Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering[*]

Yiqiu Wang
MIT CSAIL
yiqiuw@mit.edu

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

This paper presents new parallel algorithms for generating Euclidean minimum spanning trees and spatial clustering hierarchies (known as HDBSCAN*). Our approach is based on generating a well-separated pair decomposition followed by using Kruskal's minimum spanning tree algorithm and bichromatic closest pair computations. We introduce a new notion of well-separation to reduce the work and space of our algorithm for HDBSCAN*. We also give a new parallel divide-and-conquer algorithm for computing the dendrogram and reachability plots, which are used in visualizing clusters of different scale that arise for both EMST and HDBSCAN*. We show that our algorithms are theoretically efficient: they have work (number of operations) matching their sequential counterparts, and polylogarithmic depth (parallel time).

We implement our algorithms and propose a memory optimization that requires only a subset of well-separated pairs to be computed and materialized, leading to savings in both space (up to 10x) and time (up to 8x). Our experiments on large real-world and synthetic data sets using a 48-core machine show that our fastest algorithms outperform the best serial algorithms for the problems by 11.13–55.89x, and existing parallel algorithms by at least an order of magnitude.

ACM Reference Format:
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 (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457296

1 Introduction

This paper studies the two related geometric problems of Euclidean minimum spanning tree (EMST) and hierarchical density-based spatial clustering with added noise [14]. The problems take as input a set of n points in a d-dimensional space. EMST computes a minimum spanning tree on a complete graph formed among the n points with edges between two points having the weight equal to their Euclidean distance. EMST has many applications, including in single-linkage clustering [28], network placement optimization [53], and approximating the Euclidean traveling salesman problem [52].

Hierarchical density-based spatial clustering of applications with noise (*HDBSCAN**) is a popular hierarchical clustering algorithm [14]. The goal of density-based spatial clustering is to cluster points that are in dense regions and close together in proximity. One of the most widely-used density-based spatial clustering methods is the *density-based spatial clustering of applications with noise* (*DBSCAN*) method by Ester et al. [20]. DBSCAN requires two parameters, ϵ and minPts, which determine what is considered "close" and "dense", respectively. In practice, minPts is usually fixed to a small constant, but many different values of ϵ need to be explored in order to find high-quality clusters. Many efficient DBSCAN algorithms have been designed both for the sequential [16, 19, 24, 30] and the parallel context (both shared memory and distributed memory) [27, 32, 37, 44, 51, 54]. To avoid repeatedly executing DBSCAN for different values of ϵ, the OPTICS [7] and HDBSCAN* [14] algorithms have been proposed for constructing DBSCAN clustering hierarchies, from which clusters from different values of ϵ can be generated. These algorithms are known to be robust to outliers in the data set. The algorithms are based on generating a minimum spanning tree on the input points, where a subset of the edge weights are determined by Euclidean distance and the remaining edge weights are determined by a DBSCAN-specific metric known as the core distance (to be defined in Section 2). Thus, the algorithms bear some similarity to EMST algorithms.

There has been a significant amount of theoretical work on designing fast sequential EMST algorithms (e.g., [6, 8, 12, 48, 56]). There have also been some practical implementations of EMST [9, 15, 38, 41], although most of them are sequential (part of the algorithm by Chatterjee et al. [15] is parallel). The state-of-the-art EMST implementations are either based on generating a well-separated pair decomposition (WSPD) [13] and applying Kruskal's minimum spanning tree (MST) algorithm on edges produced by the WSPD [15, 41], or dual-tree traversals on k-d trees integrated into Boruvka's MST algorithm [38]. Much less work has been proposed for parallel HDBSCAN* and OPTICS [45, 47]. In this paper, we design new algorithms for EMST, which can also be leveraged to design a fast parallel HDBSCAN* algorithm.

This paper presents practical parallel in-memory algorithms for EMST and HDBSCAN*, and proves that the theoretical work (number of operations) of our implementations matches their state-of-the-art counterparts, while having polylogarithmic depth.[1] Our algorithms are based on finding a WSPD and then running Kruskal's algorithm on edges between pairs in the WSPD. For our HDBSCAN* algorithm, we propose a new notion of well-separation to include

[*]The full version of this paper can be found at http://arxiv.org/abs/2104.01126.

[1]The work is the total number of operations and depth (parallel time) is the length of the longest sequential dependence.

SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457296

the notion of core distances, which enables us to improve the space usage and work of our algorithm.

Given the MST from the EMST or the HDBSCAN* problem, we provide an algorithm to generate a dendrogram, which represents the hierarchy of clusters in our data set. For EMST, this solves the single-linkage clustering problem [28], and for HDBSCAN*, this gives us a dendrogram as well as a reachability plot [14]. We introduce a work-efficient[2] parallel divide-and-conquer algorithm that first generates an Euler tour on the tree, splits the tree into multiple subtrees, recursively generates the dendrogram for each subtree, and glues the results back together. An in-order traversal of the dendrogram gives the reachability plot. Our algorithm takes $O(n \log n)$ work and $O(\log^2 n \log \log n)$ depth. Our parallel dendrogram algorithm is of independent interest, as it can also be applied to generate dendrograms for other clustering problems.

We provide optimized parallel implementations of our EMST and HDBSCAN* algorithms. We introduce a memory optimization that avoids computing and materializing many of the WSPD pairs, which significantly improves our algorithm's performance (up to 8x faster and 10x less space). We also provide optimized implementations of k-d trees, which our algorithms use for spatial queries.

We perform a comprehensive set of experiments on both synthetic and real-world data sets using varying parameters, and compare the performance of our implementations to optimized sequential implementations as well as existing parallel implementations. Compared to existing EMST sequential implementations [38, 39], our fastest sequential implementation is 0.89–4.17x faster (2.44x on average). On a 48-core machine with hyper-threading, our EMST implementation achieves 14.61–55.89x speedup over the fastest sequential implementations. Our HDBSCAN* implementation achieves 11.13–46.69x speedup over the fastest sequential implementations. Compared to existing sequential and parallel implementations for HDBSCAN* [25, 39, 45, 47], our implementation is at least an order of magnitude faster. Our source code is publicly available at https://github.com/wangyiqiu/hdbscan.

We summarize our contributions below:

- New parallel algorithms for EMST and HDBSCAN* with strong theoretical guarantees.
- A new definition of well-separation that computes the HDBSCAN* MST using asymptotically less space.
- Memory-optimized parallel implementations for EMST and HDBSCAN* that give significant space and time improvements.
- A new parallel algorithm for dendrogram construction.
- A comprehensive experimental study of the proposed methods.

2 Preliminaries

2.1 Problem Definitions

EMST. The ***Euclidean Minimum Spanning Tree (EMST)*** problem takes n points $\mathcal{P} = \{p_1, \ldots, p_n\}$ and returns a minimum spanning tree (MST) of the complete undirected Euclidean graph of $\mathcal{P}$.

DBSCAN*. The ***DBSCAN**** (density-based spatial clustering of applications with noise) problem takes as input n points $\mathcal{P} = \{p_1, \ldots, p_n\}$, a distance function d, and two parameters ϵ and minPts [14, 20]. A point p is a ***core point*** if and only if $|\{p_i \mid p_i \in \mathcal{P}, d(p, p_i) \le \epsilon\}| \ge \mathsf{minPts}$. A point is called a ***noise point*** otherwise. We denote the set of core points as $\mathcal{P}_{core}$. DBSCAN* computes a partition of $\mathcal{P}_{core}$, where each subset is referred to as a ***cluster***, and also returns the remaining points as noise points. Two points $p, q \in \mathcal{P}_{core}$ are in the same cluster if and only if there exists a list of points $p = \bar{p}_1, \bar{p}_2, \ldots, \bar{p}_{k-1}, \bar{p}_k = q$ in $\mathcal{P}_{core}$ such that $d(\bar{p}_{i-1}, \bar{p}_i) \le \epsilon$ for all $1 < i \le k$. For a given set of points and two parameters ϵ and minPts, the clusters returned are unique.[3]

HDBSCAN*. The ***HDBSCAN**** (hierarchical DBSCAN*) problem [14] takes the same input as DBSCAN*, but without the ϵ parameter, and computes a hierarchy of DBSCAN* clusters for all possible values of ϵ. The ***core distance*** of a point p, $\mathrm{cd}(p)$, is the distance from p to its minPts-nearest neighbor (including p itself). The ***mutual reachability distance*** between two points p and q is defined to be $d_m(p, q) = max\{\mathrm{cd}(p), \mathrm{cd}(q), d(p, q)\}$. The ***mutual reachability graph*** G_{MR} is a complete undirected graph, where the vertices are the points in $\mathcal{P}$, and the edges are weighted by the mutual reachability distances.[4]

The HDBSCAN* hierarchy is sequentially computed in two steps [14]. The first step computes an MST of G_{MR} and then adds a self-edge to each vertex weighted by its core distance. An example MST is shown in Figure 1a. We note that the HDBSCAN* MST with $\mathsf{minPts} = 1$ is equivalent to the EMST, since the mutual reachability distance at $\mathsf{minPts} = 1$ is equivalent to the Euclidean distance. A dendrogram representing clusters at different values of ϵ is computed by removing edges from the MST plus self-edges graph in decreasing order of weight. The root of the dendrogram is a cluster containing all points. Each non-self-edge removal splits a cluster into two, which become the two children of the cluster in the dendrogram. The height of the split cluster in the dendrogram is equal to the weight of the removed edge. If the removed edge is a self-edge, we mark the component (point) as a noise point. An example of a dendrogram is shown in Figure 1b. If we want to return the clusters for a particular value of ϵ, we can horizontally cut the dendrogram at that value of ϵ and return the resulting subtrees below the cut as the clusters or noise points. This is equivalent to removing edges from the MST of G_{MR} with weight greater than ϵ.

For HDBSCAN*, the reachability plot (OPTICS sequence) [7] contains all points in $\mathcal{P}$ in some order $\{p_i | \, i = 1, \ldots, n\}$, where each point p_i is represented as a bar with height $\min\{d_m(p_i, p_j) \mid j < i\}$. For HDBSCAN*, the order of the points is the order that they are visited in an execution of Prim's algorithm on the MST of G_{MR} starting from an arbitrary point [7]. An example is shown in Figure 1c. Intuitively, the "valleys" of the reachability plot correspond to clusters [14].

2.2 Parallel Primitives

We use the classic ***work-depth model*** for analyzing parallel shared-memory algorithms [18, 34, 35]. The ***work*** W of an algorithm is the number of instructions in the computation, and the ***depth*** D is the longest sequential dependence. Using Brent's scheduling theorem [10], we can execute a parallel computation in

[2]A ***work-efficient*** parallel algorithm has a work bound that matches the best sequential algorithm for the problem.

[3]The original DBSCAN definition includes the notion of border points, which are non-core points that are within a distance of ϵ to core points [20]. DBSCAN* chooses to omit this to be more consistent with a statistical interpretation of clusters [14].

[4]The related OPTICS problem also generates a hierarchy of clusters but with a definition of reachability distance that is asymmetric, leading to a directed graph [7].

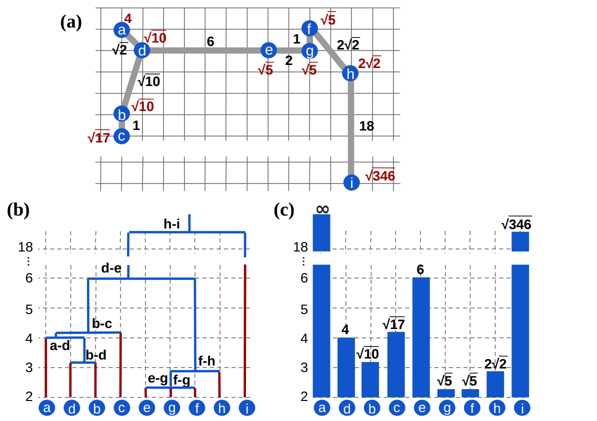

Figure 1: (a) An MST of the HDBSCAN* mutual reachability graph on an example data set in 2D. The red number next to each point is the core distance of the point for minPts $= 3$. The Euclidean distances between points are denoted by grey edges, whose values are marked in black. For example, a's core distance is 4 because b is a's third nearest neighbor (including itself) and $d(a, b) = 4$. The edge weight of (a, d) is $\max\{4, \sqrt{10}, \sqrt{2}\} = 4$. (b) An HDBSCAN* dendrogram for the data set. A point becomes a noise point when its vertical line becomes red. For example, if we cut the dendrogram at $\epsilon = 3.5$, then we have two clusters $\{d, b\}$ and $\{e, g, f, h\}$, while a, c and i are noise points. (c) A reachability plot for the data set starting at point a. The two "valleys", $\{a, b, c, d\}$ and $\{e, f, g, h\}$, are the two most obvious clusters.

$W/p + D$ running time using p processors. In practice, we can use randomized work-stealing schedulers that are available in existing languages such as Cilk, TBB, X10, and Java Fork-Join. We assume that priority concurrent writes take $O(1)$ work and depth.

Prefix sum takes as input a sequence $[a_1, a_2, \ldots, a_n]$, an associative binary operator $\oplus$, and an identity element i, and returns the sequence $[i, a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$ as well as the overall sum (using $\oplus$) of the elements. ***Filter*** takes an array A and a predicate function f, and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order that they appear in A. Filter can be implemented using prefix sum. ***Split*** takes an array A and a predicate function f, and moves all of the "true" elements before the "false" elements. Split can be implemented using filter. The ***Euler tour*** of a tree takes as input an adjacency list representation of the tree and returns a directed circuit that traverses every edge of the tree exactly once. ***List ranking*** takes a linked list with values on each node and returns for each node the sum of values from the node to the end of the list. All of the above primitives can be implemented in $O(n)$ work and $O(\log n)$ depth [34]. ***Semisort*** [29] takes as input n items, each with a key, and groups the items with the same key together, without any guarantee on the ordering of items with different keys. This algorithm takes $O(n)$ expected work and $O(\log n)$ depth with high probability. A parallel ***hash table*** supports n inserts, deletes, and finds in $O(n)$ work and $O(\log n)$ depth with high probability [26]. WriteMin is a priority concurrent write that takes as input two arguments, where the first argument is the location to write to and the second argument is the value to write; on concurrent writes, the smallest value is written [50].

2.3 Relevant Techniques

k-NN Query. A ***k-nearest neighbor (k-NN) query*** takes a point data set $\mathcal{P}$ and a distance function, and returns for each point in $\mathcal{P}$ its k nearest neighbors (including itself). Callahan and Kosaraju [11] show that k-NN queries in Euclidean space for all points can be solved in parallel in $O(kn \log n)$ work and $O(\log n)$ depth.

kd-tree. A ***kd-tree*** is a commonly used data structure for k-NN queries [22]. It is a binary tree that is constructed recursively: each node in the tree represents a set of points, which are partitioned between its two children by splitting along one of the dimensions; this process is recursively applied on each of its two children until a leaf node is reached (a leaf node is one that contains at most c points, for a predetermined constant c). It can be constructed in parallel by processing each child in parallel. A k-NN query can be answered by traversing nodes in the tree that are close to the input point, and pruning nodes further away that cannot possibly contain the k nearest neighbors.

BCCP and BCCP*. Existing algorithms, as well as some of our new algorithms, use subroutines for solving the ***bichromatic closest pair (BCCP)*** problem, which takes as input two sets of points, A and B, and returns the pair of points p_1 and p_2 with minimum distance between them, where $p_1 \in A$ and $p_2 \in B$. We also define a variant, the ***BCCP**** problem, that finds the pair of points with the minimum mutual reachability distance, as defined for HDBSCAN*.

Well-Separated Pair Decomposition. We use the same definitions and notations as in Callahan and Kosaraju [13]. Two sets of points, A and B, are ***well-separated*** if A and B can each be contained in spheres of radius r, and the minimum distance between the two spheres is at least sr, for a ***separation constant*** s (we use $s = 2$ throughout the paper). An ***interaction product*** of point sets A and B is defined to be $A \otimes B = \{\{p, p'\} | \ p \in A, \ p' \in B, \ p \neq p'\}$. The set $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ is a ***well-separated realization*** of $A \otimes B$ if: **(1)** $A_i \subseteq A$ and $B_i \subseteq B$ for all $i = 1, ..., k$; **(2)** $A_i \cap B_i = \emptyset$ for all $i = 1, ..., k$; **(3)** $(A_i \otimes B_i) \bigcap (A_j \otimes B_j) = \emptyset$ for all i, j where $1 \leq i < j \leq k$; **(4)** $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$; **(5)** A_i and B_i are well-separated for all $i = 1, ..., k$.

For a point set $\mathcal{P}$, a ***well-separated pair decomposition (WSPD)*** is a well-separated realization of $\mathcal{P} \otimes \mathcal{P}$. We discuss how to construct a WSPD using a kd-tree in Section 3.

Notation. Table 1 shows notation frequently used in the paper.

Notation	Definition
$d(p, q)$	Euclidean distance between points p and q.
$d_m(p, q)$	Mutual reachability distance between points p and q.
$d(A, B)$	Minimum distance between the bounding spheres of points in tree node A and points in tree node B.
$w(u, v)$	Weight of edge (u, v).
A_{diam}	Diameter of the bounding sphere of points in tree node A.
$\text{cd}_{\min}(A)$	Minimum core distance of points in tree node A.
$\text{cd}_{\max}(A)$	Maximum core distance of points in tree node A.

Table 1: Summary of Notation

3 Parallel EMST and HDBSCAN*

In this section, we present our new parallel algorithms for EMST and HDBSCAN*. We also introduce our new memory optimization to improve space usage and performance in practice.

3.1 EMST

To solve EMST, Callahan and Kosaraju present an algorithm for constructing a WSPD that creates an edge between the BCCP of each pair in the WSPD with weight equal to their distance, and then runs an MST algorithm on these edges. They show that their algorithm takes $O(T_d(n, n) \log n)$ work [12], where $T_d(n, n)$ refers to the work of computing BCCP on two sets each of size n.

Algorithm 1 Well-Separated Pair Decomposition

```
1:  procedure WSPD($A$)
2:      if $|A| > 1$ then
3:          do in parallel
4:              WSPD($A_{left}$)              ▷ parallel call on the left child of $A$
5:              WSPD($A_{right}$)             ▷ parallel call on the right child of $A$
6:          FINDPAIR($A_{left}$, $A_{right}$)
7:  procedure FINDPAIR($P$, $P'$)
8:      if $P_{\text{diam}} < P'_{\text{diam}}$ then
9:          SWAP($P$, $P'$)
10:     if WELLSEPARATED($P$, $P'$) then RECORD($P$, $P'$)
11:     else
12:         do in parallel
13:             FINDPAIR($P_{left}$, $P'$)     ▷ $P_{left}$ is the left child of $P$
14:             FINDPAIR($P_{right}$, $P'$)    ▷ $P_{right}$ is the right child of $P$
```

Algorithm 2 Parallel GeoFilterKruskal

```
1:  procedure PARALLELGFK(WSPD: $S$, Edges: $E_{out}$, UnionFind: $UF$)
2:      $\beta = 2$
3:      while $|E_{out}| < (n-1)$ do
4:          $(S_l, S_u) =$ SPLIT($S$, $f_\beta$)  ▷ For a pair $(A, B)$, $f_\beta$ checks if $|A| + |B| \leq \beta$
5:          $\rho_{hi} = \min_{(A,B) \in S_u} d(A, B)$
6:          $(S_{l1}, S_{l2}) =$ SPLIT($S_l$, $f_{\rho_{hi}}$)   ▷ For a pair $(A, B)$, $f_{\rho_{hi}}$ checks if
    BCCP$(A, B) \leq \rho_{hi}$
7:          $E_{l1} =$ GETEDGES($S_{l1}$)   ▷ Retrieves edges associated with pairs in $S_{l1}$
8:          PARALLELKRUSKAL($E_{l1}$, $E_{out}$, $UF$)
9:          $S =$ FILTER($S_{l2} \cup S_u$, $f_{diff}$)  ▷ For a pair $(A, B)$, $f_{diff}$ checks points in $A$
    are in different component from $B$ in $UF$
10:         $\beta = \beta \times 2$
```

For our parallel EMST algorithm, we parallelize WSPD construction algorithm, and then develop a parallel variant of Kruskal's MST algorithm that runs on the edges formed by the pairs in the WSPD. We also propose a non-trivial optimization to make the implementation fast and memory-efficient.

3.1.1 Constructing a WSPD in Parallel We introduce the basic parallel WSPD in Algorithm 1. Prior to calling WSPD, we construct a spatial median kd-tree T in parallel with each leaf containing one point. Then, we call the procedure WSPD on Line 1 and make the root node of T its input. In WSPD, we make parallel calls to FINDPAIR on the two children of all non-leaf nodes by recursively calling WSPD. The procedure FINDPAIR on Line 7 takes as input a pair (P, P') of nodes in T, and checks whether P and P' are well-separated. If they are well-separated, then the algorithm records them as a well-separated pair on Line 10; otherwise, the algorithm splits the set with the larger bounding sphere into its two children and makes two recursive calls in parallel (Lines 13–14). This process is applied recursively until the input pairs are well-separated. The major difference of Algorithm 1 from the serial version is the parallel thread-spawning on Lines 3–5 and 12–14. This procedure generates a WSPD with $O(n)$ pairs [12].

3.1.2 Parallel GFK Algorithm for EMST The original algorithm by Callahan and Kosaraju [12] computes the BCCP between each pair in the WSPD to generate a graph from which an MST can be computed to obtain the EMST. However, it is not necessary to compute the BCCP for all pairs, as observed by Chatterjee et al. [15]. Our implementation only computes the BCCP between a pair if their points are not yet connected in the spanning forest generated so far. This optimization reduces the total number of BCCP calls. Furthermore, we propose a memory optimization that avoids materializing all of the pairs in the WSPD. We will first

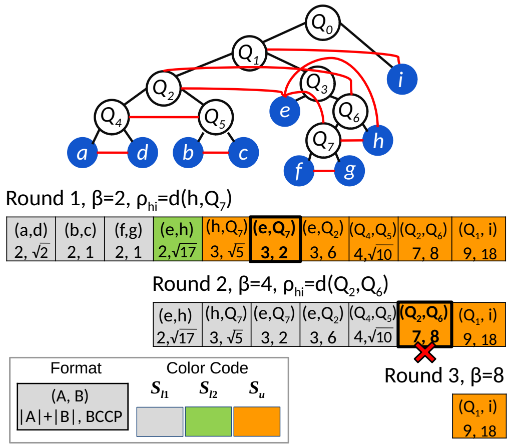

Figure 2: The is an example for both GFK (Algorithm 2) and MEMOGFK (Algorithm 3) for EMST corresponding to the data set shown in Figure 1. The red lines linking tree nodes and the boxes drawn below represent well-separated pairs. The boxes also show the cardinality and BCCP value of the pair. Their correspondence with the symbols S_{l1}, S_{l2}, and S_u from the pseudocode are color-coded. The pairs that generate ρ_{hi} are in bold squares, and the pairs filtered out have a red cross. Using our MemoGFK optimization, only the pairs in S_{l1} needs to be materialized, in contrast to needing to materialize all of the pairs in GFK.

describe how we obtain the EMST from the WSPD, and then give details of our memory optimization.

The original Kruskal's algorithm is an MST algorithm that takes input edges sorted by non-decreasing weight, and processes the edges in order, using a union-find data structure to join components for edges with endpoints in different components. Our implementation is inspired by a variant of Kruskal's algorithm, GeoFilterKruskal (GFK). This algorithm was used for sequential EMST by Chatterjee et al. [15], and for MST in general graphs by Osipov et al. [43]. It improves Kruskal's algorithm by avoiding the BCCP computation between pairs unless needed, and prioritizing BCCPs between pairs with smaller cardinalities, which are cheaper, with the goal of pruning more expensive BCCP computations.

We propose a parallel GFK algorithm as shown in Algorithm 2. It uses Kruskal's MST algorithm as a subroutine by passing it batches of edges, where each batch has edges with weights no less than those of edges in previous batches, and the union-find structure is shared across multiple invocations of Kruskal's algorithm. PARALLELGFK takes as input the WSPD pairs S, an array E_{out} to store the MST edges, and a union-find structure UF. On each round, given a constant β, we only consider node pairs in the WSPD with cardinality (sum of sizes) at most β because it is cheaper to compute their BCCPs. To do so, the set of pairs S is partitioned into S_l, containing pairs with cardinality at most β, and S_u, containing the remaining pairs (Line 4). However, it is only correct to consider pairs in S_l that produce edges lighter than any of the pairs in S_u. On Line 5, we compute an upper bound ρ_{hi} for the edges in S_l by setting ρ_{hi} equal to the minimum $d(A, B)$ for all $(A, B) \in S_u$ (this is a lower bound on the edges weights formed by these pairs). In the example shown in Figure 2, in the first round, with $\beta = 2$, the set S_l contains (a, d), (b, c), (f, g), and (e, h), and the set S_u contains (h, Q_7), (e, Q_7), (e, Q_2), (Q_4, Q_5), (Q_2, Q_6), and (Q_1, i). ρ_{hi} corresponds to

(e, Q_7) on Line 5. Then, we compute the BCCP of all elements of set S_l, and split it into S_{l1} and S_{l2}, where S_{l1} has edges with weight at most ρ_{hi} (Line 6). On Line 6, S_{l1} contains (a, d), (b, c) and (f, g), as their BCCP distances are smaller than $\rho_{hi} = d(e, Q_7)$, and S_{l2} contains (e, h) . After that, E_{l1}, the edges corresponding to S_{l1}, are passed to Kruskal's algorithm (Lines 7–8). The remaining pairs $S_{l2} \cup S_u$ are then filtered based on the result of Kruskal's algorithm (Line 9)—in particular, pairs that are connected in the union-find structure of Kruskal's algorithm can be discarded, and for many of these pairs we never have to compute their BCCP. In Figure 2, the second round processes (e, h), (h, Q_7), (e, Q_7), (e, Q_2), (Q_4, Q_5), (Q_2, Q_6), and (Q_1, i), and works similarly to Round 1. However, (Q_2, Q_6) gets filtered out during the second round, and we never have to compute its BCCP, leading to less work compared to a naive algorithm. Finally, the subsequent rounds process a single pair (Q_1, i). At the end of each round, we double the value of β to ensure that there are logarithmic number of rounds and hence better depth (in contrast, the sequential algorithm of Chatterjee et al. [15] increases β by 1 every round). Throughout the algorithm, we cache the BCCP results of pairs to avoid repeated computations. Overall, the main difference between Algorithm 2 and sequential algorithm is the use of parallel primitives on nearly every line of the pseudocode, and the exponentially increasing value of β on Line 11, which is crucial for achieving a low depth bound.

The following theorem summarizes the bounds of our algorithm.

Theorem 3.1. *We can compute the EMST on a set of n points in constant dimensions in $O(n^2)$ work and $O(\log^2 n)$ depth.*

Proof. Callahan [11] shows that a WSPD with $O(n)$ well-separated pairs can be computed in $O(n \log n)$ work and $O(\log n)$ depth, which we use for our analysis. Our parallel GeoFilterKruskal algorithm for EMST proceeds in rounds, and processes the well-separated pairs in an increasing order of cardinality. Since β doubles on each round, there can be at most $O(\log n)$ rounds since the largest pair can contain n points. Within each round, the Split on Line 4 and Filter on Line 9 both take $O(n)$ work and $O(\log n)$ depth. We can compute the BCCP for each pair on Line 6 by computing all possible point distances between the pair, and using WriteMin to obtain the minimum distance. Since the BCCP of each pair will only be computed once and is cached, the total work of BCCP on Line 6 is $\sum_{A,B \in S} |A||B| = O(n^2)$ work since the WSPD is an exact set cover for all distinct pairs. Therefore, Line 6 takes $O(n^2)$ work across all rounds and $O(1)$ depth for each round. Given n edges, the MST computation on Line 8 can be done in $O(n \log n)$ work and $O(\log n)$ depth using existing parallel algorithms [34]. Therefore, the overall work is $O(n^2)$. Since each round takes $O(\log n)$ depth, and there are $O(\log n)$ rounds, the overall depth is $O(\log^2 n)$. □

We implemented our own sequential and parallel versions of the GFK algorithm as a baseline based on Algorithm 2, which we found to be faster than the implementation of Chatterjee et al. [15] in our experiments. In addition, because the original GFK algorithm requires materializing the full WSPD, its memory consumption can be excessive, limiting the algorithm's practicality. This issue worsens as the dimensionality of the points increases, as the number of pairs in the WSPD increases exponentially with the dimension. While Chatterjee et al. [15] show that their GFK algorithm is efficient, they consider much smaller data sets than the ones in this paper.

Algorithm 3 Parallel MemoGFK

```
1: procedure ParallelMemoGFK(kd-tree root: $R$, Edges: $E_{out}$, UnionFind: $UF$)
2:     $\beta = 2$, $\rho_{lo} = 0$
3:     while $|E_{out}| < (n-1)$ do
4:         $\rho_{hi}$ = GetRho($R$, $\beta$)
5:         $S_{l1}$ = GetPairs($R$, $\beta$, $\rho_{lo}$, $\rho_{hi}$, $UF$)
6:         $E_{l1}$ = GetEdges($S_{l1}$)    ▷ Retrieves edges associated with pairs in $S_{l1}$
7:         ParallelKruskal($E_{l1}$, $E_{out}$, $UF$)
8:         $\beta = \beta \times 2$, $\rho_{lo} = \rho_{hi}$
```

3.1.3 The MemoGFK Optimization To tackle the memory consumption issue, we propose an optimization to the GFK algorithm, which reduces its space usage and improves its running time in practice. We call the resulting algorithm ***MemoGFK*** (memory-optimized GFK). The basic idea is that, rather than materializing the full WSPD at the beginning, we partially traverse the kd-tree on each round and retrieve only the pairs that are needed. The pseudocode for our algorithm is shown in Algorithm 3, where ParallelMemoGFK takes in the root R of a kd-tree, an array E_{out} to store the MST edges, and a union-find structure UF.

The algorithm proceeds in rounds similar to parallel GeoFilterKruskal, and maintains lower and upper bounds (ρ_{lo} and ρ_{hi}) on the weight of edges to be considered each round. On each round, it first computes ρ_{hi} based on β by a single kd-tree traversal, which will be elaborated below (Line 4). Then, together with ρ_{lo} from the previous round ($\rho_{lo} = 0$ on the first round), the algorithm retrieves pairs with BCCP distance in the range $[\rho_{lo}, \rho_{hi})$ via a second kd-tree traversal on Line 5. The edges corresponding to these pairs are then passed to Kruskal's algorithm on Line 7. An example of the first round of the algorithm with MemoGFK is illustrated in Figure 2. Without the optimization, the GFK algorithm needs to first materialize all of the pairs in Round 1. With MemoGFK, $\rho_{hi} = d(e, Q_7)$ is computed via a tree traversal on Line 4, after which only the pairs in the set $S_{l1} = \{(a, d), (b, c), (f, g)\}$ are retrieved and materialized on Line 5 via a second tree traversal. Retrieving pairs only as needed reduces memory usage and improves performance. The correctness of the algorithm follows from the fact that each round considers non-overlapping ranges of edge weights in increasing order until all edges are considered, or when MST is completed.

Now we discuss the implementation details of the two-pass tree traversal on Line 4–5. The GetRho subroutine, which computes ρ_{hi}, does so by finding the lower bound on the minimum separation of pairs whose cardinality is greater than β and are not yet connected in the MST. We traverse the kd-tree starting at the root, in a similar way as when computing the WSPD in Algorithm 1. During the process, we update a global copy of ρ_{hi} using WriteMin whenever we encounter a well-separated pair in FindPair, with cardinality greater than β. We can prune the traversal once $|A| + |B| \leq \beta$, as all pairs that originate from (A, B) will have cardinality at most β. We also prune the traversal when the two children of a tree node are already connected in the union-find structure, as these edges will not need to be considered by Kruskal's algorithm. In addition, we prune the traversal when the distance between the bounding spheres of A and B, $d(A, B)$, is larger than ρ_{hi}, as its descendants cannot produce a smaller distance.

The GetPairs subroutine then retrieves all pairs whose points are not yet connected in the union-find structure and have BCCP distances in the range $[\rho_{lo}, \rho_{hi})$. It does so also via a pruned traversal on the kd-tree starting from the root, similarly to Algorithm 1,

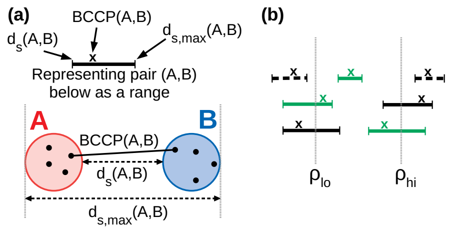

Figure 3: (a) shows a representation of a well-separated pair (A, B) as a line segment, based on the values of its $d(A, B)$ and $d_{\max}(A, B)$, which serve as the lower and upper bounds, respectively, for their BCCP and the BCCP of their descendants. The "x"'s on the line marks the value of the BCCP. (b) shows an example of tree node pairs encountered during a pruned tree traversal on Line 5 of Algorithm 3, where the pairs are represented the same way as in (a). The pairs in solid green lines, if well-separated, will be retrieved and materialized because their BCCPs are within the $[\rho_{lo}, \rho_{hi})$ range, whereas those in solid black lines will not as their BCCPs are out of range (although their BCCPs will still be computed, since their lower and upper bounds do not immediately put them out of range). The traversal will be pruned when encountering a pair represented by dotted lines as their BCCP and the BCCP of their descendants will be out of range.

but only retrieves the useful pairs. For a pair of nodes encountered in the FindPair subroutine, we estimate the minimum and maximum possible BCCP between the pair using bounding sphere calculations, an example of which is shown in Figure 3a. We prune the traversal when $d_{\max}(A, B) < \rho_{lo}$, or when $d(A, B) \geq \rho_{hi}$, in which case BCCP(A, B) (as well as those of its recursive calls on descendant nodes) will be outside of the range. An example is shown in Figure 3b. In addition, we also prune the traversal if A and B are already connected in the MST, as an edge between A and B will not be part of the MST.

We evaluate MemoGFK in Section 5. We also use the memory optimization for HDBSCAN*, which will be described next.

3.2 HDBSCAN*

3.2.1 Baseline Inspired by a sequential approximate algorithm to solve the OPTICS problem by Gan and Tao [25], we modified and parallelized their algorithm to compute the exact HDBSCAN* as our baseline. First, we perform k-NN queries using Euclidean distance with $k = \text{minPts}$ to compute the core distances. Gan and Tao's original algorithm creates a mutual reachability graph of size $O(n \cdot \text{minPts}^2)$, using an approximate notion of BCCP between each WSPD pair, and then computes its MST using Prim's algorithm. Our exact algorithm parallelizes their algorithm, and instead uses the exact BCCP* computations based on the mutual reachability distance to form the mutual reachability graph. In addition, we also compute the MST on the generated edges using the MemoGFK optimization described in Section 3.1.3. Summed across all well-separated pairs, the BCCP computations take quadratic work and constant depth. Therefore, our baseline algorithm takes $O(n^2)$ work and $O(\log^2 n)$ depth, and computes the exact HDBSCAN*.

3.2.2 Improved Algorithm We present a more space-efficient algorithm that is faster in practice by using a new definition of well-separation for the WSPD for HDBSCAN*. We denote the maximum and minimum core distances of the points in node A as $\text{cd}_{\max}(A)$ and $\text{cd}_{\min}(A)$, respectively. Consider a pair (A, B) in the WSPD. We define A and B to be ***geometrically-separated*** if $d(A, B) \geq \max\{A_{\text{diam}}, B_{\text{diam}}\}$ and ***mutually-unreachable*** if $\max\{d(A, B), \text{cd}_{\min}(A), \text{cd}_{\min}(B)\} \geq \max\{A_{\text{diam}}, B_{\text{diam}}, \text{cd}_{\max}(A), \text{cd}_{\max}(B)\}$. We consider A and B to be well-separated if they are geometrically-separated, mutually-unreachable, or both. The original definition of well-separation only includes the first condition.

This leads to space savings because in Algorithm 1, recursive calls to procedure FindPair(A, B) on Line 7 will not terminate until A and B are well-separated. Since our new definition is a disjunction between mutual-unreachability and geometric-separation, the calls to FindPair can terminate earlier, leading to fewer pairs generated. When constructing the mutual reachability subgraph to pass to MST, we add only a single edge between the BCCP* (BCCP with respect to mutual reachability distance) of each well-separated pair. With our new definition, the total number of edges generated is upper bounded by the size of the WSPD, which is $O(n)$ [13]. In contrast, Gan and Tao's approach generates $O(n \cdot \text{minPts}^2)$ edges.

Theorem 3.2. *Under the new definition of well-separation, our algorithm computes an MST of the mutual reachability graph.*

Proof. Under our new definition, well-separation is defined as the disjunction between being geometrically-separated and mutually-unreachable. We connect an edge between each well-separated pair (A, B) with the mutual-reachability distance $\max\{d(u^*, v^*), \text{cd}(u^*), \text{cd}(v^*)\}$ as the edge weight, where $u^* \in A$, $v^* \in B$, and (u^*, v^*) is the BCCP* of (A, B). We overload the notation BCCP*(A, B) to also denote the mutual-reachability distance of (u^*, v^*).

Consider the point set P_{root}, which is contained in the root node of the tree associated with its WSPD. Let T be the MST of the full mutual reachability graph G_{MR}. Let T' be the MST of the mutual reachability subgraph G'_{MR}, computed by connecting the BCCP* of each well-separated pair. To ensure that T' produces the correct HDBSCAN* clustering, we prove that it has the same weight as T—in other words, T' is a valid MST of G_{MR}.

We prove the optimality of T' by induction on each tree node P. Since the WSPD is hierarchical, each node P also has a valid WSPD consisting of a subset of pairs of the WSPD of P_{root}. Let (u, v) be an edge in T. There exists an edge $(u', v') \in T'$ that connects the same two components as in T if we were to remove (u, v). We call (u', v') the ***replacement*** of (u, v), which is ***optimal*** if $w(u', v') = w(u, v)$. Let T_P and T'_P be subgraphs of T and T', respectively, containing points in P, but not necessarily spanning P. We inductively hypothesize that all edges of T'_P are optimal. In the base case, a singleton tree node P satisfies the hypothesis by having no edges.

Now consider any node P and edge $(u, v) \in T_P$. The children of P are optimal by our inductive hypothesis. We prove that the edges connecting the children of P are optimal. Points u and v must be from a well-separated pair (A, B), where A and B are children of P in the WSPD hierarchy. Let U and V be a partition of P formed by a cut in T_P that separates point pair (u, v), where $u \in U$ and $v \in V$. We want to prove that the replacement of (u, v) in T'_P is optimal.

We now discuss the first scenario of the proof, shown in Figure 4a, where the replacement edge between U and V is $(u', v') =$ BCCP*$(A, B) = (u^*, v^*)$, and we assume without loss of generality that $u' \in A \cap U$ and $v' \in B \cap V$. Since (u, v) is the closest pair of points connecting U and V by the cut property, then (u', v'), the BCCP* of (A, B), must be optimal; otherwise, (u, v) has smaller

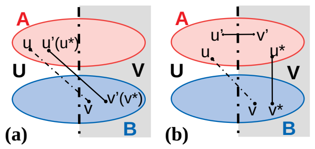

Figure 4: In this figure, we show the two proof cases for HDBSCAN*. We use an oval to represent each node in the WSPD, and solid black dots to represent data points. We represent the partition of the space to U and V using a cut represented by a dotted line.

weight than BCCP$^*(A, B)$, which is a contradiction. This scenario easily generalizes to the case where A and B happen to be completely within U and V, respectively.

We now discuss the second scenario, shown in Figure 4b, where BCCP$^*(A, B) = (u^*, v^*)$ is internal to either U or V. We assume without loss of generality that $u^* \in A \cap V$ and $v^* \in B \cap V$, and that U and V are connected by some intra-node edge (u', v') of A in T'_P. We want to prove that (u', v') is an optimal replacement edge. We consider two cases based on the relationship between A and B under our new definition of well-separation.

Case 1. Nodes A and B are mutually-unreachable, and may or may not be geometrically-separated. The weight of (u', v') is $\max\{d(u', v'), \text{cd}(u'), \text{cd}(v')\} \leq \max\{A_{\text{diam}}, \text{cd}_{\max}(A)\}$. Consider the BCCP* pair (u^*, v^*) between A and B. Based on the fact that A and B are mutually-unreachable, we have

$$\begin{aligned}\text{BCCP}^*(A, B) &= \max\{d(u^*, v^*), \text{cd}(u^*), \text{cd}(v^*)\} \\ &\geq \max\{d(A, B), \text{cd}_{\min}(A), \text{cd}_{\min}(B)\} \\ &\geq \max\{A_{\text{diam}}, B_{\text{diam}}, \text{cd}_{\max}(A), \text{cd}_{\max}(B)\} \\ &\geq \max\{A_{\text{diam}}, \text{cd}_{\max}(A)\},\end{aligned}$$

where the inequality from the second to the third line above comes from the definition of mutual-unreachability. Therefore, $w(u', v')$ is not larger than BCCP$^*(A, B) = w(u^*, v^*)$, and by definition of BCCP*, $w(u^*, v^*)$ is not larger than $w(u, v)$. Hence, $w(u', v')$ is not larger than $w(u, v)$. On the other hand, $w(u', v')$ is not smaller than $w(u, v)$, since otherwise we could form a spanning tree with a smaller weight than T_P, contradicting the fact that it is an MST. Thus, (u', v') is optimal.

Case 2. Nodes A and B are geometrically-separated and not mutually-unreachable. By the definition of BCCP*, we know that $w(u^*, v^*) \leq w(u, v)$, which implies

$$\begin{aligned}\max\{\text{cd}(u^*), \text{cd}(v^*), d(u^*, v^*)\} &\leq \max\{\text{cd}(u), \text{cd}(v), d(u, v)\} \\ \max\{\text{cd}(u^*), \text{cd}(u), d(u, u^*)\} &\leq \max\{\text{cd}(u), \text{cd}(v), d(u, v)\}.\end{aligned}$$

To obtain the second inequality above from the first, we replace $\text{cd}(v^*)$ on the left-hand side with $\text{cd}(u)$, since $\text{cd}(u)$ is also on the right-hand side; we also replace $d(u^*, v^*)$ with $d(u, u^*)$ because of the geometric separation of A and B. Since (u', v') is the lightest BCCP* edge of some well-separated pair in A, $\max\{\text{cd}(u'), \text{cd}(v'), d(u', v')\} \leq \max\{\text{cd}(u), \text{cd}(u^*), d(u, u^*)\}$. We then have

$$\max\{\text{cd}(u'), \text{cd}(v'), d(u', v')\} \leq \max\{\text{cd}(u), \text{cd}(v), d(u, v)\}.$$

This implies that $w(u', v')$ is not larger than $w(u, v)$. Since (u, v) is an edge of MST T_P, the weight of the replacement edge $w(u', v')$ is also not smaller than $w(u, v)$, and hence (u', v') is optimal.

Case 1 and 2 combined prove the optimality of replacement edges in the second scenario. Considering both scenarios, we have shown that each replacement edge in T'_p connecting the children of P is optimal, which proves the inductive hypothesis. Applying the inductive hypothesis to P_{root} completes the proof. □

Our algorithm achieves the following bounds.

Theorem 3.3. *Given a set of n points, we can compute the MST on the mutual reachability graph in $O(n^2)$ work, $O(\log^2 n)$ depth, and $O(n \cdot minPts)$ space.*

Proof. Compared to the cost of GFK for EMST, GFK for HDBSCAN* has the additional cost of computing the core distances, which takes $O(\text{minPts} \cdot n \log n)$ work and $O(\log n)$ depth using k-NN [11]. With our new definition of well-separation, the WSPD computation will only terminate earlier than in the original definition, and so the bounds that we showed for EMST above still hold. The new WSPD definition also gives an $O(n)$ space bound for the well-separated pairs. The space usage of the k-NN computation is $O(n \cdot \text{minPts})$, which dominates the space usage. Overall, this gives $O(n^2)$ work, $O(\log^2 n)$ depth, and $O(n \cdot \text{minPts})$ space. □

Our algorithm gives a clear improvement in space usage over the naive approach of computing an MST from the mutual reachability graph, which takes $O(n^2)$ space, and our parallelization of the exact version of Gan and Tao's algorithm, which takes $O(n \cdot \text{minPts}^2)$ space. We will also see that the smaller memory footprint of this algorithm leads to better performance in practice.

3.2.3 *Implementation*

We implement two algorithms for HDBSCAN*: a parallel exact algorithm based on Gan and Tao [25], and our space-efficient algorithm from Section 3.2.2. Our implementations both use Kruskal's algorithm for MST and use the memory optimization introduced for MemoGFK in Section 3.1.3. For our space-efficient algorithm, we modify the WSPD and MemoGFK algorithm to use our new definition of well-separation.

4 Dendrogram and Reachability Plot

We present a new parallel algorithm for generating a dendrogram and reachability plot, given an unrooted tree with edge weights. Our algorithm can be used for single-linkage clustering [28] by passing the EMST as input, as well as for generating the HDBSCAN* dendrogram and reachability plot (refer to Section 2 for definitions). In addition, our dendrogram algorithm can be used in efficiently generating hierarchical clusters using other linkage criteria (e.g., [42, 55, 57]).

Sequentially, the dendrogram can be generated in a bottom-up (agglomerative) fashion by sorting the edges by weight and processing the edges in increasing order of weight [19, 28, 31, 39, 40]. Initially, all points are assigned their own clusters. Each edge merges the clusters of its two endpoints, if they are in different clusters, using a union-find data structure. The order of the merges forms a tree structure, which is the dendrogram. This takes $O(n \log n)$ work, but has little parallelism since the edges need to be processed one at a time. For HDBSCAN*, we can generate the reachability plot directly from the input tree by running Prim's algorithm on the tree edges starting from an arbitrary vertex [7]. This approach takes $O(n \log n)$ work and is also hard to parallelize efficiently, since Prim's algorithm is inherently sequential.

Our new parallel algorithm uses a top-down approach to generate the dendrogram and reachability plot given a weighted tree. Our algorithm takes $O(n \log n)$ expected work and $O(\log^2 n \log \log n)$ depth with high probability, and hence is work-efficient.

4.1 Ordered Dendrogram

We discuss the relationship between the dendrogram and reachability plot, which are both used in HDBSCAN*. It is known [46] that a reachability plot can be converted into a dendrogram using a linear-work algorithm for Cartesian tree construction [23], which can be parallelized [49]. However, converting in the other direction, which is what we need, is more challenging because the children in dendrogram nodes are unordered, and can correspond to many possible sequences, only one of which corresponds to the traversal order in Prim's algorithm that defines the reachability plot.

Therefore, for a specific starting point s, we define the ***ordered dendrogram*** of s, which is a dendrogram where its in-order traversal corresponds to the reachability plot starting at point s. With this definition, there is a one-to-one correspondence between a ordered dendrogram and a reachability plot, and there are a total of n possible ordered dendrograms and reachability plots for an input of size n. Then, a reachability plot is just the in-order traversal of the leaves of an ordered dendrogram, and an ordered dendrogram is the corresponding Cartesian tree for the reachability plot.

4.2 A Novel Top-Down Algorithm

We introduce a novel work-efficient parallel algorithm to compute a dendrogram, which can be modified to compute an ordered dendrogram and its corresponding reachability plot.

Warm-up. We first propose a simple top-down algorithm for constructing the dendrogram, which does not quite give us the desired work and depth bounds. We first generate an Euler tour on the input tree [34]. Then, we delete the heaviest edge, which can be found in linear work and $O(1)$ depth by checking all edges. By definition, this edge will be the root of the dendrogram, and removing this edge partitions the tree into two subtrees corresponding to the two children of the root. We then convert our original Euler tour into two Euler tours, one for each subtree, which can be done in constant work and depth by updating a few pointers. Next, we partition our list of edges into two lists, one for each subproblem. This can be done by applying list ranking on each Euler tour to determine appropriate offsets for each edge in a new array associated with its subproblem. This step takes linear work and has $O(\log n)$ depth [34]. Finally, we solve the two subproblems recursively.

Although the algorithm is simple, there is no guarantee that the subproblems are of equal size. In the worst case, one of the subproblems could contain all but one edges (e.g., if the tree is a path with edge weights in increasing order), and the algorithm would require $O(n)$ levels of recursion. The total work would then be $O(n^2)$ and depth would be $O(n \log n)$, which is clearly undesirable.

An algorithm with $O(\log n)$ levels of recursion. We now describe a top-down approach that guarantees $O(\log n)$ levels of recursion. We define the ***heavy edges*** of a tree with n edges to be the $n/2$ (or any constant fraction of n) heaviest edges and the ***light edges*** of a tree to be the remaining edges. Rather than using a single edge to partition the tree, we use the $n/2$ heaviest edges to partition the tree. The heavy edges correspond to the part of the dendrogram closer to the root, which we refer to as the *top* part of the dendrogram, and the light edges correspond to subtrees of the top part of the dendrogram. Therefore, we can recursively construct the dendrogram on the heavy edges and the dendrograms on the light edges in parallel. Then, we insert the roots of the dendrograms for the light edges into the leaf nodes of the heavy-edge dendrogram. The base case is when there is a single edge, from which we can trivially generate a dendrogram.

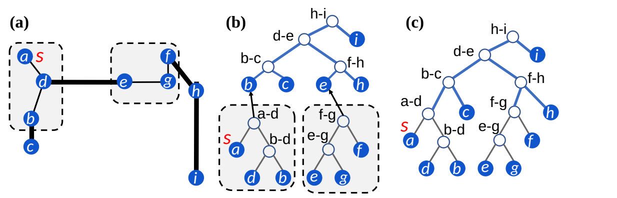

Figure 5: An example of the dendrogram construction algorithm on the tree from Figure 1. The input tree is shown in (a). The 4 heavy edges are in bold. We have three subproblems—one for the heavy edges and two for the light edges. The dendrograms for the subproblems are generated recursively, as shown in (b). The edge labeled on an internal node is the edge whose removal splits a cluster into the two clusters represented by its children. As shown in (c), we insert the roots of the dendrograms for the light edges at the corresponding leaf nodes of the heavy-edge dendrogram. For the ordered dendrogram, the in-order traversal of the leaves corresponds to the reachability plot shown in Figure 1 when the starting point $s = a$.

An example is shown in Figure 5. We first construct the Euler tour of the input tree (Figure 5a). Then, we find the median edge based on edge weight, separate the heavy and light edges and compact them into a heavy-edge subproblem and multiple light-edge subproblems. For the subproblems, we construct their Euler tours by adjusting pointers, and mark the position of each light-edge subproblem in the heavy-edge subproblem where it is detached. Then, recursively and in parallel, we compute the dendrograms for each subproblem (Figure 5b). After that, we insert the roots of the light-edge dendrograms to the appropriate leaf nodes in the heavy-edge dendrogram, as marked earlier (Figure 5c).

Figure 5 shows how this algorithm applies to the input in Figure 1 with source vertex a. The four heaviest edges (b, c), (d, e), (f, h), and (h, i) divide the tree into two light subproblems, consisting of $\{(a, d), (d, b)\}$ and $\{(e, g), (g, f)\}$. The heavy edges form another subproblem. We mark vertices b and e, where the light subproblems are detached. After constructing the dendrogram for the three subproblems, we insert the light dendrograms at leaf nodes b and e, as shown in Figure 5b. It forms the correct dendrogram in Figure 5c.

We now describe the details of the steps to separate the subproblems and re-insert them into the final dendrogram.

Subproblem Finding. To find the position in the heavy-edge dendrogram to insert a light-edge dendrogram at, every light-edge subproblem will be associated with a unique heavy edge. The dendrogram of the light-edge subproblem will eventually connect to the corresponding leaf node in the heavy-edge dendrogram associated with it. We first explain how to separate the heavy-edge subproblem and the light-edge subproblems.

First, we compute the unweighted distance from every point to the starting point s in the tree, and we refer to them as the ***vertex distances***. For the ordered dendrogram, s is the starting point of the reachability plot, whereas s can be an arbitrary vertex if the ordering property is not needed. We compute the vertex distances by performing list ranking on the tree's Euler tour rooted at s. These distances can be computed by labeling each downward edge (away from s) in the tree with a value of 1 and each upward edge (towards s) in the tree with a value of -1, and running list ranking on the edges. The vertex distances are computed only once.

We then identify the light-edge subproblems in parallel by using the vertex distances. For each light edge (u, v), we find an adjacent edge (w, u) such that w has smaller vertex distance than both u and v. We call (w, u) the ***predecessor edge*** of (u, v). Each edge can only have one predecessor edge (an edge adjacent to s will choose itself as the predecessor). In a light-edge subproblem not containing the starting vertex s, the predecessor of each light edge will either be a light edge in the same light-edge subproblem, or a heavy edge. The edges in each light-edge subproblem will form a subtree based on the pointers to predecessor edges. We can obtain the Euler tour of each light-edge subproblem by adjusting pointers of the original Euler tour. The next step is to run list ranking to propagate a unique label (the root's label of the subproblem subtree) of each light-edge subproblem to all edges in the same subproblem. To create the Euler tour for the heavy subproblem, we contract the subtrees for the light-edge subproblems: for each light-edge subproblem, we map its leaves to its root using a parallel hash table. Now each heavy edge adjacent to a light-edge subproblem leaf can connect to the heavy edge adjacent to the light-edge subproblem root by looking it up in the hash table. The Euler tour for the heavy-edge subproblem can now be constructed by adjusting pointers. We assign the label of the heavy-edge subproblem root to all of the heavy edges in parallel. Then, we semisort the labeled edges to group edges of the same light-edge subproblems and the heavy-edge subproblem. Finally, we recursively compute the dendrograms on the light-edge subproblems and the heavy-edge subproblem. In the end, we connect the light-edge dendrogram for each subproblem to the heavy-edge dendrogram leaf node corresponding to the shared endpoint between the light-edge subproblem and its unique heavy predecessor edge. For the light-edge subproblem containing the starting point s, we simply insert its light-edge dendrogram into the left-most leaf node of the heavy-edge dendrogram.

Consider Figure 5a. The heavy-edge subproblem contains edges $\{(b, c), (d, e), (f, h), (h, i)\}$, and its dendrogram is shown in Figure 5b. For the light-edge subproblem $\{(e, g), (g, f)\}$, (e, g) has heavy predecessor edge (d, e), and (g, f) has light predecessor edge (e, g). The unique heavy edge associated with the light-edge subproblem is hence (d, e), with which it shares vertex e. Hence, we insert the light-edge dendrogram for the subproblem into leaf node e in the heavy-edge dendrogram, as shown in Figure 5b. The light-edge subproblem containing $\{(a, d), (d, b)\}$ contains the starting point $s = a$, and so we insert its dendrogram into the leftmost leaf node b of the heavy-edge dendrogram, as shown in Figure 5b.

We first show that our algorithm correctly computes a dendrogram, and analyze its cost bounds (Theorem 4.1). Then, we describe and analyze additional steps needed to generate an ordered dendrogram and obtain a reachability plot from it (Theorem 4.2).

Theorem 4.1. *Given a weighted spanning tree with n vertices, we can compute a dendrogram in $O(n \log n)$ expected work and $O(\log^2 n \log\log n)$ depth with high probability.*

Proof. We first prove that our algorithm correctly produces a dendrogram. In the base case, we have one edge (u, v), and the algorithm produces a tree with a root representing (u, v), and with u and v as children of the root, which is trivially a dendrogram. We now inductively hypothesize that recursive calls to our algorithm correctly produce dendrograms. The heavy subproblem recursively computes a top dendrogram consisting of all of the heavy edges, and the light subproblems form dendrograms consisting of light edges. We replace the leaf vertices in the top dendrogram associated with light subproblems by the roots of the dendrograms on light edges. Since the edges in the heavy subproblem are heavier than all edges in light subproblems, and are also ancestors of the light edges in the resulting tree, this gives a valid dendrogram.

We now analyze the cost of the algorithm. To generate the Euler tour at the beginning, we first sort the edges and create an adjacency list representation, which takes $O(n \log n)$ work and $O(\log n)$ depth [17]. Next, we root the tree, which can be done by list ranking on the Euler tour of the tree. Then, we compute the vertex distances to s using another round of list ranking based on the rooted tree.

There are $O(\log n)$ recursive levels since the subproblem sizes are at most half of the original problem. We now show that each recursive level takes linear expected work and polylogarithmic depth with high probability. Note that we cannot afford to sort the edges on every recursive level, since that would take $O(n \log n)$ work per level. However, we only need to know which edges are heavy and which are light, and so we can use parallel selection [34] to find the median and partition the edges into two sets. This takes $O(n)$ work and $O(\log n \log\log n)$ depth. Identifying predecessor edges takes a total of $O(n)$ work and $O(1)$ depth: we find and record for each vertex its edge where the other endpoint has a smaller vertex distance than it (using WriteMin); then, the predecessor of each edge is found by checking the recorded edge for its endpoint with smaller vertex distance. We then use list ranking to assign labels to each subproblem, which takes $O(n)$ work and $O(\log n)$ depth [34]. The hash table operations to contract and look up the light-edge subproblems cost $O(n)$ work and $O(\log n)$ depth with high probability. The semisort to group the subproblems takes $O(n)$ expected work and $O(\log n)$ depth with high probability. Attaching the light-edge dendrograms to the heavy-edge dendrogram takes $O(n)$ work and $O(1)$ depth across all subproblems. Multiplying the bounds by the number of levels of recursion proves the theorem. □

Theorem 4.2. *Given a starting vertex s, we can generate an ordered dendrogram and reachability plot in the same cost bounds as in Theorem 4.1.*

Proof. We have computed the vertex distances of all vertices from s. When generating the ordered dendrogram and constructing each internal node of the dendrogram corresponding to an edge (u, v), and without loss of generality let u have a smaller vertex distance than v, our algorithm puts the result of the subproblem attached to u in the left subtree, and that of v in the right subtree. This additional comparison does not increase the work and depth of our algorithm.

Our algorithm recursively builds ordered dendrograms on the heavy-edge subproblem and on each of the light-edge subproblems, which we assume to be correct by induction. The base case is a single edge (u, v), and without loss of generality let u have a smaller vertex distance than v. Then, the dendrogram will contain a root node representing edge (u, v), with u as its left child and v as its right child. Prim's algorithm would visit u before v, and so is the in-order traversal of the dendrogram, so it is an ordered dendrogram.

We now argue that the way that light-edge dendrograms are attached to the leaves of the heavy-edge dendrogram correctly produces an ordered dendrogram. First, consider a light-edge subproblem that contains the source vertex s. In this case, its dendrogram is attached as the leftmost leaf of the heavy-edge dendrogram, and will be the first to be traversed in the in-order traversal. The vertices in the light-edge subproblem form a connected component A. They will be traversed before any other vertices in Prim's algorithm because all incident edges that leave A are heavy edges, and thus are heavier than any edge in A. Therefore, vertices outside of A can only be visited after all vertices in A have been visited, which correctly corresponds to the in-order traversal.

Next, we consider the case where the light-edge subproblem does not contain s. Let (u, v) be the predecessor edge of the light-edge subproblem, and let A be the component containing the edges in the light-edge subproblem (v is a vertex in A). Now, consider a different light-edge subproblem that does not contain s, whose predecessor edge is (x, y), and let B be the component containing the edges in this subproblem (y is a vertex in B). By construction, we know that A is in the right subtree of the dendrogram node corresponding to edge (u, v) and B is in the right subtree of node corresponding to (x, y). The ordering between A and B is correct as long as they are on different sides of either node (u, v) or node (x, y). For example, if B is in the left subtree of node (u, v), then its vertices appear before A in the in-order traversal of the dendrogram. By the inductive hypothesis on the heavy-edge subproblem, in Prim's order, B will be traversed before (u, v), and (u, v) is traversed before A. We can apply a similar argument to all other cases where A and B are on different sides of either node (u, v) or node (x, y).

We are concerned with the case where A and B are both in the right subtrees of the nodes representing their predecessor edges. We prove by contradiction that this cannot happen. Without loss of generality, suppose node (x, y) is in the right subtree of node (u, v), and let both A and B be in the right subtree of (x, y). There exists a lowest common ancestor (LCA) node (x', y') of A and B. (x', y') must be a heavy edge in the right subtree of (x, y). By properties of the LCA, A and B are in different subtrees of node (x', y'). Without loss of generality, let A be in the left subtree. Now consider edge (x', y') in the tree. By the inductive hypothesis on the heavy-edge dendrogram, in Prim's traversal order, we must first visit the leaf that A attaches to (and hence A) before visiting (x', y'), which must be visited before the leaf that B attaches to (and hence B). On the other hand, edge (x, y) is also along the same path since it is the predecessor of B. Thus, we must either have (x', y') in (x, y)'s left subtree or (x, y) in (x', y')'s right subtree, which is a contradiction to (x', y') being in the right subtree of (x, y).

We have shown that given any two light-edge subproblems, their relative ordering after being attached to the heavy-edge dendrogram is correct. Since the heavy-edge dendrogram is an ordered dendrogram by induction, the order in which the light-edge subproblems are traversed is correct. Furthermore, each light-edge subproblem generates an ordered dendrogram by induction. Therefore, the overall dendrogram is an ordered dendrogram.

Once the ordered dendrogram is computed, we use list ranking to perform an in-order traversal on the Euler tour of the dendrogram to give each node a rank, and write them out in order. We then filter out the non-leaf nodes to to obtain the reachability plot. Both list ranking and filtering take $O(n)$ work and $O(\log n)$ depth. $\square$

Implementation. In our implementation, we simplify the process of finding the subproblems by using a sequential procedure rather than performing parallel list ranking, because in most cases parallelizing over the different subproblems already provides sufficient parallelism. We set the number of heavy edges to $n/10$, which we found to give better performance in practice, and also preserves the theoretical bounds. We switch to the sequential dendrogram construction algorithm when the problem size falls below $n/2$.

5 Experiments

Environment. We perform experiments on an Amazon EC2 instance with $2 \times$ Intel Xeon Platinum 8275CL (3.00GHz) CPUs for a total of 48 cores with two-way hyper-threading, and 192 GB of RAM. By default, we use all cores with hyper-threading. We use g++ compiler (version 7.4) with -O3 flag, and use Cilk for parallelism [36]. We do not report times for tests that exceed 3 hours.

We test the following implementations for EMST (note that the EMST problem does not include dendrogram generation):

- ***EMST-Naive***: The method of creating a graph with the BCCP edges from all well-separated pairs and then running MST on it.
- ***EMST-GFK***: The parallel GeoFilterKruskal algorithm described in Section 3.1.2 (Algorithm 2).
- ***EMST-MemoGFK***: The parallel GeoFilterKruskal algorithm with the memory optimization described in Section 3.1.3 (Algorithm 3).

We test the following implementations for HDBSCAN*:

- ***HDBSCAN*-GanTao***: The modified algorithm of Gan and Tao for exact HDBSCAN* described in Section 3.2.1.
- ***HDBSCAN*-MemoGFK***: The HDBSCAN* algorithm using our new definition of well-separation described in Section 3.2.2.

Both ***HDBSCAN*-GanTao*** and ***HDBSCAN*-MemoGFK*** use the memory optimization described in Section 3.1.3. All HDBSCAN* running times include constructing an MST of the mutual reachability graph and computing the ordered dendrogram. We use a default value of minPts = 10 (unless specified otherwise), which is also adopted in previous work [14, 25, 39].

Our algorithms are designed for multicores, as we found that multicores are able to process the largest data sets in the literature for these problems (machines with several terabytes of RAM can be rented at reasonable costs on the cloud). Our multicore implementations achieve significant speedups over existing implementations in both the multicore and distributed memory contexts.

Data Sets. We use the synthetic seed spreader data sets produced by the generator in [24]. It produces points generated by a random walk in a local neighborhood (***SS-varden***). We also use ***UniformFill*** that contains points distributed uniformly at random inside a bounding hypergrid with side length $\sqrt{n}$ where n is the total number of points. We generated the synthetic data sets with 10 million points (unless specified otherwise) for dimensions $d = 2, 3, 5, 7$.

We use the following real-world data sets. ***GeoLife*** [2, 58] is a 3-dimensional data set with $24,876,978$ data points. This data set contains user location data, and is extremely skewed. ***Household*** [3, 5] is a 7-dimensional data set with $2,049,280$ points representing electricity consumption measurements in households. ***HT*** [4, 33] is a 10-dimensional data set with $928,991$ data points containing home sensor data. ***CHEM*** [1, 21] is a 16-dimensional data set with $4,208,261$ data points containing chemical sensor data. All of the data sets fit in the RAM of our machine.

	Speedup over Best Sequential		Self-relative Speedup	
Method	Range	Average	Range	Average
EMST-Naive	3.51-10.69x	6.90x	16.79-33.47x	24.15x
EMST-GFK	1.52-7.01x	3.60x	8.11-11.51x	9.08x
EMST-MemoGFK	14.61-55.89x	31.31x	14.61-55.89x	31.31x
HDBSCAN*-MemoGFK	11.13-46.69x	26.29x	11.13-46.69x	26.29x
HDBSCAN*-GanTao	4.29-35.14x	13.76x	8.23-40.32x	20.97x

Table 2: Speedup over the best sequential algorithm as well as the self-relative speedup on 48 cores.

Comparison with Previous Implementations. For EMST, we tested the sequential Dual-Tree Boruvka algorithm of March et al. [38] (part of `mlpack`), and our single-threaded EMST-MemoGFK times are 0.89–4.17 (2.44 on average) times faster. We also tested McInnes and Healy's sequential HDBSCAN* implementation which is based on Dual-Tree Boruvka [39]. We were unable to run their code on our data sets with 10 million points in a reasonable amount of time. On a smaller data set with 1 million points (2D-SS-varden-1M), their code takes around 90 seconds to compute the MST and dendrogram, which is 10 times slower than our HDBSCAN*-MemoGFK implementation on a single thread, due to their code using Python and having fewer optimizations. We observed a similar trend on other data sets for McInnes and Healy's implementation.

The GFK algorithm implementation for EMST of [15] in the `Stann` library supports multicore execution using OpenMP. We found that, in parallel, their GFK implementation always runs much slower when using all 48 cores than running sequentially, and so we do not include their parallel running times in our experiments. In addition, our own sequential implementation of the same algorithm is 0.79–2.43x (1.23x on average) faster than theirs, and so we parallelize our own version as a baseline. We also tested the multicore implementation of the parallel OPTICS algorithm in [45] using all 48 cores on our machine. Their code exceeded our 3-hour time limit for our data sets with 10 million points. On a smaller data set of 1 million points (2D-SS-varden-1M), their code took 7988.52 seconds, whereas our fastest parallel implementations take only a few seconds. We also compared with the parallel HDBSCAN* code by Santos et al. [47], which mainly focuses on approximate HDBSCAN* in distributed memory. As reported in their paper, for the HT data set with minPts = 30, their code on 60 cores takes 42.54 and 31450.89 minutes to build the approximate and exact MST, respectively, and 124.82 minutes to build the dendrogram. In contrast, our fastest implementation using 48 cores builds the MST in under 3 seconds, and the dendrogram in under a second.

Overall, we found the fastest sequential methods for EMST and HDBSCAN* to be our EMST-MemoGFK and HDBSCAN*-MemoGFK methods running on 1 thread. Therefore, we also based our parallel implementations on these methods.

Performance of Our Implementations. Table 2 shows the self-relative speedups and speedups over the fastest sequential time of our parallel implementations on 48 cores. Figures 6 and 7 show the parallel speedup as a function of thread count for our implementations of EMST and HDBSCAN* with minPts = 10, respectively, against the fastest sequential times. For most data sets, we see additional speedups from using hyper-threading compared to just using a single thread per core. A decomposition of parallel timings for our implementations on two data sets is presented in Figure 8.

EMST Results. In Figure 6, we see that our fastest EMST implementations (EMST-MemoGFK) achieve good speedups over the best sequential times, ranging from 14.61–55.89x on 48 cores with hyper-threading. On the lower end, 10D-HT-0.93M has a speedup of 14.61x (Figure 6k), because for a small data set, the total work done is small and the parallelization overhead becomes prominent.

EMST-MemoGFK significantly outperforms EMST-GFK and EMST-Naive by up to 17.69x and 8.63x, respectively, due to its memory optimization, which reduces memory traffic. We note that EMST-GFK does not get good speedup, and is slower than EMST-Naive in all subplots of Figure 6. This is because the WSPD input to EMST-GFK (S in Algorithm 2) needs to store references to the well-separated pair as well as the BCCP points and distances, whereas EMST-Naive only needs to store the BCCP points and distances. This leads to increased memory traffic for EMST-GFK for operations on S and its subarrays, which outweighs its advantage of computing fewer BCCPs. This is evident from Figure 8, which shows that EMST-GFK spends more time in WSPD, but less time in Kruskal compared to EMST-Naive. EMST-MemoGFK spends the least amount of time in WSPD due to its pruning optimizations, while spending a similar amount of time in Kruskal as EMST-GFK.

HDBSCAN* Results. In Figure 7, we see that our HDBSCAN*-MemoGFK method achieves good speedups over the best sequential times, ranging from 11.13–46.69x on 48 cores. Similar to EMST, we observe a similar lower speedup for 10D-HT-0.93M due to its small size, and observe higher speedups for larger data sets. The dendrogram construction takes at least 50% of the total time for Figures 7a, b, and e–h, and hence has a large impact on the overall scalability. We discuss the dendrogram scalability separately.

We find that HDBSCAN*-MemoGFK consistently outperforms HDBSCAN*-GanTao due to having a fewer number of well-separated pairs (2.5–10.29x fewer) using the new definition of well-separation. This is also evident in Figure 8, where we see that HDBSCAN*-MemoGFK spends much less time than HDBSCAN*-GanTao in WSPD computation.

We tried varying minPts over a range from 10 to 50 for our HDBSCAN* implementations and found just a moderate increase in the running time for increasing minPts.

MemoGFK Memory Usage. Overall, the MemoGFK method for both EMST and HDBSCAN* reduces memory usage by up to 10x compared to materializing all WSPD pairs.

Dendrogram Results. We separately report the performance of our parallel dendrogram algorithm in Figure 9, which shows the speedups and running times on all of our data sets. We see that the parallel speedup ranges from 5.69–49.74x (with an average of 17.93x) for the HDBSCAN* MST with minPts =10, and 5.35–52.58x (with an average 20.64x) for single-linkage clustering, which is solved by generating a dendrogram on the EMST. Dendrogram construction for single-linkage clustering shows higher scalability because the heavy edges are more uniformly distributed in space, which creates a larger number of light-edge subproblems and increases parallelism. In contrast, for HDBSCAN*, which has a higher value of minPts, the sparse regions in the space tend to have clusters of edges with large weights even if some of them have small Euclidean distances, since these edges have high mutual reachability distances. Therefore, these heavy edges are less likely to divide up the edges into a uniform distribution of subproblems in the space, leading to lower parallelism. On the other hand, we observe that

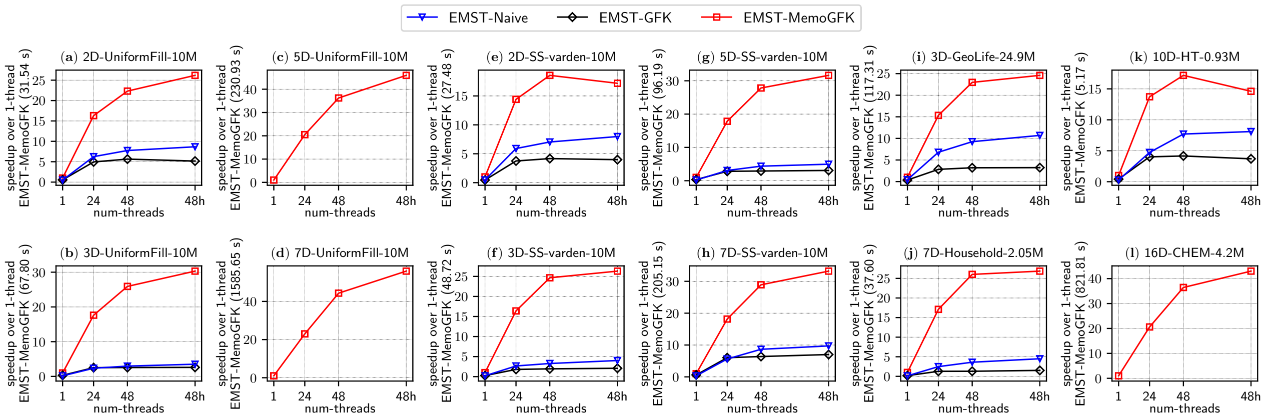
Figure 6: Speedup of EMST implementations over the *best* serial baselines vs. thread count. The best serial baseline and its running time for each data set is shown on the y-axis label. “48h” on the x-axis refers to 48 cores with hyper-threading.

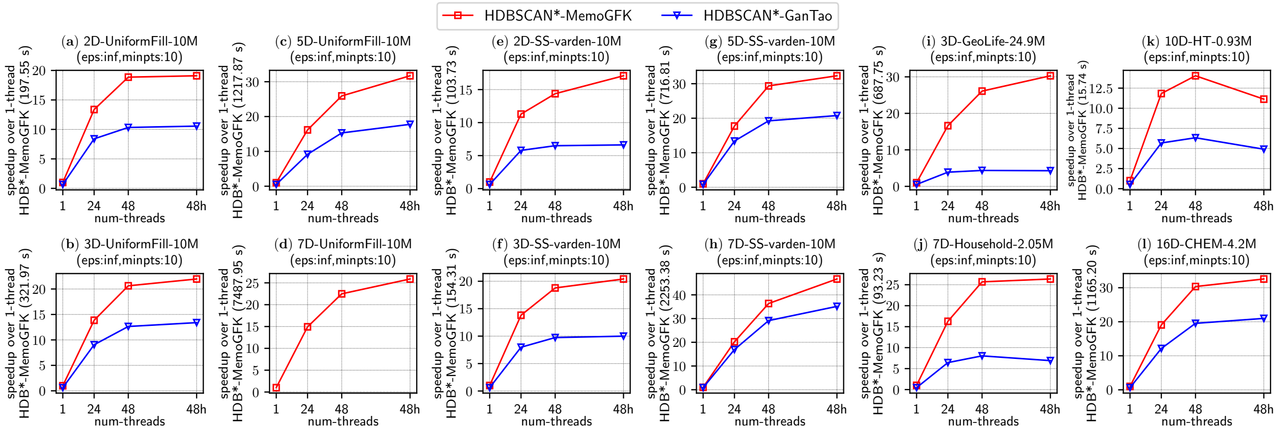
Figure 7: Speedup of HDBSCAN* implementations (minPts $= 10$) over the *best* serial baselines vs. thread count. The best serial baseline and its running time for each data set is shown on the y-axis label. “48h” on the x-axis refers to 48 cores with hyper-threading.

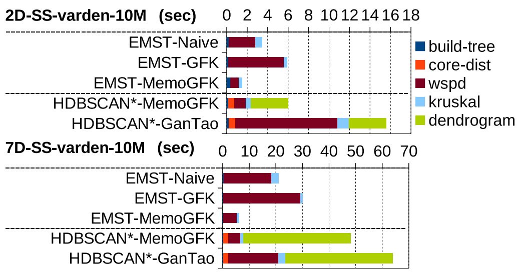
Figure 8: Running time decomposition for EMST and HDBSCAN* (with minPts $= 10$) on two data sets using 48 cores with hyper-threading. "dendrogram" refers to computing the ordered dendrogram; “kruskal” refers to Kruskal’s MST algorithm; "wspd" refers to computing the WSPD, or the sum of WSPD tree traversal times across rounds; "core-dist" refers to computing core distances of all points; and "build-tree" refers to building a k-tree on all points.

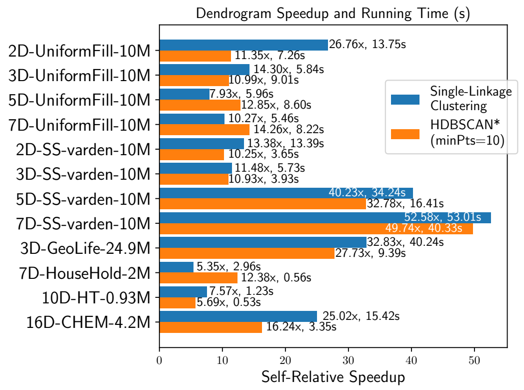

Figure 9: Self-relative speedups and times for ordered dendrogram computation for single-linkage clustering and HDBSCAN* (minPts $= 10$). The x-axis indicates the self-relative speedup on 48 cores with hyper-threading. The speedup and time is shown at the end of each bar.

across all data sets, the dendrogram for single-linkage clustering takes an average of 16.44 seconds, whereas the dendrogram for HDBSCAN* takes an average of 9.27 seconds. This is because the single-linkage clustering generates more light-edge subproblems and hence requires more work. While it is possible to tune the fraction of heavy edges for different values of minPts, we found that using $n/10$ heavy edges works reasonably well in all cases.

Acknowledgements. We thank Pilar Cano for helpful discussions. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] [n.d.]. CHEM Dataset. https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures.

[2] [n.d.]. GeoLife Dataset. https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/.

[3] [n.d.]. Household Dataset. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption.

[4] [n.d.]. HT Dataset. https://archive.ics.uci.edu/ml/datasets/Gas+sensors+for+home+activity+monitoring.

[5] [n.d.]. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml.

[6] Pankaj K. Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf, and Emo Welzl. 1991. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry* (1991), 407–422.

[7] Mihael Ankerst, Markus Breunig, H. Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *ACM SIGMOD International Conference on Management of Data*. 49–60.

[8] Sunil Arya and David M. Mount. 2016. A Fast and Simple Algorithm for Computing Approximate Euclidean Minimum Spanning Trees. In *ACM-SIAM Symposium on Discrete Algorithms*. 1220–1233.

[9] Bentley and Friedman. 1978. Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces. *IEEE Trans. Comput.* C-27, 2 (Feb 1978), 97–105.

[10] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.

[11] Paul B Callahan. 1993. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 332–340.

[12] Paul B. Callahan and S. Rao Kosaraju. 1993. Faster Algorithms for Some Geometric Graph Problems in Higher Dimensions. In *ACM-SIAM Symposium on Discrete Algorithms*. 291–300.

[13] Paul B. Callahan and S. Rao Kosaraju. 1995. A Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields. *J. ACM* 42, 1 (1995), 67–90.

[14] Ricardo 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)*, Article 5 (2015), 5:1–5:51 pages.

[15] Samidh Chatterjee, Michael Connor, and Piyush Kumar. 2010. Geometric Minimum Spanning Trees with GeoFilterKruskal. In *International Symposium on Experimental Algorithms (SEA)*, Vol. 6049. 486–500.

[16] Danny Z. Chen, Michiel Smid, and Bin Xu. 2005. Geometric Algorithms for Density-Based Data Clustering. *International Journal of Computational Geometry & Applications* 15, 03 (2005), 239–260.

[17] Richard Cole. 1988. Parallel Merge Sort. *SIAM J. Comput.* 17, 4 (Aug. 1988), 770–785.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press.

[19] Mark de Berg, Ade Gunawan, and Marcel Roeloffzen. 2019. Faster DB-scan and HDB-scan in Low-Dimensional Euclidean Spaces, In International Symposium on Algorithms and Computation (ISAAC). *International Journal of Computational Geometry & Applications* 29, 01, 21–47.

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 226–231.

[21] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[22] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1976. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software* 3, 3 (7 1976), 209–226.

[23] Harold N. Gabow, Jon L. Bentley, and Robert E. Tarjan. 1984. Scaling and related techniques for geometry problems. In *ACM Symposium on Theory of Computing (STOC)*. 135–143.

[24] Junhao Gan and Yufei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 14:1–14:45.

[25] Junhao Gan and Yufei Tao. 2018. Fast Euclidean OPTICS with Bounded Precision in Low Dimensional Space. In *ACM SIGMOD International Conference on Management of Data*. 1067–1082.

[26] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 698–710.

[27] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In *MLHPC*. Article 2, 2:1–2:10 pages.

[28] John C. Gower and Gavin J. S. Ross. 1969. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 18, 1 (1969), 54–64.

[29] 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.

[30] Ade Gunawan. 2013. A faster algorithm for DBSCAN. Master's thesis, Eindhoven University of Technology.

[31] William Hendrix, Md Mostofa Ali Patwary, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2012. Parallel hierarchical clustering on shared memory platforms. In *International Conference on High Performance Computing*. 1–9.

[32] Xu Hu, Jun Huang, and Minghui Qiu. 2017. A Communication Efficient Parallel DBSCAN Algorithm Based on Parameter Server. In *ACM Conference on Information and Knowledge Management (CIKM)*. 2107–2110.

[33] Ramón Huerta, Thiago Schiavo Mosqueiro, Jordi Fonollosa, Nikolai F. Rulkov, and Irene Rodríguez-Luján. 2016. Online Humidity and Temperature Decorrelation of Chemical Sensors for Continuous Monitoring. *Chemometrics and Intelligent Laboratory Systems* 157, 169–176.

[34] Joseph Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[35] Richard M. Karp and Vijaya Ramachandran. 1990. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 869–941.

[36] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *J. Supercomputing* 51, 3 (2010). Springer.

[37] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. 2016. NG-DBSCAN: Scalable Density-based Clustering for Arbitrary Data. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 157–168.

[38] William B March, Parikshit Ram, and Alexander G Gray. 2010. Fast Euclidean minimum spanning tree: algorithm, analysis, and applications. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 603–612.

[39] Leland McInnes and John Healy. 2017. Accelerated hierarchical density clustering. *arXiv preprint arXiv:1705.07321* (2017).

[40] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. arXiv:1109.2378 [stat.ML]

[41] Giri Narasimhan and Martin Zachariasen. 2001. Geometric Minimum Spanning Trees via Well-Separated Pair Decompositions. *ACM Journal of Experimental Algorithmics* 6 (2001), 6.

[42] Clark F. Olson. 1995. Parallel algorithms for hierarchical clustering. *Parallel Comput.* 21, 8 (1995), 1313 – 1325.

[43] Vitaly Osipov, Peter Sanders, and Johannes Singler. 2009. The Filter-Kruskal Minimum Spanning Tree Algorithm. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 52–61.

[44] M. Patwary, D. Palsetia, A. Agrawal, W. K. Liao, F. Manne, and A. Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[45] M. Patwary, D. Palsetia, A. Agrawal, W. K. Liao, F. Manne, and A. Choudhary. 2013. Scalable parallel OPTICS data clustering using graph algorithmic techniques. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[46] Jörg Sander, Xuejie Qin, Zhiyong Lu, Nan Niu, and Alex Kovarsky. 2003. Automatic extraction of clusters from hierarchical clustering representations. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. 75–87.

[47] J. Santos, T. Syed, M. Coelho Naldi, R. J. G. B. Campello, and J. Sander. 2019. Hierarchical Density-Based Clustering using MapReduce. *IEEE Transactions on Big Data* (2019), 1–1.

[48] Michael Ian Shamos and Hoey Dan. 1975. Closest-point problems. (1975), 151–162.

[49] J. Shun and G. E. Blelloch. 2014. A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction. *ACM Transactions on Parallel Computing (TOPC)* 1, 1, Article 8 (Oct. 2014), 8:1–8:20 pages.

[50] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention Through Priority Updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 152–163.

[51] Hwanjun Song and J. Lee. 2018. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *ACM SIGMOD International Conference on Management of Data*. 1173–1187.

[52] Vijay V. Vazirani. 2010. *Approximation Algorithms*. Springer Publishing Company, Incorporated.

[53] P. J. Wan, G. Călinescu, X. Y. Li, and O. Frieder. 2002. Minimum-Energy Broadcasting in Static Ad Hoc Wireless Networks. *Wireless Networks* 8, 6 (2002), 607–617.

[54] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-efficient and practical parallel DBSCAN. In *ACM SIGMOD International Conference on Management of Data*. 2555–2571.

[55] Ying Xu, Victor Olman, and Dong Xu. 2001. Minimum Spanning Trees for Gene Expression Data Clustering. *Genome Informatics* 12 (02 2001), 24–33.

[56] Andrew Chi-Chih. Yao. 1982. On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems. *SIAM J. Comput.* 11, 4 (1982), 721–736.

[57] Meichen Yu, Arjan Hillebrand, Prejaas Tewarie, Jil Meier, Bob van Dijk, Piet Van Mieghem, and Cornelis Jan Stam. 2015. Hierarchical clustering in minimum spanning trees. *Chaos: An Interdisciplinary Journal of Nonlinear Science* 25, 2 (2015), 023107.

[58] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning Transportation Mode from Raw GPS Data for Geographic Applications on the Web. In *International Conference on World Wide Web*. 247–256.