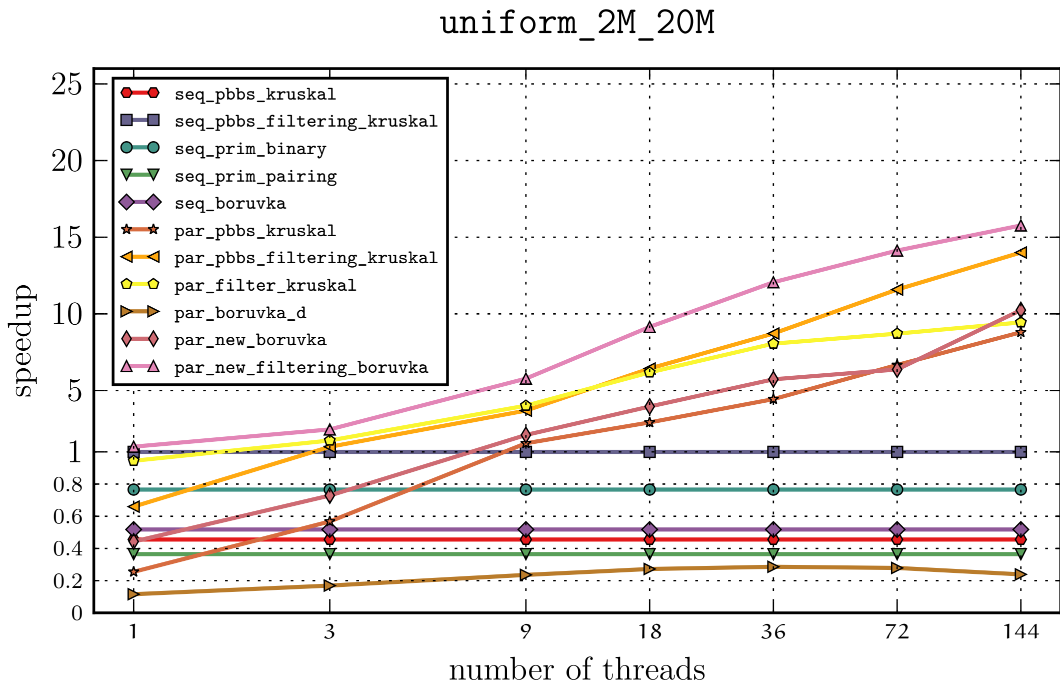


Figure 18: Benchmark results for `uniform_2M_20M` graph.

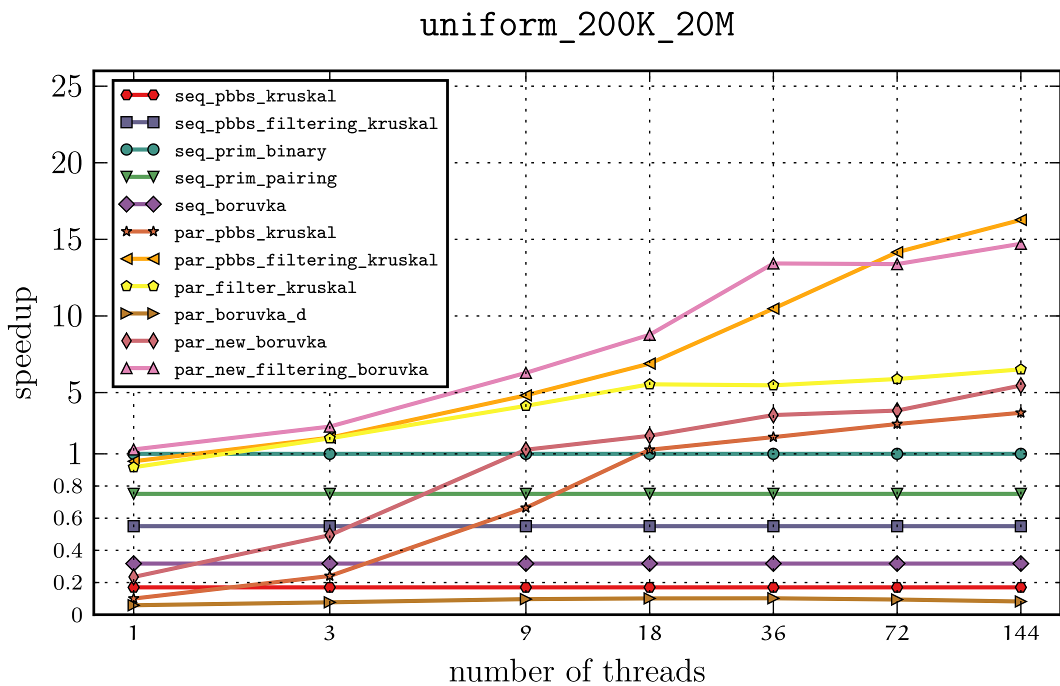


Figure 19: Benchmark results for `uniform_200K_20M` graph. `Seq_pbbs_kruskal` is slow for this graph because of the full sorting, but it works better with filtering.

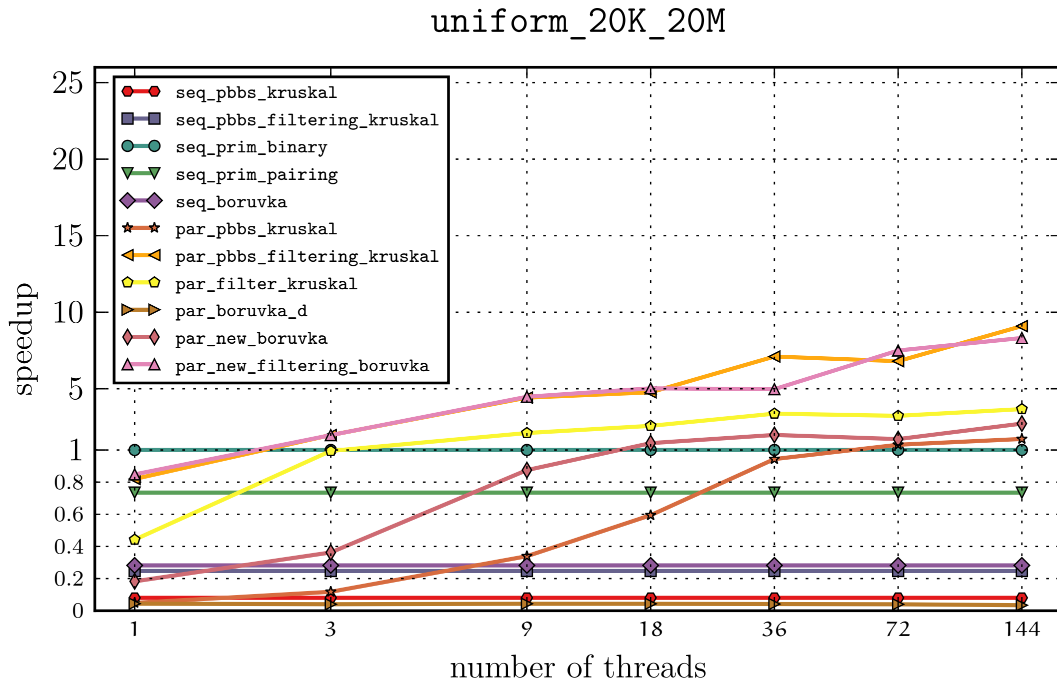


Figure 20: Benchmark results for `uniform_20K_20M` graph. `Seq_pbbs_kruskal` is slow for this graph because of the full sorting, but filtering remedies a bit.

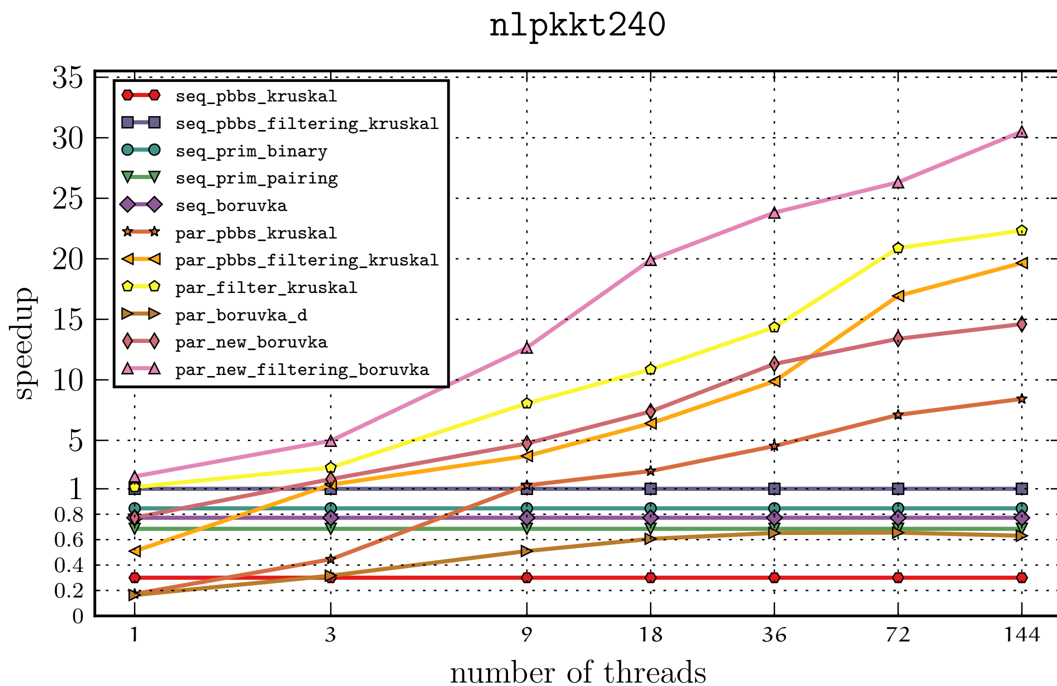


Figure 21: Benchmark results for `nlpkkt240` graph. Kruskal's algorithm without filtering is slow because of the full sorting. With filtering, it becomes the fastest sequential algorithm.

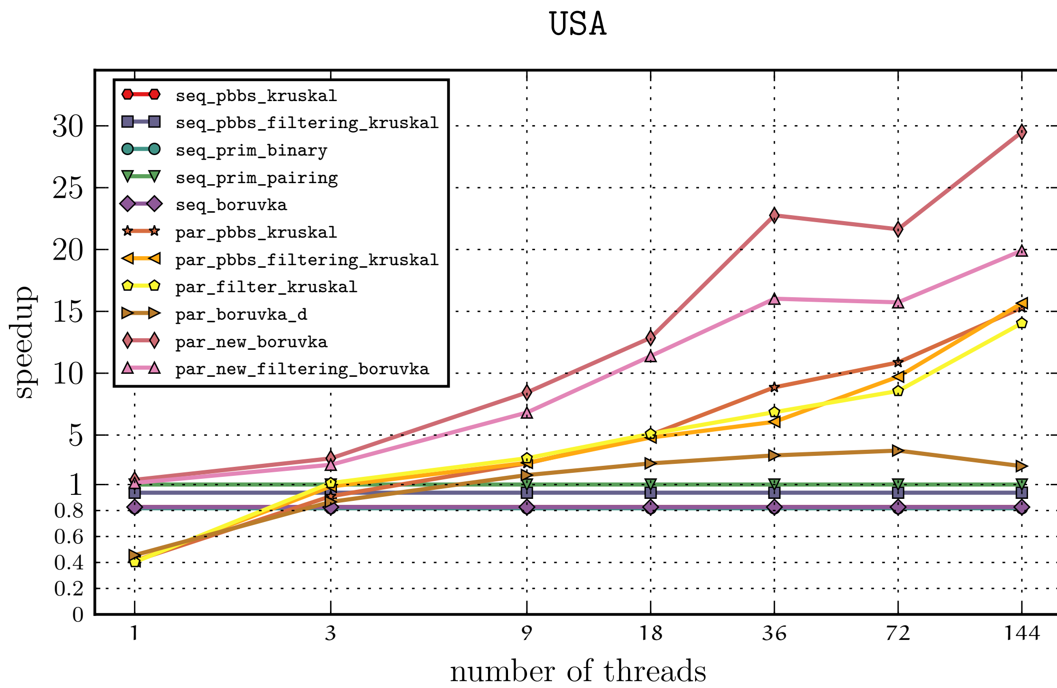


Figure 22: Benchmark results for `USA` graph. Lines for `seq_pbbs_kruskal`, `seq_boruvka` and `seq_prim_binary` overlap.

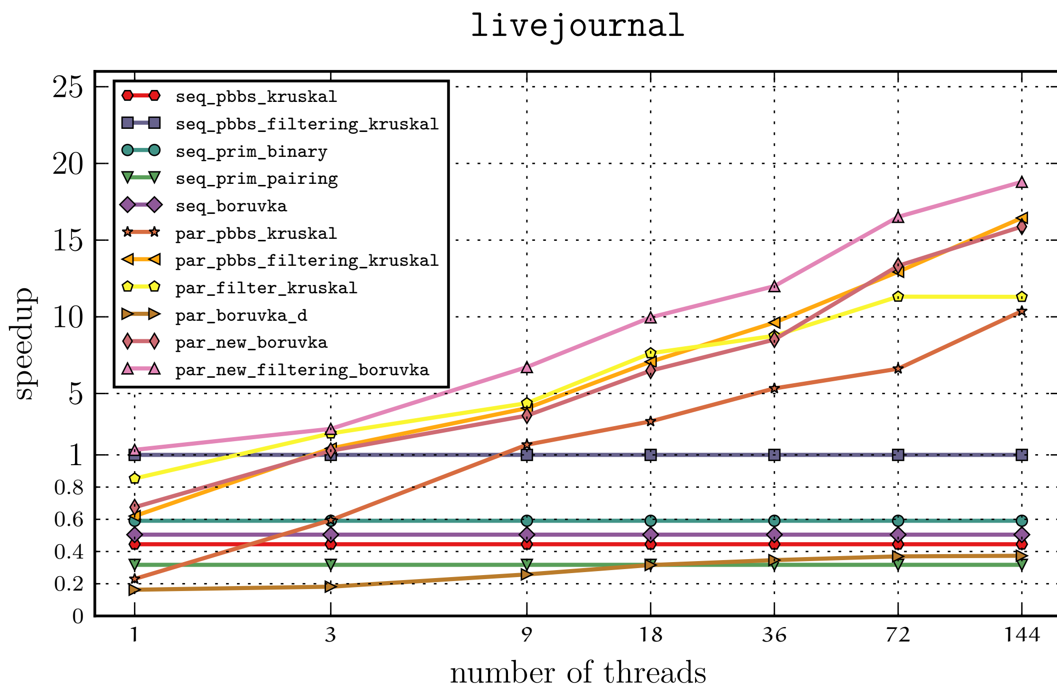


Figure 23: Benchmark results for `livejournal` graph.

	seq pbbs kruskal	seq pbbs filtering kruskal	seq prim binary	seq prim pairing	seq boruvka
randLocal_20M	18.300	11.450	33.041	46.291	43.638
rMat_20M	18.813	13.500	51.646	79.168	44.280
2Dgrid_20M	8.404	6.246	16.844	15.138	22.585
3Dgrid_20M	11.963	7.798	26.658	28.791	31.948
stars_20M	3.285	3.110	25.281	47.348	1.891
chain_20M	1.315	1.311	0.246	0.645	0.247
delaunay_20M	13.113	11.538	15.670	14.292	22.692
delaunay_20M-n	8.535	7.668	14.935	13.591	25.060
delaunay_20M-2n	3.930	4.073	7.905	8.519	14.501
delaunay3d_10M	14.375	13.213	13.943	11.987	20.353
delaunay3d_10M-2n	10.325	8.410	12.267	11.798	22.965
delaunay3d_10M-4n	6.828	4.708	10.417	10.844	15.587
uniform_20M_20M	3.700	3.700	12.520	15.063	8.365
uniform_2M_20M	2.986	1.360	1.776	3.729	2.627
uniform_200K_20M	3.050	0.942	0.519	0.690	1.632
uniform_20K_20M	2.730	0.888	0.219	0.298	0.778
nlpkkt240	152.125	45.700	53.995	66.692	59.181
USA	6.086	5.370	6.159	5.033	6.089
livejournal	5.459	2.430	4.112	7.656	4.815

Table 4.3: Running times of sequential algorithms in seconds. The best time for each particular graph is underlined.

	par pbbs kruskal	par pbbs filtering kruskal	par filter kruskal	par boruvka d	par new boruvka	par new filtering boruvka	par pbbs kruskal	par pbbs filtering kruskal	par filter kruskal	par boruvka d	par new boruvka	par new filtering boruvka
#threads	1	1	1	1	1	1	144	144	144	144	144	144
randLocal_20M	34.500	19.800	17.238	88.823	30.600	12.400	0.765	0.544	0.636	20.075	1.026	0.624
rMat_20M	37.400	25.800	22.788	83.999	26.400	16.900	0.809	0.663	0.735	18.229	0.887	0.776
2Dgrid_20M	16.563	13.625	14.288	30.317	11.025	8.019	0.396	0.351	0.458	2.564	0.389	0.366
3Dgrid_20M	22.513	15.500	14.988	44.499	18.000	9.296	0.503	0.419	0.507	5.382	0.626	0.460
stars_20M	8.143	7.890	7.779	–	2.290	2.305	0.176	0.173	0.191	–	0.080	0.080
chain_20M	1.623	1.590	1.836	1.475	0.660	0.665	0.163	0.159	0.211	0.592	0.039	0.038
delaunay_20M	23.688	20.938	19.688	39.243	12.813	11.600	0.541	0.532	0.582	3.207	0.417	0.441
delaunay_20M-n	16.988	15.300	16.188	30.872	10.400	9.730	0.411	0.420	0.461	2.570	0.370	0.384
delaunay_20M-2n	9.369	9.223	9.155	16.709	5.333	5.355	0.236	0.233	0.250	1.822	0.200	0.200
delaunay3d_10M	24.063	20.600	9.584	42.447	15.500	12.800	0.515	0.573	0.434	5.708	0.501	0.509
delaunay3d_10M-2n	18.363	14.900	8.680	33.764	14.100	11.375	0.413	0.407	0.398	4.506	0.485	0.444
delaunay3d_10M-4n	13.263	9.320	8.085	23.375	9.601	6.858	0.299	0.265	0.312	3.103	0.347	0.286
uniform_20M_20M	9.369	9.411	8.620	18.509	5.569	5.650	0.242	0.239	0.265	2.292	0.207	0.210
uniform_2M_20M	5.340	2.061	1.440	11.752	3.070	1.016	0.155	0.097	0.144	5.691	0.133	0.086
uniform_200K_20M	5.170	0.544	0.566	8.822	2.210	0.403	0.141	0.032	0.080	6.309	0.095	0.035
uniform_20K_20M	4.500	0.268	0.496	5.013	1.200	0.259	0.128	0.024	0.060	6.395	0.081	0.026
nlpkkt240	264.375	89.713	39.988	278.142	59.100	22.588	5.423	2.325	2.046	72.458	3.129	1.498
USA	12.213	12.300	12.488	11.025	3.645	4.448	0.329	0.321	0.358	2.022	0.171	0.253
livejournal	10.575	3.919	2.848	14.941	3.600	1.830	0.234	0.148	0.215	6.504	0.153	0.129

Table 4.4: Running times of parallel algorithms with 1 and 144 threads in seconds. Underlined numbers mark the best times for the particular graph and particular number of threads. If the time for a parallel algorithm executed with one thread is even faster than the best sequential algorithm, that time is doubly underlined.

	par pbbs kruskal	par pbbs filtering kruskal	par filter kruskal	par boruvka d	par new boruvka	par new filtering boruvka	par pbbs kruskal	par pbbs filtering kruskal	par filter kruskal	par boruvka d	par new boruvka	par new filtering boruvka
relative/absolute	rel	rel	rel	rel	rel	rel	abs	abs	abs	abs	abs	abs
randLocal_20M	45.091	36.405	27.098	4.425	29.817	19.864	14.965	21.053	18.000	0.570	11.157	18.342
rMat_20M	46.259	38.943	31.025	4.608	29.763	21.782	16.698	20.377	18.380	0.741	15.220	17.400
2Dgrid_20M	41.838	38.845	31.230	11.824	28.333	21.916	15.778	17.808	13.653	2.436	16.052	17.072
3Dgrid_20M	44.768	36.960	29.547	8.268	28.754	20.193	15.506	18.593	15.372	1.449	12.456	16.937
stars_20M	46.366	45.673	40.699	–	28.580	28.948	10.767	10.946	9.893	–	23.599	23.747
chain_20M	9.946	9.992	8.703	2.490	16.910	17.545	1.509	1.547	1.167	0.416	6.312	6.499
delaunay_20M	43.805	39.356	33.849	12.237	30.707	26.304	21.336	21.687	19.837	3.598	27.651	26.162
delaunay_20M-n	41.345	36.407	35.095	12.011	28.118	25.355	18.661	18.245	16.623	2.983	20.730	19.980
delaunay_20M-2n	39.635	39.645	36.565	9.170	26.696	26.842	16.626	16.894	15.696	2.157	19.675	19.699
delaunay3d_10M	46.689	35.943	22.082	7.437	30.915	25.154	23.259	20.915	27.620	2.100	23.908	23.556
delaunay3d_10M-2n	44.421	36.632	21.789	7.493	29.050	25.605	20.345	20.676	21.111	1.866	17.327	18.931
delaunay3d_10M-4n	44.338	35.236	25.933	7.533	27.639	23.956	15.738	17.798	15.100	1.517	13.552	16.445
uniform_20M_20M	38.714	39.357	32.513	8.077	26.870	26.953	15.289	15.473	13.956	1.615	17.853	17.651
uniform_2M_20M	34.563	21.223	10.000	2.065	23.126	11.783	8.803	14.003	9.444	0.239	10.245	15.768
uniform_200K_20M	36.699	17.059	7.105	1.398	23.263	11.429	3.682	16.272	6.514	0.082	5.460	14.714
uniform_20K_20M	35.294	11.093	8.308	0.784	14.907	9.806	1.721	9.093	3.671	0.034	2.725	8.317
nlpkkt240	48.755	38.586	19.543	3.839	18.889	15.083	8.428	19.656	22.335	0.631	14.606	30.518
USA	37.120	38.273	34.857	5.452	21.363	17.596	15.298	15.661	14.049	2.489	29.498	19.914
livejournal	45.144	26.523	13.244	2.297	23.529	14.159	10.374	16.447	11.302	0.374	15.882	18.801

Table 4.5: Relative and absolute speedups of parallel algorithms. Underlined numbers mark the best relative and absolute speed-ups for the particular graph with 144 threads.

4.2.2 Analysis for Sequential Algorithms

For sequential algorithms, `seq_pbbs_filtering_kruskal` is almost always the fastest. Notable exceptions include:

- `stars_20M` (Figure 9). Borůvka's algorithm only needs a single Borůvka step here because the leaves of the star will choose their only outgoing edge as the minimum edge, connecting the graph in one step. Thus Borůvka's algorithm runs in linear time here. Kruskal's algorithm needs a full sorting of random weights, and Prim's algorithm has to perform n `decrease-key`s or `insert`s, therefore both algorithms have a running time of $\Theta(n \log n)$. Note that $m = \mathcal{O}(n)$ for stars. The highly optimized sorting implementation from the standard library of GCC has a much lower constant factor, making Kruskal's algorithm considerably faster than Prim's algorithm and even comparable to the linear-time Borůvka's algorithm here. The case would be slightly different if Prim's algorithm first builds a priority queue in linear time and then calls n constant-time `decrease-key`s. However, a large constant factor would be inevitable.
- `chain_20M` (Figure 10), where `seq_boruvka` and `seq_prim_binary` are equally fast. On chains with *increasing weights*, Borůvka's algorithm finds the MST in a single Borůvka step and requires only linear time. Prim's algorithm is also fast because the priority queue only has a single element in it during the whole execution. `Seq_prim_pairing` is slower because of the hidden constants in its time complexity. Kruskal's algorithm needs a sorting and is thus slower. However, because the edge weights are already in increasing order, the sorting is much cheaper than sorting random numbers, which is why its running time is less than half of that for `stars_20M`.
- Dense uniform graphs like `uniform_200K_20M` (Figure 19) and `uniform_20K_20M` (Figure 20). That is because the number of vertices is too small compared to the number of edges, which makes sorting of the edges too costly. Prim's algorithm excels here because it does not even have to process all the edges before finding the MST. As mentioned before in Section 2.1.2, Prim's algorithm with binary heaps has an expected running time of $\mathcal{O}(m + n \log n \log \log n)$ for graphs with random weights, which is $\mathcal{O}(m)$ for denser graphs like these two.

On real-world graphs like `nlpkkt240` (Figure 21) and `livejournal` (Figure 23), the effect of filtering is well demonstrated because vertices have a larger average degree and hence the graphs are relatively dense. Sorting the whole set of edges is too costly, but sorting only a portion and then doing a filtering reduces the total time to only one third to one half.

It is also interesting to note that Prim's algorithm with pairing heaps is generally slower than with binary heaps except on several graphs. That is because the effect of a faster `decrease-key` is only visible when it is needed many times due to larger constant hidden in the Big-Oh notation.

Due to the lack of removal of used edges, Borůvka's algorithm generally does not work well with the sole exceptions of `stars_20M` (Figure 9) and `chain_20M` (Figure 10) where the Borůvka step is executed a single time anyway. However, with filtering and removal of self-edges, the new *parallel* algorithm works quite well with one thread. That is because the algorithm only does linear work in each Borůvka step when it is executed with a single thread. As shown in Table 4.4, the new parallel algorithm with or without filtering even beats the fastest sequential algorithm on all three real-world graphs and two uniform graphs. On other graphs, `par_new_boruvka` and `par_new_filtering_boruvka` also have comparable performance to the

fastest sequential implementation. This suggests the new algorithm can be applied even when the number of available processors is small. Other algorithms do not have such a property. The only graphs where other parallel algorithms perform better with one thread are `delaunay3d_10M` (Figure 14) and `delaunay3d_10M-2n` (Figure 15), where `par_filter_kruskal` excels because its multi-stage filtering removes most of the edges.

4.2.3 Analysis for Parallel Algorithms

We now turn to the analysis of the running times of parallel algorithms.

`par_boruvka_d` [42] does not perform well on our test graphs. On `randLocal_20M` (Figure 5), `rMat_20M` (Figure 6), `nlpkkt240` (Figure 21) and `livejournal` (Figure 23), the parallel algorithm with all available cores even runs longer than most of the sequential algorithms. Reading through the detailed logs reveals that more than $\frac{2}{3}$ of the time is spent on the compaction step where atomic instructions are used in an unavoidably inefficient manner as described in Section 2.3.2, in contrast to our new parallel algorithm where most of the atomic instructions are only reads rather than writes. Even on graphs where `par_boruvka_d` exhibits speedup against sequential algorithms, the speedup comes quite late, namely when at least 9 cores are used where 3 are enough for other algorithms most of the times. Even when a speedup is visible, the algorithm is not as fast as other algorithms. For example, on graph `USA` (Figure 22), the algorithm needs 1.344 seconds with 72 cores, but all other algorithms are at least twice as fast. Its bottlenecks seem to be the compaction and, surprisingly, a step that marks MST edges. The latter involves a parallel for-loop of length n with one random read and one random write memory access in each Borůvka iteration.

The effect of filtering manifested itself also in the parallel settings: Borůvka's algorithm and Kruskal's algorithm with filtering are generally faster than without. The only notable exceptions are `delaunay3d_10M` (Figure 14) for Kruskal's algorithm and `USA` (Figure 22) for Borůvka's. The former is because the filtering only removes about half of the edges and thus does not justify the extra cost for partitioning and filtering. The latter is because `USA` is very sparse so that the first partition already contains most of the edges. Therefore the cost for this partitioning does not pay off.

As shown in the graphics, `par_new_filtering_boruvka` or `par_new_boruvka` are the fastest parallel implementation on more than half of the graphs, especially on `stars_20M` (Figure 9) and `chain_20M` (Figure 10), where the difference is large due to the reasons described above in the sequential part. On real-world graphs, i.e. `nlpkkt240` (Figure 21), `USA` (Figure 22) and `livejournal` (Figure 23), the new algorithm leads by a remarkable margin and reaches an absolute speedup of more than 30. The sole exceptions where the new algorithm loses by a noticeable margin are `delaunay3d_10M` (Figure 14) and `delaunay3d_10M-2n` (Figure 15). `Par_filter_kruskal` is faster on these graphs due to the random weights and the graph topology. The progressive multi-stage filtering of `par_filter_kruskal` works in its full ability and removes most of the edges. `Par_pbbs_filtering_kruskal` only does one filtering and cannot achieve the same effect.

A remark on `par_pbbs_filtering_kruskal` and `par_pbbs_kruskal` is that they even need slightly more time on the union-find loop than the sequential implementation on `chain_20M` (Figure 10) because the algorithm is forced to run sequentially after the sorting step by the structure and weights of the graph. Though this behavior would unlikely cause problems on

graphs of practical size, it is of theoretical interest to note that this implementation can be forced to run sequentially by an adversary. The remarkable relative speedups shown in Table 4.5 are due to the highly efficient sample sort implementation in PBBS and the *slowness* of the algorithm with one thread.

The experiments have demonstrated that our new algorithm, together with filtering, is indeed efficient on a wide range of graphs. Unlike other parallel algorithms, the algorithm is also very efficient when only a small number of processors are available. The same even holds when only a single processor is present.

5 Conclusions and Outlooks

In the present thesis we have briefly described known sequential and parallel algorithms for computing minimum spanning trees (MST) and forests (MSF) on shared-memory architectures. We also have presented a new conceptually quite simple yet remarkably efficient parallel algorithm for MST/MSF computation based on Borůvka's algorithm. The algorithm utilizes priority writes (`pwrite`) as a primitive to achieve its simplicity and reduced contention. `pwrite`s can be easily and efficiently implemented with atomic compare-and-swap (CAS) which is widely supported by modern processors. Coarse-grained and balanced parallelism is realized nicely this way. Several optimizations that aim at improving locality of memory accesses are applied to achieve further speedup.

Experimental results on a rich set of synthetic and real-world graphs have demonstrated the extraordinary efficiency of the new algorithm. The new algorithm is faster than or as fast as its rivals on almost every graph in the benchmark. A reasonable speedup with respect to classical sequential algorithms is achieved even with only a few processors. The parallel algorithm outperforms many sequential implementations even with a single processor.

Though the algorithm proved to be efficient on many graphs, there is still room for improvement and future work. The following list gives some possible directions for future research:

Outlook 1. Theoretical time bound of the find-min step. The algorithm exhibits simplicity and excellent performance in practice. However, not much is known about its theoretical efficiency. Though we conjecture stars are the worst case for the find-min step, it remains to be carefully analyzed.

Outlook 2. Applying priority writes to other algorithms or problems. Priority writes (`pwrite`s) are the key to the efficiency of the new MST algorithm. They seems to have the potential to be utilized to implement some known algorithms for the CRCW-Priority PRAM model on real computers. Whether this is possible or beneficial stays open for future research. It is also conceivable that they can be used in other algorithms that involve finding some minimum across multiple processors. It would be interesting to see more such examples.

Outlook 3. More thorough experiments. We have conducted benchmarks with a number of classical algorithms and state-of-the-art parallel ones. The new algorithm works very well on a wide range of test graphs. Yet the experiments did not, and could not, cover all known algorithms. An interesting future work might be a more thorough benchmark that includes more algorithms and more graphs.

Outlook 4. Generalization to other architectures. The algorithm is targeted at shared-memory architectures and achieves good performance there. It would be natural to ask whether it can be generalized to other architectures like distributed-memory ones. At first glance it is not obvious how this can be done because synchronization is harder and more expensive on these architectures and atomic instructions are virtually nonexistent.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. (Pages 1, 3, 7, 10, 11, 12).

[2] J. Shun, L. Dhulipala, and G. Blelloch, "A simple and practical linear-work parallel algorithm for connectivity," in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures.* ACM, 2014, pp. 143–153. (Page 2).

[3] R. Motwani and P. Raghavan, *Randomized algorithms.* Chapman & Hall/CRC, 2010. (Page 3).

[4] O. Borůvka, "O jistém problému minimálním (About a certain minimal problem) (in Czech and German)," *Práce Mor. Prírodoved. Spol. V Brne III*, vol. 3, pp. 35–78, 1926. (Page 3).

[5] O. Borůvka, "Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí (Contribution to the solution of a problem of economical construction of electrical networks) (in Czech)," *Elektrotechnickỳ obzor*, vol. 15, pp. 153–154, 1926. (Page 3).

[6] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985. (Pages 3, 7).

[7] A. C.-C. Yao, "An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees," *Information Processing Letters*, vol. 4, no. 1, pp. 21–23, 1975. (Pages 4, 10).

[8] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Labs Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957. (Page 7).

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959. (Page 7).

[10] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987. (Pages 7, 12).

[11] H. Kaplan and R. E. Tarjan, "New heap data structures," Technical Report TR-597-99, Department of Computer Science, Princeton University, Tech. Rep., 1999. (Page 7).

[12] B. M. Moret and H. D. Shapiro, "An empirical analysis of algorithms for constructing a minimum spanning tree," in *Workshop on Algorithms and Data Structures.* Springer, 1991, pp. 400–411. (Pages 7, 8, 11).

[13] C. F. Bazlamaçcı and K. S. Hindi, "Minimum-weight spanning tree algorithms a survey and empirical study," *Computers & Operations Research*, vol. 28, no. 8, pp. 767–785, 2001. (Pages 7, 8).

[14] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The pairing heap: A new form of self-adjusting heap," *Algorithmica*, vol. 1, no. 1-4, pp. 111–129, 1986. (Page 7).

[15] M. L. Fredman, "On the efficiency of pairing heaps and related data structures," *Journal of the ACM (JACM)*, vol. 46, no. 4, pp. 473–501, 1999. (Page 7).

[16] A. Elmasry, "Pairing heaps with O(log log n) decrease cost," in *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms.* SIAM, 2009, pp. 471–476. (Page 7).

[17] C. Martel, "The expected complexity of Prim's minimum spanning tree algorithm," *Information processing letters*, vol. 81, no. 4, pp. 197–201, 2002. (Page 7).

[18] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956. (Pages 8, 14).

[19] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984. (Page 8).

[20] R. E. Tarjan, *Data structures and network algorithms.* SIAM, 1983. (Pages 10, 11).

[21] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 724–742, 1976. (Pages 10, 11, 13).

[22] C. A. Crane, "Linear lists and priority queues as balanced binary trees," Ph.D. dissertation, Stanford, CA, USA, 1972. (Page 11).

[23] P. Erdős and A. Rényi, "On random graphs, I," *Publicationes Mathematicae (Debrecen)*, vol. 6, pp. 290–297, 1959. (Page 11).

[24] V. Osipov, P. Sanders, and J. Singler, "The filter-Kruskal minimum spanning tree algorithm," in *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX).* SIAM, 2009, pp. 52–61. (Pages 12, 29).

[25] R. Paredes and G. Navarro, "Optimal incremental sorting," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments.* Society for Industrial and Applied Mathematics, 2006, pp. 171–182. (Page 12).

[26] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 181–192. (Pages 12, 18, 23).

[27] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica*, vol. 6, no. 2, pp. 109–122, 1986. (Page 13).

[28] B. Chazelle, "A minimum spanning tree algorithm with inverse-ackermann type complexity," *Journal of the ACM (JACM)*, vol. 47, no. 6, pp. 1028–1047, 2000. (Page 13).

[29] B. Chazelle, "The soft heap: An approximate priority queue with optimal error rate," *J. ACM*, vol. 47, no. 6, pp. 1012–1027, Nov. 2000. (Page 13).

[30] S. Pettie, "Finding minimum spanning trees in O(mα(m, n)) time," 1999. (Page 13).

[31] S. Pettie and V. Ramachandran, "An optimal minimum spanning tree algorithm," *Journal of the ACM (JACM)*, vol. 49, no. 1, pp. 16–34, 2002. (Page 13).

[32] M. L. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Journal of Computer and System Sciences*, vol. 48, no. 3, pp. 533–551, 1994. (Page 13).

[33] D. R. Karger, P. N. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM (JACM)*, vol. 42, no. 2, pp. 321–328, 1995. (Page 14).

[34] V. King, "A simpler minimum spanning tree verification algorithm," in *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, ser. WADS '95. London, UK, UK: Springer-Verlag, 1995, pp. 440–448. (Page 14).

[35] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook, "Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing.* ACM, 1998, pp. 279–288. (Page 14).

[36] I. Katriel, P. Sanders, and J. L. Träff, "A practical minimum spanning tree algorithm using the cycle property," in *European Symposium on Algorithms.* Springer, 2003, pp. 679–690. (Page 14).

[37] M. Thorup, "Near-optimal fully-dynamic graph connectivity," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing.* ACM, 2000, pp. 343–350. (Page 14).

[38] J. JáJá, *An introduction to parallel algorithms.* Addison-Wesley Reading, 1992, vol. 17. (Pages 15, 16, 18, 24, 25).

[39] G. E. Blelloch, "Prefix sums and their applications," 1990. (Page 16).

[40] W. D. Frazer and A. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM (JACM)*, vol. 17, no. 3, pp. 496–507, 1970. (Page 16).

[41] D. A. Bader and G. 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. (Pages 17, 18).

[42] C. da Silva Sousa, A. Mariano, and A. Proença, "A generic and highly efficient parallel variant of Borůvka's algorithm," in *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).* IEEE, 2015, pp. 610–617. [Online]. Available: https://github.com/Beatgodes/BoruvkaUMinho (Pages 17, 20, 30, 35, 47).

[43] G. Cong and I. Tanase, "Composable locality optimizations for accelerating parallel forest computations," in *High Performance Computing and Communications.* IEEE, 2016, pp. 190–197. (Pages 18, 27, 28).

[44] G. Cong, I. Tanase, and Y. Xia, "Accelerating minimum spanning forest computations on multicore platforms," in *European Conference on Parallel Processing.* Springer, 2015, pp. 541–552. (Page 18).

[45] A. Katsigiannis, N. Anastopoulos, K. Nikas, and N. Koziris, "An approach to parallelize kruskal's algorithm using helper threads," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 1601–1610. (Page 20).

[46] B. Awerbuch and Y. Shiloach, "New connectivity and msf algorithms for shuffle-exchange network and pram," *IEEE Transactions on Computers*, vol. 36, no. 10, pp. 1258–1263, 1987. (Pages 20, 21).

[47] C. D. Zaroliagis, "Simple and work-efficient parallel algorithms for the minimum spanning tree problem," *Parallel processing letters*, vol. 7, no. 01, pp. 25–37, 1997. (Pages 20, 21).

[48] V. Vineet, P. Harish, S. Patidar, and P. Narayanan, "Fast minimum spanning tree for large graphs on the GPU," in *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 2009, pp. 167–171. (Page 20).

[49] W. Wang, S. Guo, F. Yang, and J. Chen, "GPU-based fast minimum spanning tree using data parallel primitives," in *2nd International Conference on Information Engineering and Computer Science (ICIECS)*. IEEE, 2010, pp. 1–4. (Page 20).

[50] W. Wang, Y. Huang, and S. Guo, "Design and implementation of GPU-based prim's algorithm," *International Journal of Modern Education and Computer Science*, vol. 3, no. 4, p. 55, 2011. (Page 20).

[51] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 205–214. (Page 20).

[52] S. Chung and A. Condon, "Parallel implementation of Borůvka's minimum spanning tree algorithm," in *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*. IEEE, 1996, pp. 302–308. (Page 20).

[53] F. Dehne and S. Götz, "Practical parallel algorithms for minimum spanning trees," in *Seventeenth IEEE Symposium on Reliable Distributed Systems*. IEEE, 1998, pp. 366–371. (Page 20).

[54] V. Lončar, S. Škrbić, and A. Balaž, "Parallelization of minimum spanning tree algorithms using distributed memory architectures," in *Transactions on Engineering Technologies*. Springer, 2014, pp. 543–554. (Page 20).

[55] F. Y. Chin, J. Lam, and I.-N. Chen, "Efficient parallel algorithms for some graph problems," *Communications of the ACM*, vol. 25, no. 9, pp. 659–665, 1982. (Page 21).

[56] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 478–491. (Page 21).

[57] D. R. Karger, "Approximating, verifying, and constructing minimum spanning forests," 1992. (Page 21).

[58] D. B. Johnson and P. Metaxas, "A parallel algorithm for computing minimum spanning trees," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1992, pp. 363–372. (Page 21).

[59] R. Cole, P. N. Klein, and R. E. Tarjan, *Linear-work Parallel Algorithm for Finding Minimum Spanning Trees*. Princeton University, Department of Computer Science, 1994. (Page 21).

[60] R. Cole, P. N. Klein, and R. E. Tarjan, "Finding minimum spanning forests in logarithmic time and linear work using random sampling," in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1996, pp. 243–250. (Page 21).

[61] C. K. Poon and V. Ramachandran, "A randomized linear work erew pram algorithm to find a minimum spanning forest," in *International Symposium on Algorithms and Computation*. Springer, 1997, pp. 212–222. (Page 21).

[62] S. Pettie and V. Ramachandran, "A randomized time-work optimal parallel algorithm for finding a minimum spanning forest," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1879–1895, 2002. (Page 21).

[63] K. W. Chong, Y. Han, and T. W. Lam, "Concurrent threads and optimal parallel minimum spanning trees algorithm," *Journal of the ACM (JACM)*, vol. 48, no. 2, pp. 297–323, 2001. (Page 21).

[64] K. W. Chong, Y. Han, Y. Igarashi, and T. W. Lam, "Improving the efficiency of parallel minimum spanning tree algorithms," *Discrete Applied Mathematics*, vol. 126, no. 1, pp. 33–54, 2003. (Page 21).

[65] Intel Corporation, *Intel ®64 and IA-32 Architectures Software Developer's Manual*, 2013. (Page 23).

[66] Advanced Micro Devices Inc., *AMD64 Architecture Programmer's Manual*, 2013, vol. 3: General-Purpose and System Instructions. (Page 23).

[67] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons, "Reducing contention through priority updates," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013, pp. 152–163. (Pages 23, 24).

[68] R. Sedgewick and P. Flajolet, *An introduction to the analysis of algorithms*. Addison-Wesley, 2013. (Page 23).

[69] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 68–70. (Page 29).

[70] A. Tavory, V. Dreizin, and B. Kosnik, "Policy-based data structures," 2004. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/ (Page 30).

[71] C. E. Leiserson, "The Cilk++ concurrency platform," in *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*. IEEE, 2009, pp. 522–527. (Page 30).

[72] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446. (Page 30).

[73] R. J. Wilson, *An introduction to graph theory*. Pearson Education India, 1970. (Page 30).

[74] The MathWorks, Inc., "Matlab R2017a," 2017. [Online]. Available: http://www.mathworks.com (Page 30).

[75] Wolfram Research, Inc., "Mathematica 11.0," 2016. [Online]. Available: http://www.wolfram.com/ (Page 31).

[76] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011. (Page 31).

[77] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS implementation challenge–shortest paths," *American Mathematical Society*, 2006. (Page 31).

[78] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014. (Page 31).