Optimal Parallel Algorithms for Dendrogram Computation and Single-Linkage Clustering

Laxman Dhulipala
University of Maryland
College Park, MD, USA
laxman@umd.edu

Xiaojun Dong
University of California
Riverside, CA, USA
xdong038@ucr.edu

Kishen N Gowda
University of Maryland
College Park, MD, USA
kishen19@cs.umd.edu

Yan Gu
University of California
Riverside, CA, USA
ygu@cs.ucr.edu

ABSTRACT

Computing a Single-Linkage Dendrogram (SLD) is a key step in the classic single-linkage hierarchical clustering algorithm. Given an input edge-weighted tree T, the SLD of T is a binary dendrogram that summarizes the $n-1$ clusterings obtained by contracting the edges of T in order of weight. Existing algorithms for computing the SLD all require $\Omega(n \log n)$ work where $n = |T|$. Furthermore, to the best of our knowledge no prior work provides a parallel algorithm obtaining non-trivial speedup for this problem.

In this paper, we design faster parallel algorithms for computing SLDs both in theory and in practice based on new structural results about SLDs. In particular, we obtain a deterministic output-sensitive parallel algorithm based on parallel tree contraction that requires $O(n \log h)$ work and $O(\log^2 n \log^2 h)$ depth, where h is the height of the output SLD. We also give a deterministic bottom-up algorithm for the problem inspired by the nearest-neighbor chain algorithm for hierarchical agglomerative clustering, and show that it achieves $O(n \log h)$ work and $O(h \log n)$ depth. Our results are based on a novel divide-and-conquer framework for building SLDs, inspired by divide-and-conquer algorithms for Cartesian trees. Our new algorithms can quickly compute the SLD on billion-scale trees, and obtain up to 150x speedup over the highly-efficient Union-Find algorithm typically used to compute SLDs in practice.

1 INTRODUCTION

Single-linkage clustering is a fundamental technique in unsupervised learning and data mining that groups objects based on a similarity (or dissimilarity) function [36]. The single-linkage clustering of an edge-weighted tree T is defined as the tree of clusters (a dendrogram) obtained by sequentially merging the edges of T in decreasing (increasing) order of similarity (dissimilarity) as follows:

(1) place each vertex of T in its own cluster
(2) sort the edges of T by weight
(3) for each edge (u, v) in sorted order, merge the clusters of u and v to form a new cluster.

The output of this clustering process is called the ***single-linkage dendrogram (SLD)***. The SLD is a binary tree of clusters, where the vertices of T are the leaf clusters, and the internal nodes correspond to merging two clusters by contracting an edge. The dendrogram enables users to easily process, visualize, and analyze the $n-1$ different clusterings induced by the single-linkage hierarchy.

Since hierarchical structure frequently occurs in real world data, single-linkage dendrograms have been widely used to analyze real-world data in fields ranging from computational biology [15, 26, 42], image analysis [17, 21, 33], and astronomy [4, 14], among many others [23, 27, 43]. Due to its real-world importance, and its importance as a sub-step in other fundamental clustering algorithms such as HDBSCAN* [10, 41], computing the SLD of an input weighted tree has been widely studied by parallel algorithms researchers in recent years, with novel algorithms and implementations being proposed for the shared-memory setting [21, 41], GPUs [31, 35], and distributed memory settings [17, 22].

In this paper, we are interested in *both* theoretically-efficient and practically-efficient parallel algorithms for computing the single-linkage dendrogram. Sequentially, a simple and practical SLD algorithm is to faithfully simulate the specification by essentially running Kruskal's minimum spanning tree algorithm using Union-Find. In this algorithm, which we call SeqUF, the m tree edges are first sorted by weight. Then, the algorithm runs in m sequential iterations, where the i-th iteration takes the i-th edge and merges the clusters corresponding to the endpoints of the edge. Maintaining information about the current cluster for a node in the tree is done using Union-Find. Overall, the work is $O(n \log n)$ due to sorting the edges, and the algorithm has $\Omega(n)$ depth (longest dependence chain) since the edges are merged sequentially.

Wang et al. [41] recently gave the first work-efficient parallel algorithm for the problem—their algorithm computes the SLD in $O(n \log n)$ expected work and $O(\log^2 n \log \log n)$ depth with high probability (*whp*). Although their algorithm is work-efficient with respect to SeqUF, it is challenging to implement as it relies on applying divide-and-conquer over the weights, which is implemented using the Euler Tour Technique [24]. Due to its complicated nature, this algorithm does not consistently outperform the simple SeqUF. Thus, the authors only released the code for SeqUF and suggested to always use SeqUF rather than the theoretically-efficient algorithm. The algorithm is also randomized due to the use of semisort [19, 41], and there is no obvious way to derandomize it to obtain a deterministic parallel algorithm for the problem.

As a fundamental problem on trees with significant real-world applicability, an important question then is whether there is a *relatively simple* parallel algorithm for this problem that has good theoretical guarantees, is more readily implementable, and can obtain non-trivial speedups over SeqUF. In this paper, we give a strong positive answer to this question by giving two new algorithms for SLD computation that achieve consistent and large speedups over SeqUF, both of which have good theoretical guarantees. Both of our algorithms are obtained through a better understanding of the structure of SLDs, and in particular, by showing how to build SLDs in a divide-and-conquer fashion using a *merge* primitive that can merge the SLDs of two trees under certain conditions (Sec. 3.1).

We leverage these structural results to design two novel deterministic parallel single-linkage dendrogram algorithms. The first algorithm is based on parallel tree contraction and stores spines (node-to-root paths in the dendrogram) in meldable heaps; it uses

heap meld and filter operations to implement dendrogram merging (Sec. 3.2) in the rake and compress steps. We describe a modification of this algorithm that eliminates the need for meldable heaps, which makes our algorithm simple to describe and implement (Sec. 4.2). The second algorithm, which we call ParUF is a bottom-up algorithm inspired by the nearest-neighbor chain algorithm for hierarchical agglomerative clustering that merges all local-minima (edges that are merged before all other neighboring edges by the sequential algorithm) in parallel (Sec. 4.1). We design an asynchronous version of ParUF, whose practical success shows that there is ample parallelism in many instances that the SeqUF algorithm does not exploit.

Our theoretical analysis of both algorithms reveals that the $\Theta(n \log n)$ solution obtained by existing SLD algorithms is in fact sub-optimal in many cases; both of our algorithms deterministically run in $O(n \log h)$ work where h is the height of the output dendrogram, where $\lfloor \log n \rfloor \leq h \leq n-1$. Thus, our algorithms can require asymptotically less work when the output SLD is not highly skewed. For instance, when $h = O(\log n)$ as in the case of a balanced dendrogram, our algorithms only use $O(n \log \log n)$ work. We complement our upper bounds with a simple comparison-based lower bound showing that our work bounds are optimal. Our algorithms are readily implementable and enable us to consistently compute the dendrogram of a billion-node tree in roughly 10 seconds on a 96-core machine, achieving up to 149x speedup over the highly-optimized SeqUF implementation.

The major contributions of this paper are:

- A novel merge-based framework for computing the single-linkage dendrogram (Sec. 3.1), and two instantiations of this framework, the first using parallel tree contraction with meldable heaps (Sec. 3.2), and the second algorithm (called RCTT) using the RC-tree tracing technique (Sec. 4.2), which is readily implementable.
- ParUF, a bottom-up algorithm that is a natural parallelization of the SeqUF algorithm, and is readily implementable using a fast asynchronous approach (Sec. 4.1).
- Analyses of our algorithms showing that the heap-based algorithm (Sec. 3.2) and ParUF (Sec. 4.1) deterministically achieve $O(n \log h)$ work, which is optimal for comparison-based algorithms. Our heap-based algorithm has poly-logarithmic depth.
- An experimental study of our ParUF and RCTT algorithms showing that they achieve between 2.1–150x speedup over SeqUF on a collection of billion-scale input trees (Sec. 5). Our implementation can be found at https://github.com/kishen19/ParSLD.

2 PRELIMINARIES

Notation. We denote a graph by $G(V, E)$, where V is the set of vertices and E is the set of edges in the graph. For weighted graphs, the edges store real-valued weights. We denote a weight or similarity of an edge $e = (x, y)$ either by writing $w(x, y)$ or $w(e)$ where $w : E \to \mathbb{R}$ is a weight function, or by placing weight $w \in \mathbb{R}$ in a tuple $(\{u, v\}, w)$ or (e, w). Given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, $G_1 \cup G_2$ denotes the graph $G(V_1 \cup V_2, E_1 \cup E_2)$. We also use the notation $G \cup \{e\}$, where $e = (u, v)$ is an edge, to denote the graph $G(V \cup \{u, v\}, E \cup \{e\})$. The number of vertices in a graph is $n = |V|$, and the number of edges is $m = |E|$. Cut(X, Y) denotes the set of edges between two sets of vertices X and Y.

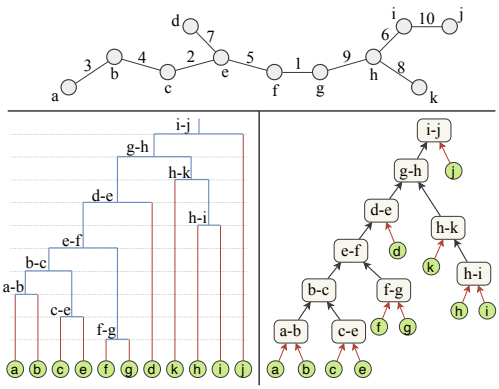

Figure 1: Example of single-linkage clustering on the input tree shown in the top panel. The bottom left panel shows a typical visualization of the dendrogram based on the "height" of each edge, and the bottom right panel shows the structure of the output SLD.

Model. We analyze the theoretical efficiency of our parallel algorithms in the binary-forking model [8], a concrete *work-depth model* used to analyze many recent modern parallel algorithms. The model is defined in terms of two complexity measures ***work*** and ***depth*** [8, 11, 24]. The ***work*** is the total number of operations executed by the algorithm. The ***depth*** is the longest chain of sequential dependencies. We assume that concurrent reads and writes are supported in $O(1)$ work/depth. A ***work-efficient*** parallel algorithm is one with work that asymptotically matches the best-known sequential time complexity for the problem.

2.1 Parallel Tree Contraction

Given a tree G, the parallel tree contraction framework by Miller and Reif [29] contracts G to a single node (or cluster) by repeatedly applying alternate rounds of `rake` and `compress`.

- **`rake`(u, v):** Given a vertex v of degree 1 and its (only) neighbor u, contract v and merge it into u.
- **`compress`(u, v, w):** Given a vertex v of degree 2 and its neighbors u and w, contract v and merge it into u (arbitrarily), and make u and w neighbors with $w(u, w) = w(v, w)$.

Parallel tree contraction has been studied since the 1980s [1–3, 20, 29, 34, 38] and can be solved deterministically in $O(n)$ work and $O(\log^2 n)$ span in the binary-forking model by simulating the PRAM algorithm of Gazit et al. [16]. The output of parallel tree contraction assigns each vertex to one of $O(\log n)$ rounds, specifying whether it is raked/compressed, and the edge(s) it is raked/compressed along. Another way of viewing the output is as a well-defined hierarchical decomposition (clustering) of trees. Starting with each vertex as a singleton cluster, each rake or compress merges two clusters along an edge; thus the subgraph induced on the vertices in a cluster will always be a connected subtree. Structurally, it can be represented as a rooted tree known as the rake-compress tree (or RC-tree). The RC-tree is essentially a hierarchical clustering defined over the input. In RC-trees, we have a node (which we call `rcnode`) corresponding to each vertex in the input tree. If a vertex v is contracted, say via the edge $e = (u, v)$, then the parent of `rcnode`(v) will be `rcnode`(u), and we also associate the edge e to `rcnode`(v).

2.2 Meldable Heaps

We make use of *meldable* heaps in this paper. The heaps contain edges keyed by a comparable edge weight where the comparison function orders edges in sorted order of rank. Concretely, we make use of binomial heaps [11], which support the *meld* operation in logarithmic time. The primitives we use are:

- $H' \leftarrow \textsc{Insert}(H, e)$: insert e into H; return the new heap H'.
- $(H', e) \leftarrow \textsc{DeleteMin}(H)$: remove the minimum element.
- $H' \leftarrow \textsc{Meld}(H_1, H_2)$: meld the two heaps H_1 and H_2.
- $(S, H') \leftarrow \textsc{Filter}(H, e)$: given a heap H and edge e, return a pair (S, H') where S contains all edges smaller than e in H, and H' contains all elements greater than e in H.

The filter_and_insert primitive used in Alg. 3 and Alg. 4 works by first performing Insert, followed by Filter.

Basic Operations. Sequential binomial heaps support performing Insert, DeleteMin, and Meld operations in $O(\log n)$ worst case time where n is the number of total elements in the heap(s) [11].

Filter. We implement Filter on a heap with s elements in $O(k \log s)$ work and $O(\log^2 s)$ depth where k is the number of elements extracted by the filter operation as follows. We independently filter the $O(\log s)$ roots of the binomial trees stored in the binomial heap in parallel. To filter a binomial tree, we check whether the root is filtered, marking it if so, and recursively proceed in parallel on all children of the root. This traversal costs $O(k)$ work and runs in $O(\log^2 s)$ depth; the number of nodes filtered in each tree can be computed in the same bounds by treating the set bits as an augmented value. Emitting the k removed elements into a single array can also be done in the same bounds by using prefix sums.

To rebuild the binomial trees and restore the invariant of a single binomial tree per-rank, we can first emit the children of all nodes removed in the previous step into a single array, where each subtree is stored along with its associated rank. Rebuilding the heap can then be done by simply performing the same procedure used to build a binomial heap on the trees in this collection. In a little more detail, the number of subtrees when we remove k nodes is at most $O(k \log s)$. We can group these subtrees by rank by sorting using a parallel counting sort, which can sort N elements in the range $[0, M]$ in $O(\log n + M)$ depth [7]. After sorting the trees into the $O(\log s)$ ranks, rebuilding the trees can be done within each rank in $O(\log s)$ depth using parallel reduce. We note that the overall structure we maintain is exactly the same as an ordinary binomial heap; we simply augment the heaps with a parallel filter operation that relies on a parallel rebuilding procedure.

2.3 Single-Linkage Clustering

Consider an input weighted undirected graph $G(V, E)$. In *single linkage* clustering, the similarity $\mathcal{W}(X, Y)$ between two clusters X and Y is the minimum similarity between two vertices in X and Y, i.e.,

$$\mathcal{W}(X, Y) = \min_{(x,y) \in \text{Cut}(X,Y)} w(x, y).$$

In this paper, we assume the input graph is an edge-weighted tree as it is well known that single-linkage clustering on weighted (connected[1]) graphs can be reduced to single-linkage clustering on weighted trees; the edges considered for merges are exactly that of the minimum spanning tree of the graph [18].

[1]If the graph is not connected, we can solve single-linkage clustering on each connected component independently.

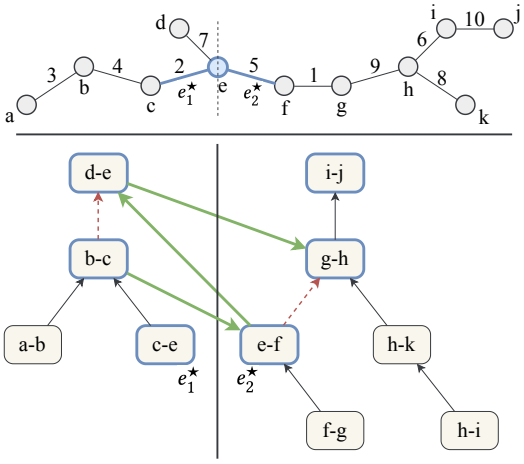

Figure 2: An example illustrating SLD-Merge. The tree is split at node e into two trees (the left and right sides of the dashed line) which share no edges, and only share the vertex e. The SLD-Merge routine merges the two spines formed by the lowest-rank edge incident to e in both trees.

Given an input edge-weighted tree $G(V, E)$, let the rank $r_e \in [n]$ of an edge e be the position of this edge in the edge sequence sorted by weight (ties broken consistently). We note that our algorithms do not require us to compute the ranks; however, this simplifies the presentation of our algorithms.

The single-linkage dendrogram (SLD) of a tree $G(V, E)$ is a rooted-tree $D(G)$ where each leaf corresponds to a vertex in V and each internal node in $D(G)$ corresponds to an edge in E. We denote the internal node corresponding to edge e as $node(e)$ or simply node e. See Figure 1 for an example.

The SLD problem has been widely-studied in recent years [10, 21, 31, 35, 41]. However, only Wang et al.'s algorithm [41] is work-efficient with respect to SeqUF. We discuss related work on SLDs in more detail in Appendix A.

We assume the output SLD will be stored as a linked list, where each node e points to its parent node $p(e)$. For convenience, we drop the leaf nodes and only consider the tree on the $n - 1$ internal edge nodes. For an edge $e \in E$, the $\text{spine}_{D(G)}(e)$ denotes the partial linked list starting from node e until the root in $D(G)$. We use $\text{spine}(e)$ when the tree and SLD are clear from context. We use the terms SLD and dendrogram interchangeably.

In this paper, we deal with three types of trees: the input tree, the SLD and RC-trees. To keep things clear, we use *vertices* and *edges* when referring to the input, *nodes* and *links* when referring to the SLD, and `rcnode` when referring to RC-trees.

3 MERGE-BASED ALGORITHMS

In this section, we discuss merge-based algorithms for computing SLDs. We first describe a merge subroutine called SLD-Merge that allows us to merge the dendrograms of two subtrees, and thus enables various divide-and-conquer algorithms. Leveraging the merge subroutine, we give an optimal algorithm for computing SLDs with the help of the parallel tree contraction framework.

3.1 Merging Dendrograms

The key component in our merge-based framework is the subroutine called SLD-Merge, which can merge the SLDs of two trees under the following conditions:

- the trees share exactly one vertex (denoted by v),
- the trees share no edges.

Given these conditions, observe that the union of the two trees will also be a tree. More abstractly, from the divide-and-conquer viewpoint merging is algorithmically useful if for example, an input tree is split at vertex v and the SLDs of the two resultant trees are computed recursively, and we wish to merge the two SLDs to compute the SLD of the entire tree; see Figure 2 for an example.

More formally, let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ denote the two input trees with $V_1 \cap V_2 = \{v\}$ and $E_1 \cap E_2 = \emptyset$, and let D_1 and D_2 denote their SLDs. We assume that the SLDs are maintained as linked lists, where each node points to its parent node. Alg. 1 defines the function SLD-Merge(G_1, G_2, v) that, given the two trees and their SLDs, returns the SLD of the merged tree.

Algorithm 1: SLD-Merge(G_1, G_2, v)

1 Let $e_1^\star$ and $e_2^\star$ denote the edges with minimum rank incident to vertex v in G_1 and G_2, respectively.
2 Merge D_1 and D_2 along the spines of $e_1^\star$ and $e_2^\star$.

Given the linked list representation, the spine of a node e is the partial linked list starting at e and ending at the root. The nodes have ranks in increasing order from e to the root. Thus, we can apply the standard list merge algorithm for merging the two (sorted) lists. The idea here is inspired by the Cartesian tree algorithm of [37]. Indeed, it is not hard to see that the Cartesian tree problem on lists is equivalent to single-linkage clustering on path graphs. In [37], the authors employ an elegant divide-and-conquer approach where they split the input list into two halves, compute the Cartesian tree of each half, and then merge them.

The key idea when merging the two Cartesian trees was that the merge impacts only nodes on certain spines, while the rest of the nodes are "protected". Interestingly, we prove that such a property holds even when we merge the dendrograms of trees with arbitrary arity. In particular, the merge potentially impacts only nodes on the spines of $e_1^\star$ and $e_2^\star$ (defined above). We will call these min-rank edges $e_1^\star$ and $e_2^\star$ as the ***characteristic edges*** of the merge, and their spines in their respective SLDs as the ***characteristic spines***. Therefore, in other words, the parent of nodes that are *not* on the corresponding characteristic spines remains unchanged. In the case where one of the trees is just a single vertex, we do not have a characteristic edge. Here, we call the empty list as the characteristic spine of this tree.

Before proving the correctness of SLD-Merge, we first state some useful structural properties of SLDs.

Definition 3.1 (Adjacent Superior and Inferior). Given a tree $G(V, E)$ and an edge $e = (u, v) \in E$, the edge $f \in E$ is an ***adjacent superior*** of e if $r_e < r_f$ and for every edge g in the unique path between e and f, $r_g < r_e$. The edge f is an ***adjacent inferior*** of e if $r_e > r_f$ and for every edge g in the unique path between e and f, $r_g < r_e$. Let $\mathcal{I}^u(e)$ (and $\mathcal{S}^u(e)$) denote the set of adjacent inferior (superior) edges to e that are closer to vertex u compared to vertex v. Similarly, we define $\mathcal{I}^v(e)$ and $\mathcal{S}^v(e)$. Let $\mathcal{I}(e) = \mathcal{I}^u(e) \cup \mathcal{I}^v(e)$ and $\mathcal{S}(e) = \mathcal{S}^u(e) \cup \mathcal{S}^v(e)$. See Figure 3 for an illustration.

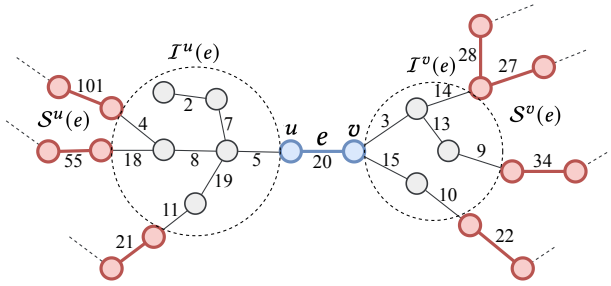

Figure 3: Adjacent Superiors and Inferiors (see Definition 3.1).

Observe that the subgraphs induced on the sets of edges $\mathcal{I}^u(e)$ and $\mathcal{I}^v(e)$ will be connected, respectively. We have the following lemma about the correspondence of these sets in the output SLD.

Lemma 3.2. *Let D be the output SLD of the tree $G(V, E)$. For the edge $e = (u, v) \in E$, let $D(e)$ denote the subtree rooted at node e, and let $D^u(e)$ denote the subtree rooted at the child of node e that contains the vertex u as a leaf. Similarly, we define $D^v(e)$. Then, $D^u(e) = \mathcal{I}^u(e)$ and $D^v(e) = \mathcal{I}^v(e)$.*

Proof. Observe that during single linkage clustering the edges in $\mathcal{I}(e)$ are processed before edge e. Just before e is processed, the clusters on the endpoints are exactly the sets $\mathcal{I}^u(e)$ and $\mathcal{I}^u(e)$. To see this, observe that after $\mathcal{I}(e)$ is processed, the minimum rank edge incident on clusters of u and v is the edge e, since all the other incident edges will be the set $\mathcal{S}(e)$. Hence, $D^u(e) = \mathcal{I}^u(e)$ and $D^v(e) = \mathcal{I}^v(e)$. □

We also have the following simple and useful observation.

Lemma 3.3. *Let $e_1, e_2, \ldots, e_d$ be the set of edges incident to some vertex u, sorted by rank. Then, $e_i \in \mathsf{spine}(e_1)$, for all $2 \le i \le d$.*

The proof follows due to the fact that these edges share an endpoint. Returning to the merge subroutine, we define a node (in D_1 and D_2) as ***protected under the merge*** if its parent doesn't change in the output after the merge. We now prove a crucial lemma. Henceforth, when we say "nearest edge", the distance here is the unweighted *hop* distance.

Lemma 3.4. *Every node in $D_1 \cup D_2$ that is not present in $\mathsf{spine}(e_1^\star) \cup \mathsf{spine}(e_2^\star)$ is protected under the merge.*

Proof. Let $e = (u, v) \in E_1$ such that $e \notin \mathsf{spine}(e_1^\star)$. Observe that if $\mathcal{I}(e)$ doesn't change when we union the two trees, then the subtree rooted at e in D_1 is protected. This follows from the fact that the subtree rooted at e in D_1 corresponds to a (connected) subtree in G_1 (by Lem. 3.2), and SLD is computed correctly on this subtree (by induction). Thus, it is enough to show that $\mathcal{I}(e)$ doesn't change for every $e \notin \mathsf{spine}(e_1^\star)$.

Let a be the lowest common ancestor of e and $e_1^\star$ in D_1. Then, observe that $r_e < r_a$ and a lies on the unique path between e and e_1', where e_1' is its nearest edge incident on v. We also know that $e_1' \in \mathsf{spine}(e_1^\star)$ by Lem. 3.3. Therefore, $\mathcal{S}(e)$ cannot change, since any possible change would be via e_1', i.e. via vertex v, but this is blocked by edge a. Hence, $\mathcal{I}(e)$ cannot change as well, completing the proof. A symmetric argument can be made for edges in E_2. □

Finally, we now prove the correctness of SLD-Merge.

Theorem 3.5. *SLD-Merge(G_1, G_2, v) outputs the correct SLD of $G_1 \cup G_2$.*

Proof. From Lem. 3.4, we know that nodes not on $\text{spine}(e_1^\star) \cup \text{spine}(e_2^\star)$ are protected. Consider some edge $e \in \text{spine}(e_1^\star)$, and let e_1' denote its nearest edge incident to v in G_1. Observe that e doesn't have an adjacent superior on the path from e to e_1'. Therefore, after the merge, e might have new adjacent superiors introduced along this path. The parent of e (say $p(e)$) will change iff $r_{p(e)} > r_f$ for some new adjacent superior f. We claim that all the new adjacent superiors for e belong to $\text{spine}(e_2^\star)$.

To see this, consider some new adjacent superior f, and let e_2' denote its nearest edge incident to v in G_2. Then, for all edges g in the unique path between e_2' and f, $r_g < r_e$ (by definition), which implies $r_g < r_f$. Thus, f too doesn't have an adjacent superior on the path from f to e_2'. From the proof of Lem. 3.4, f is not protected, implying that $f \in \text{spine}(e_2^\star)$.

Since $\text{spine}(e_2^\star)$ is sorted, if $p(e)$ changes, it will be the first node f in the list with rank greater than r_e. Thus, SLD-Merge is correct. □

3.2 Optimal Algorithm via Tree Contraction

We now describe an optimal $O(n \log h)$ work and $O(\log^2 n \log^2 h)$ depth algorithm for computing SLDs; we will refer to this algorithm as SLD-TreeContraction. We achieve this with the help of the merge subroutine described above and parallel tree contraction. Our bounds match those of the following comparison-based lower bound stated next, which we show in Appendix B:

Lemma 3.6. *For any $\lfloor \log n \rfloor \le h \le n - 1$, there is an input that every comparison-based SLD algorithm requires $\Omega(n \log h)$ work to compute the parent of every edge in the output dendrogram.*

As indicated previously, with the help of SLD-Merge, we can design divide-and-conquer algorithms that partition the input tree into smaller subtrees, compute their respective SLDs in recursive rounds, and finally applies the SLD-Merge subroutine, suitably, to obtain the overall SLD. A critical task here is to structure these recursive rounds as efficiently as possible, with low depth; parallel tree contraction [34] provides one such structure.

As discussed earlier (in Sec. 2), parallel tree contraction defines a convenient low-depth hierarchical decomposition (or clustering) of trees. Our core idea is to maintain the SLD of each cluster (which is a connected subtree) and apply SLD-Merge appropriately during rakes and compresses to obtain the SLD of the merged cluster. This way, by the end of tree contraction when we have a single cluster containing the entire input tree, we will have constructed its corresponding SLD, as required. First, we will discuss how rakes and compresses can be realized as a couple of SLD-Merge operations, assuming merges are performed as standard (linked) list merges. However, this approach leads to sub-optimal work and depth bounds. Second, we will discuss how to optimize the merges by additionally maintaining certain spines in a more efficient data structure, and prove optimal work and depth bounds.

3.2.1 A Sub-optimal Tree-Contraction Algorithm. Formally, for a cluster represented by u during tree contraction, let G_u denote the induced subtree on the vertices in cluster u, and let D_u denote the SLD of G_u. Consider some rake operation given by $\texttt{rake}(u, v)$, raking vertex v into u. The subtrees G_u and G_v are connected via the edge $e = (u, v)$. For convenience, let G_{uv} denote the union of G_u and G_v, i.e. the cluster obtained after performing the rake. We can implement the rake using two steps: (1) Add the edge $e = (u, v)$ to G_v to obtain the subtree G_v', and (2) merge the subtrees G_u and G_v'. We compute the SLD of G_{uv} in the following two step process:

Algorithm 2: $\texttt{rake}(u, v)$

1 $G_v' \leftarrow G_v \cup \{e\}$, and
  $D_v' \leftarrow$ SLD-Merge$(G_v, \{e\}, v)$,
2 $G_{uv} \leftarrow G_u \cup G_v'$, and
  $D_{uv} \leftarrow$ SLD-Merge(G_u, G_v', u).

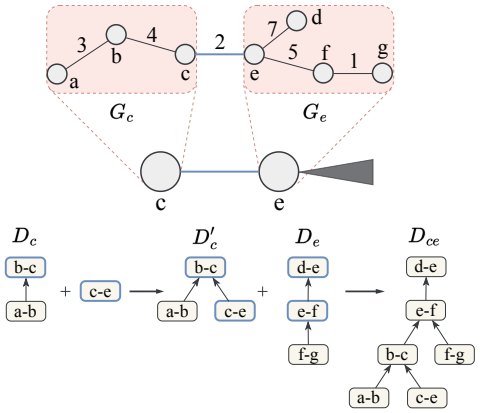

Figure 4: An example illustrating the two-step rake (see Alg. 2). Here, we perform $\texttt{rake}(e, c)$, which rakes c into e.

Compress can be performed in an identical fashion: given an operation $\texttt{compress}(u, v, w)$, we can choose to merge v with u (arbitrarily) and a similar viewpoint, as in rake, can be applied for this merge. Therefore, each rake and compress operation (identically) performs two SLD-Merge operations.

During tree contraction, if we execute the rake/compress operations in parallel, we could encounter race conditions. For instance, multiple clusters might get raked or compressed into the same cluster, and we make no assumptions about the structure of the input trees (e.g., some applications of tree contraction assume bounded-degree input trees). Note that these are the only race conditions that occur in any rake/compress round of tree contraction.

We can handle parallel rake or compress operations as follows: let $v_1, v_2, \ldots, v_d$ be vertices that are being raked (or compressed) into the same cluster u. Observe that the step (1) described above doesn't affect G_u or D_u. Thus, this step can be run safely in parallel. Let G_u'' denote the tree formed by taking the union of trees $G_{v_1}', G_{v_2}', \ldots, G_{v_d}'$. Our idea is to first compute the dendrogram of G_u'', and then finally run SLD-Merge(G_u, G_u'', u). We can compute the dendrogram of G_u'' by simply merging the dendrograms of $G_{v_1}', G_{v_2}', \ldots, G_{v_d}'$ together, i.e. merging the d sorted lists given be $\text{spine}((u, v_i))$ for each $i \in [d]$. This is correct due to a simple extension of Lem. 3.3: since all these edges incident to u will be on the

same spine, their respective spines in G'_{v_i} will also end up on the same spine. This can be computed quickly by running a parallel reduce operation with SLD-Merge as the reduce function, resulting in depth $O(\log d)$ (= $O(\log h)$ by Lem. 3.3) times the depth of SLD-Merge for all the rakes (and compresses) on u.

The naive (sequential) linked list based implementation of SLD-Merge has $O(h)$ work and depth. Thus, if we charge each merge cost of $O(h)$ to the vertex that is being raked/compressed, we get an overall work bound of $O(nh)$ for this sub-optimal version of SLD-TreeContraction. As discussed, each rake/compress round will have a worst case depth of $O(h \log h)$. Since the number of such rounds is $O(\log n)$, the overall depth will be $O(h \log h \log n)$. We will now prove its correctness.

Lemma 3.7. *The sub-optimal version of SLD-TreeContraction correctly computes the SLD.*

Proof. Observe that both step 1 and step 2 are merges between subtrees that satisfy the requirement mentioned in Sec. 3.1, i.e. the subtrees share exactly one vertex and no edges. Thus, rake correctly computes the dendrogram of the merged cluster; a similar argument works for compress. Since tree contraction is just a sequence of rake and compress operations, the correctness follows by a simple inductive argument. □

3.2.2 Optimizing the merge step. We now describe how to optimize the merge step. From Sec. 3.1, we know that merging affects only nodes on the characteristic spines associated to that merge. Our main idea is that, in addition to the linked list representation for storing the output of the dendrogram, for each cluster we (try to) maintain the characteristic spines, corresponding to the next (future) merge involving that cluster, in *parallel binomial heaps* and perform merges via these heaps. We chose binomial heaps since (to the best of our knowledge) it is the only data structure that supports fast merge (or meld) and can support low-depth parallel filter operations. We describe the heap interface and the cost bounds in Sec. 2.2. As we will see, if we perform only rakes, it is easy to always store these characteristic spines. However, compress operations pose a significant challenge towards exactly storing the characteristic spines. Nevertheless, if compresses are performed carefully (in terms of which cluster to merge with), we show that the spine consequently stored at each cluster is always sufficient for every merge operation performed in the future. Henceforth, when we mention heaps, we refer to parallel binomial heaps.

Extending the notation from before, let H_u denote the min-heap associated with the cluster represented by u. We first extend the notion of protected nodes defined in Sec. 3.1 as follows: a node e is ***protected*** if its parent node is identical to its parent in the final output. We would like to maintain the following invariant: (1) nodes present in the heap are potentially *not* protected and correspond to some spine in the dendrogram associated to that cluster, and, (2) all nodes in that cluster *not* present in the heap are protected. We also show that when a node is deleted from its heap during the course of the algorithm, it is definitely protected. Thus, we ensure that we update the output (i.e. parent array) only when a node is deleted from its heap. This invariant helps us carefully charge the associated merge costs to these nodes.

We will now describe optimized versions of the previously described two-step rake and compress operations, in which SLD-Merge will be implemented in a white-box manner via the heaps.

`rake`(u, v). Rake is implemented as follows:

Algorithm 3: `rake`(u, v) (optimized version)

```
1  Let e = (u, v).
2  S, H'_v = H_v.FILTER_AND_INSERT(e).
3  H_u ← MELD(H_u, H'_v).
4  // Update output; S contains edges f ∈ H_v that had r_f < r_e
     and were filtered out on Line 2.
5  if S.size > 0 then
6  |   Sort S according to rank.
7  |   parfor each i = 1 to S.size − 1 do
8  |   |   p(S[i]) = S[i + 1]
9  |   end
10 |   p(S[S.size]) = e.
11 end
```

Note that vertex u will be the representative of the merged cluster, hence the new spine is stored in H_u. Now, we prove the following claim about the set S.

Claim 3.8. *The nodes in the set S computed during rake are protected after the merge.*

Proof. To see this, observe that for each $f \in S$, $r_f < r_e$. Thus, e is either an adjacent superior to f, or f has an adjacent superior on the unique path between f and e. Since cluster v is being raked, for any future merge corresponding to some edge g, the unique path between f and g will contain e. Thus, e protects f from all future merges. Further, since the nodes in S are on the same spine, they will be present in sorted order, and e will be the parent of the max-rank edge in S. □

`compress`(u, v, w). Compress is executed in a similar manner as rake, except for one difference: the cluster v will always merge with the cluster along the *lesser rank edge* (previously we could merge with an arbitrary neighboring cluster). The pseudocode is shown in Alg. 4. Next, we prove a similar claim, as in rake, about the set S.

Claim 3.9. *The nodes in the set S computed during compress are protected after the merge.*

Proof. Consider some edge $f \in S$. We have $r_f < r_{e_1} < r_{e_2}$. Let e'_1 and e'_2 denote the adjacent superiors on the paths from f to e_1 and e_2, respectively. Consider some future merge involving an edge g.

(1) If $e'_1 \neq e_1$ and $e'_2 \neq e_2$, then e'_1 and e'_2 protect f from any future merges. (Interestingly, in this case, f would have already been protected by a previous rake/compress operation involving either e'_1 or e'_2, but this is not important for this proof.)

(2) If $e'_2 \neq e_2$, then f is protected by e'_2 in the case when g is closer to e_2 than e_1. Similarly, f is protected by e_1 in the case g is closer to e_1 than e_2.

(3) Finally, we have the case $e'_2 = e_2$. If g is closer to e_1, e'_1 protects f. Consider the case when g is closer to e_2 than e_1. Observe that g cannot have an endpoint in the cluster containing f until $g = e_2$. After merging along e_2, any future merge corresponding to edge g

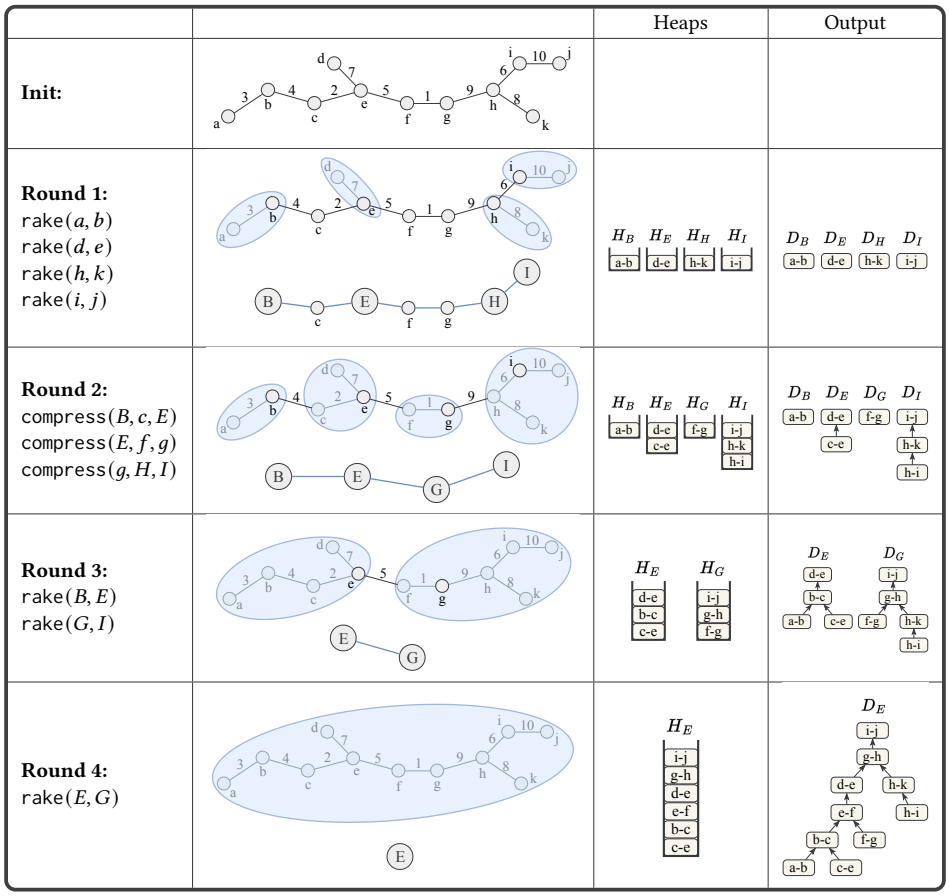

Figure 5: A full example of SLD-TreeContraction: the first column represents the rakes/compresses performed in that round; the second column displays the clustering obtained by tree contraction (as well as a compact representation); the third column displays the (non-empty) heaps maintained at each cluster; and the fourth column represents the (non-empty) SLDs of each cluster.

Algorithm 4: `compress`(u, v, w) (optimized version)

```
1  Let e_1 = (u, v) and e_2 = (v, w).
2  if r_{e_1} > r_{e_2} then
3  |  swap u and w // Thus, we will have r_{e_1} < r_{e_2}.
4  end
5  S, H'_v = H_v.FILTER_AND_INSERT(e_1).
6  H_u ← MELD(H_u, H'_v).
7  // Update output; S contains edges f ∈ H_v that had r_f < r_{e_1}
     and were filtered out on Line 5.
8  if S.size > 0 then
9  |  Sort S according to ranks.
10 |  parfor each i = 1 to S.size − 1 do
11 |  |  p(S[i]) = S[i + 1]
12 |  end
13 |  p(S[S.size]) = e_1.
14 end
```

that is closer to e_2 will not affect f since it will be protected by e_2. We claim that the merge corresponding to e_2 also doesn't affect f.

Firstly, if $r_{e_1'} < r_{e_2}$, then we are done since both e_1' and e_2 are adjacent superiors to f but e_1' will be processed before e_2 in single-linkage clustering. Hence, e_1' will be the parent of f in the output SLD. Now, assume $r_{e_1'} > r_{e_2}$. We will prove that this is not possible. We know that e_1' and f are present on the path between e_1 and e_2. Consider the merge corresponding to the edge e_1'. Observe that this has to be a compress (none of its endpoints can have degree 1). Since it is a compress, the counterpart edge, say e_1'', will have rank greater than e_1'. We know that e_1'' has to be present on the path between e_1 and e_2. It cannot be present between f and e_1' (contradiction that e_1' is an adjacent superior). Further, it cannot be present between f and e_2 as well (contradiction that e_2 is an adjacent superior). Thus, it must be present between e_1 and e_1'. However, the present compress corresponding to e_1 cannot be performed until the merge corresponding to e_1'' is performed. If we repeat the same argument for e_1'', we will reach the conclusion that either $r_{e_1'} < r_{e_2}$, or $r_{e_1'} < r_{e_1}$, both leading to contradictions. □

The following correspondence is true of Alg. 3, barring for the update output step: the filter and insert step corresponds to step 1, whereas the meld step corresponds to step 2, respectively in Alg. 2, and similarly for the optimized version of compress. The main difference is that the optimized versions of rake and compress essentially

delay the update to the output until the nodes get protected, thus having to update the parent of any node at most once.

This correspondence helps us handle the race conditions mentioned before, i.e. multiple clusters getting raked/compressed into the same cluster in the same round. Using the same notation as before, we perform the filter and insert step at the clusters being raked/compressed, in parallel, to obtain the heaps $H'_{v_1}, H'_{v_2}, \ldots, H'_{v_d}$. Then, we merge all of these heaps together to obtain the heap H''_u by running a parallel reduce operation (with *meld* now as the function). Finally, we meld H_u and H''_u to obtain the final spine.

For performing the merges correctly, we need to ensure that we indeed merge the characteristic spines associated to that merge, as required by SLD-Merge. Instead, we show that for every merge performed, at least one of the spines will be the exact characteristic spine, and the other spine will be sufficient for the corresponding merge, as stated by the following lemma.

Lemma 3.10. *For any cluster u during tree contraction, H_u stores a spine in the SLD of G_u satisfying one of the following properties:*

(1) H_u contains the characteristic spine corresponding to the next merge involving u.

(2) Let e be the characteristic edge in u, and f be the characteristic edge in the other subtree corresponding to the next merge involving cluster u. Then, H_u contains $\mathsf{spine}(e')$ such that $\mathsf{spine}(e') \subseteq \mathsf{spine}(e)$ and $r_{e'} < r_f$.

Proof. We prove the lemma by induction on the tree contraction rounds. Initially, all the clusters are singleton and the heaps are empty, which corresponds to the characteristic spine of the next merge. Let us assume that the clusters at the end of round $k-1$ store spines in their heaps that satisfy one of the properties stated. Now, consider some cluster u. At round k, if no clusters merge with u, we are done.

Suppose that in round k, the clusters $v_1, v_2, \ldots, v_d$ get *raked* into u. We first look at the filter and insert step: the single edge $e_i = (u, v_i)$ corresponds to the characteristic spine for its subtree. By induction, in the first case H_{v_i} contains the characteristic spine, in which case the merge is correct and we obtain the $\mathsf{spine}(e_i)$ in H'_{v_i}. Otherwise, let $\mathsf{spine}(f_i)$ be the required characteristic spine. Since we are in the second case, we know that H_{v_i} contains $\mathsf{spine}(f'_i) \subseteq \mathsf{spine}(f_i)$ such that $r_{f'_i} < r_{e_i}$. Observe that the output of merging $\mathsf{spine}(e_i)$ and $\mathsf{spine}(f_i)$ is equivalent to the output of merging $\mathsf{spine}(e_i)$ and $\mathsf{spine}(f'_i)$, since $\mathsf{spine}(f'_i) \subseteq \mathsf{spine}(f_i)$. Thus, the merge is correct, and we similarly obtain $\mathsf{spine}(e_i)$ in H'_{v_i}. By Lem. 3.3, the characteristic spine of the SLD of G''_u when merging with the SLD of G_u is nothing than the union of $\mathsf{spine}(e_i)$ over all $i = 1, \ldots, d$. Thus, H''_u stores the characteristic spine of the next merge. By induction, if H_u stores the characteristic spine, the merge is performed correctly. Otherwise too, with a similar (equivalence) argument as before, the merge is correct.

Now, let us instead consider that round k performs compresses. Let $e_{i_1} = (u, v_i)$ and $e_{i_2} = (v_i, w_i)$, such that $r_{e_{i_1}} < r_{e_{i_2}}$. The same argument, as in the case of rakes, can be extended to work here. Now, we are left to prove that the final computed spine in H_u satisfies one of the stated properties.

In the future, u can be involved in a merge along the edge e_{i_2}. However, the final spine computed in H_u contains $\mathsf{spine}(e_{i_1})$, which is a subset of the characteristic spine corresponding to the merge along e_{i_2}; the characteristic edge would be the min-rank edge incident at that vertex, but by Lem. 3.3, $\mathsf{spine}(e_{i_1})$ will be a subset of the characteristic spine. Thus, if e_{i_2} is the next merge, H_u satisfies property (2).

Since we do not delete nodes from H_u until cluster u is raked/-compressed, it satisfies property (2) until then. By induction, H_u will continue to satisfy property (2) for all other possible future merges determined by compresses (involving u) performed in the past. Hence, it will continue to satisfy this property even after round k (this holds even in the case when the round performs rakes).

Thus, by induction, the heaps stored at any cluster always satisfies one of the two stated properties. □

At the end of tree contraction when we obtain a single cluster, observe that we have a spine stored in the heap associated to that cluster, whose parent nodes are not updated in the output. Since there are no more merges left and the nodes form a spine, we can just sort all the nodes in the heaps and assign parents, identically to the update output step described in Algs. 3 and 4. With the algorithm fully described, we will now prove the correctness and work-depth bounds of SLD-TreeContraction.

Theorem 3.11. *SLD-TreeContraction correctly computes the SLD, and runs in $O(n \log h)$ work and $O(\log^2 n \log^2 h)$ depth.*

Proof. The proof idea is similar to Lem. 3.7, except the fact that SLD-Merge is executed in a white-box manner. By Lem. 3.10, every cluster maintains the correct characteristic spines (or a sufficient version of it) in their heaps, and thus rakes and compresses are executed correctly. By Claim 3.8 and Claim 3.9, the output will be updated correctly. Thus, as tree contraction is a sequence of rakes and compresses, we can apply a simple inductive argument as before to complete the correctness argument.

We will now analyze the work and depth bounds. Recall that we decided to use parallel binomial heaps since they support both fast meld ($O(\log(s)$ work and depth), where s is the size of the merged heap) and low-depth parallel filter operations ($O(k \log s)$ work and $O(\log^2 s)$ depth, where k is the number of nodes filtered). Note that since at any point the nodes in any heap correspond to some spine, the size of every heap will always be $O(h)$.

Work Analysis. If k nodes are protected at any rake/compress, the total work for the filter step will be $O(k \log h)$. The subsequent output SLD update step also performs $O(k \log h)$ work (sorting plus update). Thus, we can charge $O(\log h)$ work to each of the protected nodes. Since each node is protected at most once, the overall work incurred will be $O(n \log h)$. Next, each meld operation requires $O(\log h)$ work. Since every rake/compress operation is associated to exactly one edge of the input tree, we can associate this $O(\log h)$ work to that edge, leading to a total of $O(n \log h)$ work across all edges. Hence, the overall work of the algorithm is $O(n \log h)$.

Depth Analysis. Let us now analyze the depth of the algorithm. The number of rake and compress rounds is at most $O(\log n)$. The cost of spawning threads within a round is $O(\log n)$. The depth of the heap filter step is $O(\log^2 h)$, the depth of update output and heap meld is $O(\log h)$ (times $O(\log h)$ for meld due to the *reduce*). Therefore, the overall depth of each rake/compress round is $O(\log n \log^2 h)$, giving the overall depth bound of $O(\log^2 n \log^2 h)$. □

4 PRACTICAL ALGORITHMS

In this section, we describe two algorithms for dendrogram computation, both of which have strong provable guarantees (both achieve the optimal work bounds we showed in Sec. 3.1) but are also implementable and achieve good practical performance. The first is an *activation-based* algorithm that achieves the optimal work bound of $O(n \log h)$ and has $O(h \log n)$ depth (Sec. 4.1). The second algorithm is a twist on the tree contraction algorithm described in Sec. 3.2 that first performs tree contraction, and subsequently *traces* the tree contraction structure to identify the parent of each edge in the dendrogram (Sec. 4.2). It also achieves optimal work, and additionally runs in worst-case poly-logarithmic depth.

4.1 Activation-Based Algorithm (ParUF)

The sequential Kruskal algorithm processes edges in increasing order of rank, thus emulating the process of single-linkage HAC and building the output dendrogram in a bottom-up fashion, one node at a time. We can generalize this approach to safely process multiple edges at the same time by building the dendrogram in a bottom-up fashion, one "level" at a time. This approach is similar to the nearest-neighbor chain algorithm, a well-known technique for HAC [5] that obtains good parallelism in practice for other linkage criteria such as average-linkage, and complete-linkage [13, 40, 44].

Indeed, the algorithm we propose can be viewed as an optimized and parallelized version of the sequential algorithm in [13]. The striking difference between our activation-based algorithm and other nearest-neighbor chain algorithms is that our algorithm is *asynchronous* and only requires a single instance of spawning parallelism over the set of edges, whereas all other nearest-neighbor chain implementations we are aware of run in synchronized rounds.

The following simple but important observation allows us to process multiple edges at a time:

Lemma 4.1 (folklore). *If an edge $e = (u, v)$ is a local minima, i.e. $r_e < r_f$ for all other edges f incident to the clusters containing u and v, then the clusters u and v can be safely merged.*

The proof follows due to the fact that single-linkage clustering processes edges in sorted order of ranks, and hence edge e will be processed before all of the other edges incident to its endpoints. In other words, the edge e is processed only when it becomes the minimum rank edge incident to the clusters on both of its endpoints. We also have the following useful lemma.

Lemma 4.2. *Let $e = (u, v)$ be a local minima. Then, the parent of node e in the output SLD will correspond to the minimum rank edge incident on the merged cluster uv.*

Proof. Once u and v are merged along e, let $e_1, e_2, \ldots, e_d$ denote the set of edges incident to the merged cluster uv, in sorted order of ranks. The parent of e is the first edge that merges the cluster uv with some other cluster. However, uv can merge only via one of $e_1, e_2, \ldots,$ or e_d, and the first edge to be processed among them will be e_1, by the definition of single-linkage clustering. Therefore, e_1 will be the parent of e in the output SLD. □

Based on these observations, we now describe an asynchronous activation-based algorithm that we call ParUF. We give the pseudocode in Alg. 5. The idea is natural: when an edge becomes a local minima, we merge it. We maintain the set of edges incident to a cluster in a meldable min-heap (which we call the ***neighbor-heap*** of that cluster). Each unmerged edge will be present in two neighbor-heaps corresponding to the clusters containing it's endpoints. The element at the top of a cluster's heap will correspond to the min-rank edge incident to that cluster. We maintain the cluster information using a Union-Find data structure. Note that due to our strategy of only processing the local minima in parallel, we can use any sequential Union-Find structure with path compression. To identify if an edge is ready to be processed, i.e. it is a local minima, for each edge $e \in E$ we maintain an integer $status(e)$ value:

$$status(e) = \begin{cases} 2, & \textit{ready} \\ 1, & \textit{almost ready} \\ 0, & \textit{not ready} \\ -1, & \textit{inactive} \end{cases}$$

An edge is *ready* if it is at the top of both the neighbor-heaps of its endpoints, i.e. it is a local minima. An edge is *almost ready* if it is at the top of the neighbor-heap of only one of it's endpoints. If it is not on top of either neighbor-heaps, the edge is *not ready*. Finally, if the edge has already been merged/processed, it is *inactive*.

The overall algorithm is given in Alg. 5. The updates/accesses to $status(e)$ (Line 7 and Line 19) must be atomic since it could be updated/accessed by both of it's endpoints simultaneously. We now show that the rest of the steps of the algorithm do not have any race conditions, and that the algorithm is efficient:

Algorithm 5: Activation-based Algorithm (ParUF)

```
1  F ← Initialize Union-Find with all singleton clusters
2  Initialize the output SLD: ∀e ∈ E, p(e) = e
3  heaps ← Initialize neighbor heaps
4  Initialize status(.) values
5  parfor each e ∈ E do
6      cur ← e
7      while CAS(status(cur), 2, −1) do
8          (u, v) ← cur
9          (u′, v′) ← (F.Find(u), F.Find(v))
10         w ← F.Union(u′, v′)
11         heaps(u′).delete_min()
12         heaps(v′).delete_min()
13         heaps(w) ← Meld(heaps(u′), heaps(v′))
14         if heaps(w) is empty then
15             break
16         end
17         new_cur ← heaps(w).Top()
18         p(cur) = new_cur
19         atomic_inc(status(new_cur))
20         cur ← new_cur
21     end
22 end
```

Theorem 4.3. ParUF *correctly computes the SLD, and runs in $O(n \log h)$ work and $O(h \log n)$ depth.*

Further Optimizations and Implementation. As we will see in Sec. 5, the height of the resultant SLD is typically large in many instances. However, in most cases, the number of nodes in each level of the output dendrogram, as we go upwards, converges to 1 quickly. In other words, the number of local-minima edges is exactly one the majority of the time, rendering ParUF ineffective. However, we can apply a very simple optimization in this case.

If the number of local-minima edges is 1, this means we have to process the edges one-by-one. However, we know that they will be processed in sorted order. Thus, when running ParUF, if the number of local-minima edges (or the number of *ready* edges) drops to 1, we can stop and compute the set of remaining edges, say E'. Then, we sort E' based on rank, and assign $parent[E'[i]] = E'[i+1]$. In terms of finding out the number of ready edges, a simple approach is to periodically stop and check the count. This optimization provides incredible speed-ups in most of our experiments. However, it is not too hard to generate adversarial inputs that have low parallelism, but elude this strategy (for instance, if the output dendrogram has two nodes in each level for the majority of the time.)

4.2 RC-Tree Tracing Algorithm (RCTT)

Implementing a fast and practical algorithm that computes the SLD by leveraging the properties of SLD-Merge is highly non-trivial. The practical bottleneck of a faithful implementation of SLD-TreeContraction, our merge-based algorithm appears to be the need to maintain meldable heaps supporting the *heap-filter* operation for merging spines. In this section, we explore a few more structural properties of SLD-Merge and parallel tree contraction to design a fast and practical $O(n \log n)$ work and $O(\log^2 n)$ depth algorithm for computing the SLD that completely removes the requirement of maintaining the spines. The idea is to use the RC-tree representation of the tree contraction process and apply a post-processing *tracing* step to compute the final output.

Alg. 6 gives pseudocode for the RCTT algorithm. It first computes the RC-tree associated to the tree contraction performed by SLD-TreeContraction, without computing the output SLD or maintaining any spines (Line 1). We note that when a vertex (cluster) is compressed in the RC-tree, it will merge with the neighbor along the lesser rank edge, as required by the algorithm SLD-TreeContraction.

Given the RC-tree, consider some edge e. Recall that each edge e is associated to the `rcnode` of some vertex v that gets raked or compressed via e. From the viewpoint of SLD-TreeContraction, `rcnode`(v) corresponds to the stage when e gets introduced into some heap. Edge e will successively be involved in every rake/compress operation involving the cluster containing v until (if at all) it gets protected (or filtered during a heap-filter operation) by some edge f. More specifically, e is either protected by the first edge f it encounters during tree contraction, after it's introduction, such that $r_e < r_f$, or it doesn't encounter such an edge and is, consequently, present in the heap at the root `rcnode`, or in other words, it is protected at the root `rcnode`. Let u denote the `rcnode` where edge e gets protected. A critical observation here is that the set of rake/compress operations that includes e until it becomes protected are associated to the `rcnode`(s) along the (unique) path between

Algorithm 6: RCTreeTracing

```
1  Compute the RC-Tree RCT.
2  bkts ← set of empty buckets corresponding to each u ∈ V
   parfor each e ∈ E do
3      u ← rcnode associated to e
4      u ← u.parent
5      f ← edge associated to u
6      while r_f < r_e and u is not the root do
7          u ← u.parent
8          f ← edge associated to u
9      end
10     Add e to bucket corresponding to u
11 end
12 parfor each u ∈ V do
13     Let bkt be the bucket associated to rcnode(u)
14     Sort the edges in bkt by ranks
15     Let e ← edge associated to rcnode(u)
16     parfor each i = 1 to bkt.size − 1 do
17         p[bkt[i]] = bkt[i + 1]
18     end
19     if u is not root then
20         p[bkt[bkt.size]] ← e
21     end
22 end
```

`rcnode`(u) and `rcnode`(v). This is true due to the properties of RC-trees: the set of clusters containing the edge e throughout tree contraction correspond exactly to the set of `rcnodes` on the path from `rcnode`(v) until the root (in order). But, in SLD-TreeContraction, e gets filtered out from its heap once it encounters f (or `rcnode`(u)).

Thus, for each edge e, starting from `rcnode`(v), we traverse the $O(\log n)$ length path in the RC-tree along the path towards the root until we find `rcnode`(u). This way, for each `rcnode`, we can collect all nodes e that were protected at this node. This corresponds exactly to the set S obtained by the first step via the *heap-filter* operation; in case of the root, it is the remaining set of nodes in the spine. We can finally post-process these sets by sorting each of them by rank and updating the output SLD same as before. The pseudocode of this algorithm, which we refer to as RCTreeTracing (RCTT in short), is given in Alg. 6.

Analysis and Implementation. RCTreeTracing is a simpler algorithm than SLD-TreeContraction in the sense that it doesn't require meldable or filterable heaps and, in fact, doesn't require us to perform the actual merges of edges. The main practical challenge in the algorithm is to maintain a dynamic adjacency list as the input tree contracts due to rakes and compresses. The RC-tree can easily be computed in $O(n)$ work and $O(\log^2 n)$ depth. Sorting within buckets to compute the final output runs in $O(n \log h)$ work and $O(\log^2 n)$ depth, since the bucket sizes are $O(h)$ (all of these nodes are along some spine). The tree tracing step has the most work, i.e. $O(n \log n)$ work and $O(\log^2 n)$ depth, since it requires us to trace the entire height of the RC tree from each node in the worst case, and the height of the RC Tree is $O(\log n)$. However, in terms of

experiments, we see that the tracing step is very fast; indeed the bottleneck is the RC tree construction time (see Fig. 7).

5 EXPERIMENTAL EVALUATION

In this section we evaluate our parallel implementations for SLD construction and show the following main experimental results:

- RCTT is usually fastest on our inputs, achieving 2.1–132x speedup (16.9x geometric mean) over SeqUF on billion-scale inputs.
- ParUF achieves 2.1–150x speedup over SeqUF (5.92x geometric mean) on billion-scale inputs, but can be up to 151x slower than SeqUF on adversarial inputs.

Experimental Setup. Our experiments are performed on a 96-core Dell PowerEdge R940 (with two-way hyperthreading) with 4×2.4GHz Intel 24-core 8160 Xeon processors (with 33MB L3 cache) and 1.5TB of main memory. Our programs use the work-stealing scheduler provided by ParlayLib [6]. Our programs are compiled with the g++ compiler (version 11.4) with the -O3 flag.

Inputs. The ***path*** input is a path containing n vertices and $n-1$ edges arranged in a path (or chain); ***star*** is a star on n vertices where one vertex, the star center, has degree $n-1$, and all other vertices are connected to the center, and have degree 1; ***knuth*** is a tree similar to the dependency structure of the Fischer-Yates-Knuth shuffle [9], as follows: vertex $i > 0$ picks a neighbor in $[0, i-1]$ uniformly at random and connects itself to it.

We also generate several real-world tree inputs drawn real-world graphs. ***Friendster*** is an undirected graph describing friendships from a gaming network.[2] ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [25].[3] We build tree inputs for these real-world graphs by (1) symmetrizing them if needed (2) setting the weight of each edge (u, v) to be $\frac{1}{1+t(u,v)}$, where $t(u, v)$ is the number of triangles incident on the edge (u, v) and (3) computing a minimum spanning tree.

We build another real-world tree input using the ***BigANN*** dataset of SIFT image similarity descriptors; we compute the minimum spanning tree over an approximate k-nearest neighbor graph over a 100 million point subset of the BigANN dataset.[4] For our construction, we used an in-memory version of DiskANN algorithm [39] implemented in the ParlayANN library [28].

Weight Schemes. We consider several different weight-schemes. The unit ***unit*** assigns all edges a weight of 1. The ***perm*** scheme generates a random permutation of the edges and assigns each edge a weight equal to its index in the random permutation. The ***low-par*** scheme is only applicable to paths, and is designed to be adversarial for the ParUF algorithm. This scheme assigns weights in increasing order to the first half of the edges in the path, and assigns weights in decreasing order for the second half of the path.

5.1 Algorithm Performance

Next, we analyze the performance of our algorithms, including (1) their self-speedup, (2) their speedup over SeqUF, and (3) the performance breakdown of our algorithms. Fig. 6 shows the running times of our algorithms on a representative subset of the 100M-scale inputs as a function of the number of threads.

[2]Source: https://snap.stanford.edu/data/.
[3]Source: http://law.di.unimi.it/webdata/twitter-2010/.
[4]Source: http://corpus-texmex.irisa.fr/.

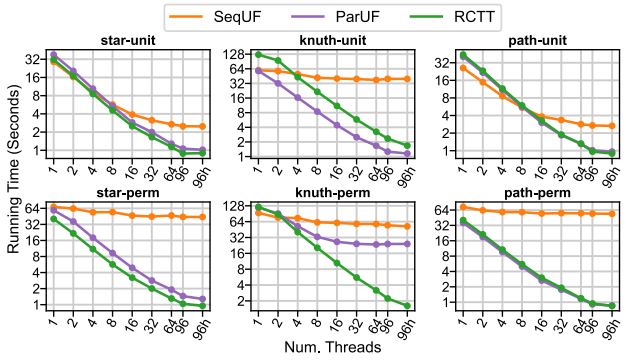

Figure 6: Running time of our SLD implementations on different input trees as a function of the number of threads. All inputs contain 100M vertices.

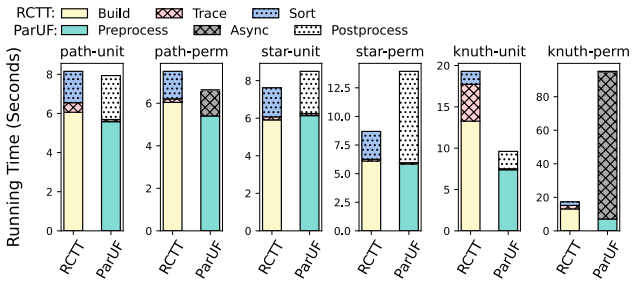

Figure 7: Parallel 192 thread running time breakdowns of the RCTT and ParUF algorithms on billion-scale inputs. For RCTT, the Build step corresponds to building an RC tree; the Trace step finds the bucket associated with each edge; and the Sort step sorts each of the buckets. For ParUF, the Preprocess step sorts the edges and identifies initial local minima; the Async step performs bottom-up clustering; the Postprocess step sorts all remaining edges once the number of local-minima in the Async step becomes 1.

SeqUF. SeqUF achieves between 1.36–11.6x self-relative speedup (2.94x geometric mean); it achieves the best self-speedups for the star and path graphs using unit weights as shown in Fig. 6. We emphasize that despite its name, SeqUF is able to leverage parallelism since the first step in the algorithm is to sort the edges, and we use a highly optimized parallel sort from ParlayLib [6]. For all other inputs, in which the input tree or edge weights induce an irregular access pattern, the algorithm achieves poor speedup (under 2x). In more detail, for star and path graphs using unit weights, the edges merged all lie one after the other in memory, and so the access pattern of the sequential algorithm has good locality. When the weights are permuted or the input tree has an irregular structure (e.g., in the case of the knuth inputs), the algorithm accesses essentially two random cache-lines in every iteration.

ParUF. ParUF achieves between 4.91–50.1x self-relative speedup (30.1x geometric mean). In Fig. 6, it achieves the lowest speedups on the knuth input with permuted weights due to this input tree resulting in a high height dendrogram (h = 2.5M) which is not amenable to our post-processing optimization. Fig. 7, which shows the performance breakdown of ParUF on different inputs shows that on the knuth input with permuted weights, almost all of the time is spent on the asynchronous Union-Find step (the while loop starting on Line 7 in Alg. 5). Although several other inputs have high height (e.g., knuth with unit weights, whose dendrogram forms a path of

Table 1: Parallel running times of our SLD implementations on different tree inputs. The last two columns show the speedup of our implementations over SeqUF.

Type	Sizes	SeqUF	ParUF	RCTT	$\frac{\text{SeqUF}}{\text{ParUF}}$	$\frac{\text{SeqUF}}{\text{RCTT}}$
path	10M	.253	.120	.101	2.10	2.50
	100M	2.20	.962	.897	2.28	2.45
	1B	21.8	7.93	8.15	2.74	2.67
path perm	10M	5.56	.090	.099	61.7	56.1
	100M	68.6	.881	.904	77.8	75.8
	1B	989	6.61	7.49	149.6	132
path low-par	10M	.335	41.9	.101	0.007	3.31
	100M	2.42	366	.884	0.006	2.73
	1B	23.4	2640	7.73	0.008	3.02
star	10M	.252	.125	.111	2.01	2.27
	100M	2.04	1.03	.895	1.98	2.28
	1B	20.3	8.49	7.61	2.39	2.66
star perm	10M	4.71	.141	.116	33.4	40.6
	100M	56.1	1.29	1.01	43.4	55.5
	1B	824	13.9	8.68	59.2	94.9
knuth	10M	1.70	.140	.156	12.1	10.9
	100M	39.2	1.12	1.69	35.0	23.1
	1B	458	9.61	19.2	47.6	23.8
knuth perm	10M	5.83	2.55	.155	2.28	37.6
	100M	79.9	37.7	1.61	2.11	49.6
	1B	1110	95.1	17.3	11.6	64.1

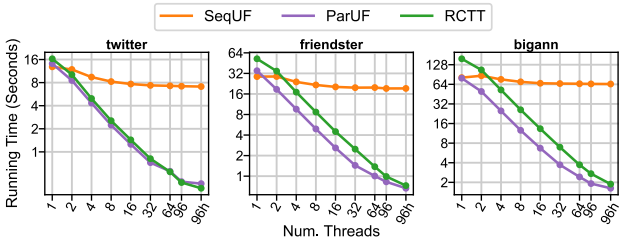

Figure 8: Running time of our SLD implementations on different real-world input trees as a function of the number of threads.

length $n-1$), they are amenable to the post-processing optimization, and thus ParUF achieves good speedup since the post-processing step simply sorts the remaining edges. ParUF typically begins to out-perform SeqUF with more than 8 threads. Compared to SeqUF, as shown in Tab. 1, it obtains between 2.1—150x speedup over SeqUF (5.92x geometric mean speedup) on the billion-scale inputs; however, it performs poorly on the adversarial low-parallelism input (path low-par) it is 151x worse at the 100M scale.

RCTT. The RCTT algorithm achieves the most consistent speedups among the algorithms studied in this paper, achieving between 35.2–75.5x self-speedup (52.1x geometric mean). From Fig. 6 and Tab. 1, we can see that RCTT is usually the fastest algorithm when using all threads, and is always within a factor of 2x of the performance of ParUF. Similar to ParUF, it starts to outperform SeqUF on all inputs after about 8 threads, and is never slower than SeqUF on any input, at any of the scales that we evaluated. Unlike ParUF, whose behavior depends on the amount of parallelism available to it and sometimes performs much worse than SeqUF, RCTT always obtains speedups over SeqUF, achieving between 2.1—132x speedup (16.9x geometric mean speedup) over SeqUF on the billion-scale inputs. We can see from Fig. 7 that despite the *Trace* step being the most costly step in theory (see Sec. 4.2), it takes at most 23% of the time across our inputs, and is usually only a few percentage of the total running time. The majority of the time is spent on the RC tree construction; optimizing this step by designing faster tree contraction algorithms is an interesting direction for future work.

Real-World Inputs. We also ran our implementations on three real-world tree inputs described earlier in this section (results in Figure 8). On these inputs, we observe that SeqUF achieves modest speedups more similar to the permuted weights, rather than the high self-relative speedup in the unit weight case. In particular, it achieves between 1.2–1.8x self-relative speedup. On the other hand, both ParUF and SeqUF achieve strong self-speedups, with ParUF achieving between 36–52x self-speedup and RCTT achieving between 48.7–84x self-speedup. Both of our parallel algorithms achieve strong speedups over SeqUF—on all 192 threads ParUF is between 18.4–39.8x faster, and RCTT is between 21.1–34.4x faster.

6 CONCLUSION

In this paper, we gave optimal parallel algorithms for computing the single-linkage dendrogram. We described a framework for obtaining merge-based divide-and-conquer algorithms, and instantiated the framework using parallel tree contraction, showing that it yields an optimal work deterministic algorithm with poly-logarithmic depth. We also designed two practical algorithms, ParUF and RCTT, both of which have provable guarantees on their work and depth, and which achieve strong speedups over a highly optimized sequential baseline. An interesting question is whether we can extend our approach to obtain good dynamic algorithms for maintaining the single-linkage dendrogram.

ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-2103483 and CNS-2317194, NSF CAREER Award CCF-2339310, the UCR Regents Faculty Award, and the Google Research Scholar Program. We thank the anonymous reviewers for their useful comments.

REFERENCES

[1] Karl Abrahamson, Norm Dadoun, David G. Kirkpatrick, and T Przytycka. 1989. A simple parallel tree contraction algorithm. *Journal of Algorithms* 10, 2 (1989), 287–302.

[2] Umut A Acar, Vitaly Aksenov, and Sam Westrick. 2017. Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[3] Daniel Anderson. 2023. *Parallel Batch-Dynamic Algorithms Dynamic Trees, Graphs, and Self-Adjusting Computation.* Ph. D. Dissertation. Carnegie Mellon University.

[4] Dalya Baron. 2019. Machine Learning in Astronomy: a practical overview. arXiv:1904.07248 [astro-ph.IM]

[5] J-P Benzécri. 1982. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données* 7, 2 (1982), 209–218.

[6] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509. https://cmuparlay.github.io/parlaylib/

[7] Guy E. Blelloch, Laxman Dhulipala, and Yihan Sun. 2021. Introduction to Parallel Algorithms. https://www.cs.cmu.edu/~guyb/paralg/paralg/parallel.pdf. Carnegie Mellon University.

[8] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[9] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Randomized Incremental Convex Hull is Highly Parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 103–115.
[10] Ricardo JGB Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. 2015. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10, 1 (2015), 1–51.
[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*.
[12] Erik D Demaine, Gad M Landau, and Oren Weimann. 2009. On Cartesian trees and range minimum queries. In *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*. Springer, 341–353.
[13] Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. 2021. Hierarchical agglomerative graph clustering in nearly-linear time. In *International Conference on Machine Learning*. PMLR, 2676–2686.
[14] E.D. Feigelson and G.J. Babu. 1998. Statistical Methodology for Large Astronomical Surveys. *Symposium - International Astronomical Union* 179 (1998), 363–370.
[15] Molly Gasperini, Andrew J Hill, José L McFaline-Figueroa, Beth Martin, Seungsoo Kim, Melissa D Zhang, Dana Jackson, Anh Leith, Jacob Schreiber, William S Noble, et al. 2019. A genome-wide framework for mapping gene regulation via cellular genetic screens. *Cell* 176, 1 (2019), 377–390.
[16] Hillel Gazit, Gary L Miller, and Shang-Hua Teng. 1988. Optimal tree contraction in the EREW model. In *Concurrent Computations: Algorithms, Architecture, and Technology*. 139–156.
[17] Markus Götz, Gabriele Cavallaro, Thierry Géraud, Matthias Book, and Morris Riedel. 2018. Parallel computation of component trees on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (2018), 2582–2598.
[18] J. C. Gower and G. J. S. Ross. 1969. Minimum Spanning Trees and Single Linkage Cluster Analysis. *Journal of the Royal Statistical Society. seriesx C (Applied Statistics)* 18, 1 (1969), 54–64.
[19] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.
[20] MohammadTaghi Hajiaghayi, Marina Knittel, Hamed Saleh, and Hsin-Hao Su. 2022. Adaptive Massively Parallel Constant-Round Tree Contraction. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*, Mark Braverman (Ed.), Vol. 215. 83:1–83:23.
[21] Jiří Havel, François Merciol, and Sébastien Lefèvre. 2019. Efficient tree construction for multiscale image representation and processing. *Journal of Real-Time Image Processing* 16 (2019), 1129–1146.
[22] William Hendrix, Diana Palsetia, Md Mostofa Ali Patwary, Ankit Agrawal, Weikeng Liao, and Alok Choudhary. 2013. A scalable algorithm for single-linkage hierarchical clustering on distributed-memory architectures. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 7–13.
[23] David B Henry, Patrick H Tolan, and Deborah Gorman-Smith. 2005. Cluster analysis in family psychology research. *Journal of Family Psychology* 19, 1 (2005), 121.
[24] J. JaJa. 1992. *Introduction to Parallel Algorithms*.
[25] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.
[26] Ivica Letunic and Peer Bork. 2007. Interactive Tree Of Life (iTOL): an online tool for phylogenetic tree display and annotation. *Bioinformatics* 23, 1 (2007), 127–128.
[27] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*.
[28] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 270–285.
[29] Gary L Miller and John H Reif. 1985. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, Vol. 26. 478–489.
[30] Ugo Moschini, Arnold Meijster, and Michael HF Wilkinson. 2017. A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images. *IEEE transactions on pattern analysis and machine intelligence* 40, 3 (2017), 513–526.
[31] Corey J. Nolet, Divye Gala, Alex Fender, Mahesh doixjade, Joe Eaton, Edward Raff, John Zedlewski, Brad Rees, and Tim Oates. 2023. cuSLINK: Single-Linkage Agglomerative Clustering on the GPU. In *ECML PKDD*. 711–726.
[32] Georgios K Ouzounis. 2020. Segmentation strategies for the alpha-tree data structure. *Pattern Recognition Letters* 129 (2020), 232–239.
[33] Georgios K Ouzounis and Pierre Soille. 2012. The alpha-tree algorithm. *JRC Scientific and Policy Report* (2012).
[34] John H. Reif and Stephen R. Tate. 1994. Dynamic parallel tree contraction (extended abstract). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 114–121.
[35] Piyush Sao, Andrey Prokopenko, and Damien Lebrun-Grandié. 2024. PANDORA: A Parallel Dendrogram Construction Algorithm for Single Linkage Clustering on GPU. arXiv:2401.06089 [cs.LG]
[36] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*.
[37] Julian Shun and Guy E. Blelloch. 2014. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Trans. Parallel Comput.* 1, 1 (2014).
[38] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2015. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.
[39] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *Neural Information Processing Systems (NeurIPS)*.
[40] Baris Sumengen, Anand Rajagopalan, Gui Citovsky, David Simcha, Olivier Bachem, Pradipta Mitra, Sam Blasiak, Mason Liang, and Sanjiv Kumar. 2021. Scaling Hierarchical Agglomerative Clustering to Billion-sized Datasets. arXiv:2105.11653 [cs.LG]
[41] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering. In *Proceedings of the 2021 International Conference on Management of Data*. 1982–1995.
[42] Loïc Yengo, Sailaja Vedantam, Eirini Marouli, Julia Sidorenko, Eric Bartell, Saori Sakaue, Marielisa Graff, Anders U Eliasen, Yunxuan Jiang, Sridharan Raghavan, et al. 2022. A saturated map of common genetic variants associated with human height. *Nature* 610, 7933 (2022), 704–712.
[43] Odilia Yim and Kylee T Ramdeen. 2015. Hierarchical cluster analysis: comparison of three linkage measures and application to psychological data. *The Quantitative Methods for Psychology* 11, 1 (2015), 8–21.
[44] Shangdi Yu, Yiqiu Wang, Yan Gu, Laxman Dhulipala, and Julian Shun. 2021. ParChain: a framework for parallel hierarchical agglomerative clustering using nearest-neighbor chain. *Proc. VLDB Endow.* 15, 2 (2021), 285–298.

A RELATED WORK

Single-linkage clustering has been studied for over half a century, starting with the early work of Gower and Ross [18]. Since then, it has found widespread application in a variety of scientific disciplines and industrial applications [4, 14, 15, 17, 21, 23, 26, 27, 33, 42, 43].

Single-Linkage Dendrogram Algorithms. In the past decade, due to the importance of single-linkage clustering, serious algorithmic consideration of the core problem of computing a single-linkage dendrogram began. The work of Demaine et al. [12] gives an algorithm showing that if the cost of sorting the edges is "free", SLD can be solved in $O(n)$ time. Their algorithm is based on an nice argument using decremental tree connectivity. Their linear-work bound is not directly comparable to our results since they assume the edges are sorted (and thus bypass comparison lower bounds).

In more recent years, different communities have studied the SLD problem, and other related hierarchical tree building problems. A large body of work has come out of the image analysis community, where SLD is studied under the moniker of "alpha-tree" algorithms [17, 30, 32, 33]. Unfortunately the algorithms are highly specific to analyzing 2D and 3D data, and parallel algorithms in this domain [17, 30] are not work-efficient and typically not rigorously analyzed in parallel models.

The most relevant related work is the paper of Wang et al. [41] who recently gave the first work-efficient parallel algorithm for SLD. Their algorithm is randomized and computes the SLD in $O(n \log n)$ expected work and $O(\log^2 n \log\log n)$ depth with high probability (*whp*). Although their algorithm is work-efficient with respect to SeqUF, it relies on applying divide-and-conquer over the weights,

which is implemented using the Euler Tour Technique [24]. Based on private communication with the authors, we understand that due to its complicated nature, this algorithm does not consistently outperform the simple SeqUF algorithm. The authors only released the code for SeqUF and suggested to always use SeqUF rather than the theoretically-efficient algorithm. The algorithm is randomized due to the use of semisort [19, 41], and there is no clear way to derandomize it to obtain a deterministic parallel algorithm.

Our paper gives two algorithms that have better work than the algorithm of Wang et al. [41], but have worse depth bounds since the depth of our tree contraction algorithm is $O(\log^2 n \log^2 h) = \Omega(\log^2 n (\log\log n)^2)$. However, our third algorithm (RCTT) provides a strict improvement over their algorithm, while also being simple and deterministic, since RCTT achieves $O(n \log n)$ work and $O(\log^2 n)$ depth in the binary-forking model. We note that in the PRAM model, the RCTT algorithm requires $O(n \log n)$ work and $O(\log n)$ depth. Improving RCTT to obtain $O(n \log h)$ work and $O(\log n)$ depth or $O(n)$ work and polylog(n) depth if the edges are pre-sorted are two interesting directions for future work.

B LOWER BOUND

Lemma 3.6. *For any $\lfloor \log n \rfloor \leq h \leq n-1$, there is an input that every comparison-based SLD algorithm requires $\Omega(n \log h)$ work to compute the parent of every edge in the output dendrogram.*

Proof. For a given value of h, we build a tree with n/h connected components, each with h elements. The goal of each component is to solve an independent instance of sorting, which has a comparison-sorting lower bound of $\Omega(h \log h)$ work [11] to solve. To create an input tree for our comparison-based SLD algorithm to solve, we connect all elements within each component into a star. It is not hard to see that after solving SLD, the elements in each star will be totally ordered based on their rank, and so solving SLD on the aforementioned input tree will solve all of the sorting instances. Since each sorting instance requires $\Omega(h \log h)$ comparisons (work), the n/h instances requires $\Omega(n \log h)$ work in total to solve. □