point $q \in N'(p)$, $\Pr[q \in N(p)] \leq 1/2$. If $s > c \log n$, then

$$\Pr\left[\left|N'(p)\right| = |N(p)|\right] \leq \frac{1}{2^s} = \frac{1}{2^{c \log n}} = n^{-c}$$

which indicates that the number of p's neighbors in G is $O(\log n)$ *whp*. The expected neighbor size is $|N'(p)| \cdot \Pr[|N'(p)| = |N(p)|] = O(1)$. Note that the results holds for all levels for the point p, if it is redistributed and processed in multiple levels (on Line 25). □

Corollary 4.8. *The size of the set $\Pi_{P_i} \cap L_k$ being check on Line 13 is $O(c^5)$ in expectation and $O(c^5 \log n)$* whp.

We can simply multiply the bounds in Lem. 3.1 with $\kappa = 1$ and Lem. 4.7 and get Col. 4.8. To efficiently acquire the neighbor set of a vertex (i.e., $\Pi_{P_i} \cap L_k$ on Line 13), we can generate $\Pi_{P_i} \cap L_k$ once Π_{P_i} is generated on Line 7. We note that L_k can change when new tree nodes are generated, but we can maintain it lazily: every time when we loop over the points in L_k, we skip those that are removed. We note that the work is only $O(m\mathcal{H}(T))$ to check all points at S in every level for a batch of m inserted points, which is asymptotically bounded by other parts in this algorithm.

We now analyze the work and span bounds for the parallel batch-insertion algorithm.

Theorem 4.9. *The batch insertion algorithm (Alg. 4) can correctly insert a set of points S into a cover tree T using $O(c^5 m\mathcal{H}(T))$ expected work and $O(\mathcal{H}(T) \log m(\log c + \log m \log\log n))$ span* whp, *where $m = |S|$, $n = |T|$, and $m \leq n$.*

Proof. The correctness of this algorithm is already shown in Lem. 4.6.

First of all, Thm. 2.1 shows that after a constant number of samplings, the expansion rate of the metric (X, D_X) only changed by at most a constant fraction, which will be hidden by the asymptotic notation. In all of our analyses, we apply union bound (Boole's inequality) on high probability bounds, which means our analysis only requires sampling for one round.

For the traversal on Line 7, the work and span for each query is given in Sec. 3.2, multiplied that by m gives the total work for all points. The output size is no more than the work, so semisorting them takes $O(c^5 m\mathcal{H}(T))$ expected work and $O(\log c + \log m + \log\log n)$ span *whp*. Then for the cover tree construction, based on Col. 4.8, building the graph G requires $O(c^5 m)$ expected work for all levels, and computing the MIS on G uses $O(m)$ expected work. Also, according to Lem. 4.7 since the maximum degree for each node is $O(\log n)$, computing the MIS at each level has $O(\log m \log\log n)$ span *whp* [58], and the total span for all levels is $O(\mathcal{H}(T)(\log c + \log m \log\log n))$ span *whp*. Adding the new tree nodes has the same work and span bounds as the step to generate MIS.

Finally, let's analyze the work and span to construct the Π sets for new tree nodes (the parallel-for loop on Line 18). While there can be many points in Π_{P_i}, we note that if we look at a specific point q, if we first insert $S \setminus \{q\}$ and then insert q, we will run exactly the same checks, but in the traversal part (Line 7). We know the traversal work is $O(c^5\mathcal{H}(T))$ per node in the batch, which also bounds the work for constructing the conflict sets here. To parallelize this step, we can generate all the pairs and semisort them, which is work-efficient in expectation and has span asymptotically bounded by the MIS steps.

Hence, the work of the batch insertion algorithm is bounded by the traversal step, and span is bounded by the MIS step. In addition, we have the prefix-doubling step that partitions the batch into $\log |S|$ sub-batches. Prefix-doubling does not cause additional work, but will increase the span by a factor of $O(\log m)$. Combining them together gives the stated bounds in the theorem. □

When assuming constant expansion rate ($c = O(1)$) and bounded aspect ratio ($\mathcal{H}(T) = \Theta(\log n)$), Alg. 4 has $O(m \log n)$ expected work and $O(\log n \log^2 m \log\log n)$ span *whp*. Constructing cover trees can also be parallelized using the same algorithm and analysis.

Theorem 4.10. *Constructing a cover tree that contains n points takes $O(c^5 n\mathcal{H}(T))$ expected work and the span of $O(\mathcal{H}(T) \log n(\log c + \log n \log\log n))$* whp.

With the same assumptions, the work is $O(n \log n)$ in expectation, and the span is $O(\log^3 n \log\log n)$ *whp*.

4.2 The Batch-Deletion Algorithm

We now discuss the parallel batch-deletion algorithm. It is interesting that, unlike many other data structures, batch-deletion is easier than batch-insertion—the hardest part in the nearest search structure is to locate the updated points. For insertion, if too many points are added, their proximity information and nearest neighbors cannot be directly given since they can be in the inserted points. However, for deletion, the original cover tree can provide sufficient proximity information for the points either in the batch or not, since they are all in the cover tree before batch-deletion. Hence, for deletion, we do not need prefix doubling, and can finish the entire batch-deletion in one round.

The key observation for batch-deletion is that, for each undeleted point, if its directed parent is also not deleted, then this local structure still satisfies the cover tree properties and can remain unchanged. For each deleted point p, if p is a leaf, it can be directly removed; otherwise, it may uncover p's direct child, who needs to be either redistributed to another undeleted point in the same level, or promoted to the current level. Meanwhile, multiple points can be promoted to the same level. Similar to the batch-insertion algorithm, we need to run an MIS for all uncovered points at one level, and then decide those that get promoted and the others that will then be covered.

Based on these key insights, our parallel batch-deletion algorithm is shown in Alg. 5. The first step is similar to that in the insertion algorithm—we first run the traverse algorithm to track all tree nodes that cover each point with twice the covering distances, and the tree nodes in all levels each point in the tree. We then semisort the output key-value pairs to transpose the keys and values, and these precomputed results will be used later in the algorithm.

Then we start to process each level, delete the nodes in the given batch while maintaining the cover tree properties. This is from Line 4. We denote the uncovered points at each level using the set X, and initially, when we start to process the leaf level, X is empty. For each level, we first delete the tree nodes corresponding to the nodes in the delete batch, which may uncover some nodes

Algorithm 5: BatchDelete(T, S).

Input: A cover tree T and a set of node S.
Output: The new cover tree T' that excludes all nodes in S.

1 **parallel foreach** $p_i \in S$ **do**
2 Run Traverse(T, p_i) that tracks all nodes $q_j \in T$ with $d(p_i, q_j) < 2^{k+1}$ where q_j is at the k-th level, and record tree nodes $\bar{q}_{j'}$ for all levels p_i is in T
3 Semisort pairs $(\bar{q}_j, p_i)$ and let $L_k = \{\bar{q}_k \mid (\bar{q}_k, p_i)$ *exists*$\}$
4 $X = \emptyset$
5 **for** k from the leaf level to root level **do**
6 Remove all tree nodes in L_k, and let Y be the children set of these nodes
7 $X \leftarrow X \cup Y$
8 **parallel foreach** $q_i \in X$ **do**
9 **if** a (undeleted) tree node q_j at level k covers q_i **then**
10 remove q_i from X and redirect q_i's parent to q_j
11 Semisort pairs (q_j, q_i) (from Line 2) where $q_i \in X$ and q_j is at the $(k+1)$-th level, and let $\Pi_{q_j} = \{q_i \mid (q_j, q_i)$ *exists*$\}$
12 Initialize a graph $G = (X, \emptyset)$
13 **parallel foreach** $q_i \in X$ **do**
14 Let A_i be q_i's original ancestor at level $k+1$
15 **parallel foreach** $q_j \in \Pi_{A_i}$ **do**
16 **if** $d(q_i, q_j) \le 2^k$ **then**
17 Create an edge between q_i and q_j
18 Compute the MIS of G and let I be the selected vertices
19 Duplicate and insert the tree node $q_i \in I$ at level k
20 Redirect the tree node $q_j \in X \setminus I$ to be the child of a new node $q_i \in I$ that covers q_j (i.e., (q_i, q_j) is an edge in G)
21 $X \leftarrow I$
22 **if** $X \neq \emptyset$ **then**
23 Pick an arbitrary node $q_i \in X$, duplicate it, set it as the root, and link all other nodes as q_i's children

denoted as the set Y (Line 6). We then merge the set Y with the uncovered points in X from the previous level. We first check if other tree nodes at this level can cover these points, and if so, we redirect them to these nodes, and remove them from the set X. Otherwise, we need to promote them to the higher level, but we cannot do so for all points in X since that might violate the separation property. Similar to the batch-insert algorithm, for each point $p_i \in X$, we check all possible conflict points in $X \cap \Pi_{P_i}$, and create an edge if the distance is within 2^k (k is the current level). Then we run the parallel MIS algorithm on the graph, promote the selected ones to level k, and redirect the unselected ones to selected points as parents. Then we repeat this process and move one level up, until we finish all levels. Note that it is possible that X is not empty after we process the root level. In this case, we can use an arbitrary point from X as the new root at level $k+1$, and it can cover all other points since the covering distance is doubled.

Lemma 4.11. *Alg. 5 correctly deletes the batch of points in S from a cover tree T.*

Proof. The correctness proof is similar to the batch insertion algorithm, and we can in turn show that all invariants are still maintained after each loop iteration on Line 5.

All tree nodes are deleted in a bottom-up direction—all leaf nodes first, then their parents, and eventually the root. The invariants of our parallel batch deletion algorithm is that after processing the k-th level (on Line 5), all remaining tree nodes on the k-th level and their subtrees are valid cover trees, and all uncovered tree nodes are captured in the set X.

The analysis is similar to Lem. 4.1 to 4.5 of the insertion algorithm, and we only highlight the difference here. The main difference here in the deletion is that for each uncovered node $q_i \in X$, the conflict set is automatically covered by Π_{A_i} where A_i is q_i's ancestor at level $k+1$. Hence, we do not need the complicated technique to propagate the information between levels, but we can directly generate Π_{A_i} at each level (on Line 11). Other than this, the rest of the correctness proof is identical, including the radius of the conflict sets, the completeness of the conflict sets, and why computing MIS gives a valid tree node set at each level. □

Theorem 4.12. *The batch deletion algorithm (Alg. 4) can correctly delete a set of points S into a cover tree T using $O(c^9 m\mathcal{H}(T))$ expected work and $O(\mathcal{H}(T)\log c(\log c + \log m))$ span* whp*, where $m = |S|$ and $n = |T|$.*

Proof of Thm. 4.12. Lem. 4.11 shows the correctness of Alg. 5. We now consider the work of Alg. 5. There are two major parts that require the most work, one is to try other tree nodes at level k to cover vertices in X (the loop on Line 8), and the other is to construct and run MIS on the conflict graph G. For the first part, for each level, we can remove at most m tree nodes, which uncover at most $c^4 m$ tree nodes (Col. 3.2). Plus another $O(c^4 m)$ nodes from the previous level (will be shown later), $|X| = O(c^4 m)$. For each node $q_i \in X$, let A_i be q_i's ancestor at level $k+1$. Then, q_i will be checked with level k nodes in $B(A_i, 2^{k+1})$, so there can only be c^4 of these nodes (using Lem. 3.1 and $\kappa = 1$). Hence, the work of this part is $O(c^8 m)$ per level, and $O(c^8 m\mathcal{H}(T))$ for all levels. For the second part to construct and run MIS on the conflict graph G, the neighbor size of each node q_i is no more than c^5, which is similar to the insertion algorithm but with no randomization and asymptotical notation. This is because all neighbors of q_i in G must in $B(A_i, 2^{k+1})$ at level $k-1$, so the number of total candidates is bounded (using Lem. 3.1 and $\kappa = 2$). Hence, the total number of edges in G at a certain level is bounded by $O(c^5|X|) = O(c^9 m)$. After running the MIS, there can only be $O(c^4 m)$ selected tree nodes in I (Lem. 3.1 and $\kappa = 1$), and the rest will be covered by the promoted nodes. Combining both parts together gives $O(c^9 m\mathcal{H}(T))$ expected work (the randomized bound is due to semisort). Similar to the insertion algorithm, the span of the deletion algorithm is bounded by computing the MIS on G. Given the graph has $O(c^4 m)$ vertices and $d_{\max} = c^5$ (largest degree), computing the MIS has $O(\log c(\log c + \log m))$ span *whp*, and we need to repeat it for all $\mathcal{H}(T)$ levels. This gives the stated bounds in Thm. 4.12. □

When assuming constant expansion rate ($c = O(1)$) and bounded aspect ratio ($\mathcal{H}(T) = \Theta(\log n)$), Alg. 5 has $O(m \log n)$ expected work and $O(\log n \log m)$ span *whp*.

5 APPLICATIONS

We can use the parallel cover tree to parallelize a list of algorithms in computational geometry and data science, which rely on nearest neighbor search.

5.1 Euclidean Minimum Spanning Tree

Given a set of n points $S \in \mathbb{R}^d$, the Euclidean Minimum Spanning Tree (EMST) problem finds the lowest weight spanning tree in the complete graph on S with edge weights given by the Euclidean distances between points. EMST is one of the earliest and widely studied problems in computational geometry and graph, as Otakar Borůvka gave an algorithm [18] when designing electricity and telegram networks in the 1920s. It is also widely used in applications such as approximating traveling salesman problem (TSP) [38], document clustering [67], analysis of gene expression data [30], wireless network connectivity [45], percolation analyses [9], and modeling of turbulent flows [60].

Given the importance of EMST, many implementations are available (e.g., [7, 19, 49, 52]), although few of them have non-trivial theoretical guarantees ($o(n^2)$ work). Among them, Shamos and Hoey [56] showed algorithms based on Voronoi diagrams, and the work is $O(n \log n)$ on 2D, but $O(n^2 \log n)$ on 3 or higher dimensions. Yao's algorithm [68] has $O((n \log n)^{1.8})$ work on 3D and $O(n^{2-2^{-k-1}} \log^{1-2^{-k-1}} n)$ work on arbitrary dimension. It is widely conjectured that on 3 or higher dimensions, no EMST algorithms exist with $o(n^{1.8})$ work.

However, most real-world datasets are not the worst case, and usually have small expansion constants and bounded aspect ratios. Hence, March et al. [49] in 2010 showed an algorithm that computes the EMST based on Borůvka's MST algorithm [18], and uses a *cover tree* [8] to search for the *nearest neighbor of a cluster* in each step of Borůvka. When assuming a slightly stronger expansion constant and bounded aspect ratio, March et al. [49] showed that the EMST can be constructed using $O(n \log n \log \log n)$ work.

Our new parallel algorithm. Now with the new parallel cover tree, we can show a highly-parallelized EMST algorithm, as shown in Alg. 6. The main body of this algorithm is the classic Borůvka's MST algorithm (Line 2–9), and the details can be found in textbooks (e.g., [40]). We can also construct the cover tree $\mathcal{D}$ in parallel (Thm. 4.10). However, the non-trivial part is for the parallel cluster queries. Unlike most cases that parallel queries are easy, parallel cluster queries need to first delete all points in the cluster from the cover tree $\mathcal{D}$, then it queries the nearest neighbor for all $p \in C$ in $\mathcal{D}$, and finally restores $\mathcal{D}$ by inserting points in C back. Hence, even with the batch-delete algorithm (Alg. 5), we cannot directly apply multiple queries simultaneously.

Our solution is based on *persistent* trees, which means that updates do not destroy the input data structure, but yield a new version as the output. Several recent papers [10, 11, 28, 61, 62] showed that we can design persistent parallel trees using path-copying, which are shown efficient both theoretically and practically. Hence, in Cluster-Query, we copy another version $\mathcal{D}'$ of the original cover tree $\mathcal{D}$ using path-copying. In this way, each Cluster-Query works on a separate version and is not affected by other parallel queries and updates.

Algorithm 6: The parallel EMST algorithm

Input: A set $P = \{p_1, p_2, \ldots, p_n\}$ of points in $\mathbb{R}^d$
Output: The EMST T

```
1  Construct the cover tree D on P
2  S ← {{p_1}, {p_2}, ..., {p_n}}
3  T ← ∅
4  while |S| > 1 do
5      parallel foreach C_i ∈ S do
6          ⟨p_i, q_i⟩ ← Cluster-Query(C_i)
7      Let T' = ∪_i {⟨p_i, q_i⟩} and T ← T ∪ T'
8      Merge the clusters using the tree edges in T' and
        update S
9  return T
10 function Cluster-Query(C)
11     D' ← D.B-Delete(C)
12     parallel foreach p_i ∈ C do
13         q_i ← D'.Query(p_i)
14         d_i ← d(q_i, p_i)
15     i* = arg min_i d_i // Using a parallel reduce
16     return ⟨p_{i*}, q_{i*}⟩
```

Theorem 5.1. *The Euclidean Minimum Spanning Tree (EMST) on n points can be computed in $O(n \log^2 n)$ work in expectation and $O(\log^3 n \log \log n)$ span* whp, *assuming constant cluster expansion, constant dimension, and bounded aspect ratio.*

Proof. For the work bound, each node is in $O(\log n)$ cluster-queries in total for all Borůvka rounds, and cost per node per query is $O(\log n)$ in deletion (Line 11) and query (Line 13) in expectation. Taking the product gives $O(n \log^2 n)$ work in expectation.

For the span bound, constructing the cover tree has the cost of $O(\log^3 n \log \log n)$ *whp*, and the batch-deletion (Line 11) costs $O(\log^2 n)$ span *whp* each, for $O(\log n)$ calls in total.

All other steps in this algorithm are the standard Borůvka steps, and their costs [69] are asymptotically bounded by the cover tree costs. Combining them gives the stated bounds in the theorem. □

5.2 Single-Linkage Clustering

Given a set P of n points, *hierarchical agglomerative clustering (HAC)* starts from every single point as a cluster, and merges two clusters with the global minimum pairwise distance for $n - 1$ iterations, creating a hierarchy for the input points. As a clustering method, hierarchical clustering is a widely used unsupervised learning approach [1, 46, 51], with numerous other applications such as building phylogenetic trees in bioinformatics [50], constructing low-dimension search structures in computer graphics [35, 64], identifying geographic districts in GIS [32, 54] and navigation in robotics [3].

Hierarchical clustering is a high-level framework for clustering a set of objects. When plugging in the cluster distance function (linkage function) $D(X, Y)$ (X, Y are two clusters), one can get a specific algorithm, and the output is clearly defined. The simplest and probably the most widely-used linkage function is *minimum*, defined as $D_m(X, Y) = \min\{d(x, y) \mid x \in X, y \in Y\}$ for $x, y \in P$ and

a metric d. When we use D_m as the linkage function, the resulting clustering is referred to as *single-linkage clustering*.

A theoretically-efficient parallel algorithm for hierarchical clustering is a long-standing open problem—even for the simplest single-linkage clustering in Euclidean space, we are unaware of any previous parallel algorithms using $o(n^2)$ work and $o(n)$ span for $d > 3$, even with assumptions such as low expansion rate.

Using the persistent parallel cover tree, in Sec. 5.1 we show how to generate Euclidean MST using the work and span shown in Thm. 5.1. We note that a recent work by Wang et al. [66] introduced an efficient parallel algorithm that converts an EMST to the dendrogram (cluster tree), which is the output for single-linkage clustering, in $O(n \log n)$ expected work and $O(\log^2 n \log\log n)$ span *whp*. The classic algorithms used Kruskal's algorithm to generate the dendrogram, which is inherently sequential. This new algorithm is quite sophisticated, and uses algorithmic techniques such as the Euler tour, semisorting, and a tricky divide-and-conquer approach. However, this algorithm remains not only theoretically efficient, but also has good practical performance [66]. Combining the new algorithm for EMST as shown in Alg. 6, we get the following result.

Theorem 5.2. *The Single-linkage clustering on n objects can be computed in $O(n \log^2 n)$ expected work and $O(\log^3 n \log\log n)$ span* whp*, assuming constant cluster expansion, bounded aspect ratio, and the pairwise distance function can be computed in $O(1)$ work.*

5.3 Bichromatic Closest Pair (BCP)

Given two sets P_1 and P_2, the goal of bichromatic closest pair (BCP) is to find the closest pair (p_1, p_2), such that $p_1 \in P_1, p_2 \in P_2$, and $d(p_1, p_2) \le d(p_1', p_2') \mid \forall p_1' \in P_1, \forall p_2' \in P_2$.

WLOG, let's assume $|P_1| = m \le n = |P_2|$. We construct a cover tree for P_1, and query the nearest neighbor for every point in P_2 in parallel. Plugging in Thm. 4.10 and Lem. 3.3 gives $O(m \log n)$ expected work and $O(\log^3 n \log\log n)$ span *whp*, assuming constant cluster expansion and bounded aspect ratio.

5.4 Density-Based Clustering

The density-based spatial clustering of applications with noise (DBSCAN) problem takes as input n points $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$, a distance function d, and two parameters ϵ and minPts [33]. A point p is a *core point* if and only if $|B(p, \epsilon)| \ge$ minPts. We denote the set of core points as C. DBSCAN computes and outputs subsets of $\mathcal{P}$, referred to as *clusters*. Each point in C is in exactly one cluster, and two points $p, q \in C$ are in the same cluster if and only if there exists a list of points $\bar{p}_1 = p, \bar{p}_2, \ldots, \bar{p}_{k-1}, \bar{p}_k = q$ in C such that $d(\bar{p}_{i-1}, \bar{p}_i) \le \epsilon$. For all non-core points $p \in \mathcal{P} \setminus C$, p belongs to cluster C_i if $p \in B(q, \epsilon)$ for any $q \in C \cap C_i$. A non-core point belonging to at least one cluster is called a *border point* and a non-core point belonging to no clusters is called a *noise point*. In the analysis, we usually assume that minPtsis a constant, and in practice, we usually pick minPts = 10.

Wang et al. [65] recently showed how to parallelize DBSCAN. Unfortunately, due to the lack of an efficient parallel data structure for nearest neighbor search, their algorithms can only achieve $O(n^2)$ work and polylogarithmic span, or $O((n \log n)^{4/3})$ expected work for $d = 3$ and $O(n^{2-(2/(\lceil d/2 \rceil + 1))+\delta})$ expected work for any constant $\delta > 0$ for $d > 3$, bottlenecked by computing bichromatic closest pairs (BCP). Using the above results for BCP gives $O(n \log n)$ expected work and $O(\log^3 n \log\log n)$ span *whp* to compute DBSCAN. Here the assumptions include: minPts and expansion rate are constant, aspect ratio is bounded, and a pairwise distance can be computed in constant time.

Hierarchical Density-Based Clustering. The output for hierarchical clustering (HDBSCAN) is a dendrogram (cluster tree), similar to single-linkage clustering. The only difference is that HDBSCAN has the parameter minPts, so a point needs to first compute its minPts-nearest neighbors. This can be achieved efficiently by constructing a cover tree in parallel, querying for all points, and then using single-linkage clustering on top of it. Using the same assumptions in DBSCAN, HDBSCAN can be computed in $O(n \log^2 n)$ expected work and $O(\log^2 n \log\log n)$ span *whp*.

5.5 k-NN Graph Construction

k-NN graphs are widely used in machine learning, such as graph clustering [34, 42, 47, 48], manifold learning [63], outlier detection [37], and proximity search [20, 53, 55]. Given a point set P in a metric space, a k-NN graph is a directed graph $G = (V, E)$, where $V = P$ and $(p, q) \in E$ if q is one of p's k-nearest neighbor in $V - \{p\}$. We first construct the cover tree on P, then apply k-NN queries on all the points in P in parallel, and finally construct the k-NN graph according to the query results. Using our parallel cover tree, we can get $O(kn \log k \log n)$ expected work and $O(\log n \cdot (k \log k + \log^2 n \log\log n))$ span *whp* by combining Thm. 4.10 and Lem. 3.3. Here we again assume constant expansion rate and bounded aspect ratio.

6 CONCLUSIONS

In this paper, we show parallel algorithms for batch insertions and batch deletions on cover trees, which are work-efficient and have polylogarithmic span. The key challenge is that the operations on the sequential cover tree, as well as many other sequential data structures with similar functionality, are processed in a depth-first manner that is inherently sequential. We show a few algorithmic ideas in this paper, and we highlight the technique to construct conflict graphs and compute the feasible set of tree nodes using maximal independent set (MIS) on the graphs. This technique enables a depth-first algorithm to be executed in a breadth-first order. We believe that this idea may of independent interest, and we will study if we can apply it to parallelize other sequential algorithms and data structures. One of such examples is the metric skip lists [41], which provide similar (but randomized) query and update costs to cover trees but do not need to assume bounded aspect ratio. We also plan to study other graph algorithms with similar challenges, and practical nearest-neighbor search algorithms and see if we can show theoretical guarantees parameterized by the expansion rate.

Acknowledgement.

ACKNOWLEDGEMENT

This work is supported by NSF grant CCF-2103483.

REFERENCES

[1] C. C. Aggarwal and C. K. Reddy. Data clustering: Algorithms and applications. *Chapman&Hall/CRC Data mining and Knowledge Discovery series, Londra*, 2014.

[2] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[3] O. Arslan, D. P. Guralnik, and D. E. Koditschek. Coordinated robot navigation via hierarchical clustering. *IEEE Transactions on Robotics*, 32(2):352–371, 2016.

[4] A. Authors. Many sequential iterative algorithms can be parallel and (almost) work-efficient. *(unpublished work, submitted to SPAA 2022)*, 2022.

[5] J. Bell. The uniform metric on product spaces. *Lecture Notes, University of Toronto.*

[6] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[7] J. L. Bentley and J. H. Friedman. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Trans. on Comput.*, 27(02):97–105, 1978.

[8] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *International Conference on Machine Learning (ICML)*, pages 97–104, 2006.

[9] S. P. Bhavsar and R. J. Splinter. The superiority of the minimal spanning tree in percolation analyses of cosmological data sets. *Monthly Notices of the Royal Astronomical Society*, 282(4):1461–1466, 1996.

[10] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[11] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[12] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[13] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[14] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[15] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. *J. ACM*, 2020.

[16] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[18] O. Boruvka. O jistém problému minimálním. *Práce Mor. Prırodved. Spol. v Brne (Acta Societ. Scienc. Natur. Moravicae)*, 3(3):37–58, 1926.

[19] S. Chatterjee, M. Connor, and P. Kumar. Geometric minimum spanning trees with geofilterkruskal. In *International Symposium on Experimental Algorithms (SEA)*, pages 486–500. Springer, 2010.

[20] E. Chávez and E. Sadit Tellez. Navigating k-nearest neighbor graphs to solve nearest neighbor searches. In *Advances in Pattern Recognition*, pages 270–280, 2010.

[21] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 339–350, 2017.

[22] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

[23] K. L. Clarkson et al. Nearest-neighbor searching and metric space dimensions. *Nearest-neighbor methods for learning and vision: theory and practice*, pages 15–59, 2006.

[24] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[26] R. R. Curtin. *Improving dual-tree algorithms*. PhD thesis, Georgia Institute of Technology, 2015.

[27] L. Dhulipala, G. E. Blelloch, Y. Gu, and Y. Sun. Pac-trees: Supporting parallel and compressed purely-functional collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[28] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.

[29] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Semi-asymmetric parallel graph algorithms for nvrams. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[30] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95(25):14863–14868, 1998.

[31] Y. Elkin and V. Kurlin. A new compressed cover tree guarantees a near linear parameterized complexity for all k-nearest neighbors search in metric spaces. *arXiv preprint:2111.15478*, 2021.

[32] D. Eppstein, M. T. Goodrich, D. Korkmaz, and N. Mamano. Defining equitable geographic districts in road networks via stable matching. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–4, 2017.

[33] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[34] P. Franti, O. Virmajoki, and V. Hautamaki. Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1875–1881, 2006.

[35] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch. Efficient BVH construction via approximate agglomerative clustering. In *High-Performance Graphics (HPG)*, 2013.

[36] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[37] V. Hautamaki, I. Karkkainen, and P. Franti. Outlier detection using k-nearest neighbour graph. In *International Conference on Pattern Recognition*, volume 3, pages 430–433, 2004.

[38] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[39] M. Izbicki and C. Shelton. Faster cover trees. In *International Conference on Machine Learning (ICML)*, pages 1162–1170. PMLR, 2015.

[40] J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[41] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 741–750, 2002.

[42] G. Karypis, E.-H. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[43] T. Kollar. Fast nearest neighbors, 2006.

[44] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 798–807. Citeseer, 2004.

[45] D. Li, X. Jia, and H. Liu. Energy efficient broadcast routing in static ad hoc wireless networks. *IEEE Transactions on Mobile Computing*, 3(2):144–151, 2004.

[46] G. Lin, C. Nagarajan, R. Rajaraman, and D. P. Williamson. A general approach for incremental approximation and hierarchical clustering. *SIAM J. on Computing*, 39(8):3633–3669, 2010.

[47] M. Lucińska and S. T. Wierzchoń. Spectral clustering based on k-nearest neighbor graph. In *Computer Information Systems and Industrial Management*, pages 254–265, 2012.

[48] M. Maier, M. Hein, and U. Von Luxburg. Optimal construction of k-nearest-neighbor graphs for identifying noisy clusters. *Theoretical Computer Science*, 410(19):1749–1764, 2009.

[49] W. March, P. Ram, and A. Gray. Fast Euclidean minimum spanning tree: Algorithm, analysis, and applications. In *KDD*, 2010.

[50] S. J. Matthews and T. L. Williams. Mrsrf: an efficient mapreduce algorithm for analyzing large collections of evolutionary trees. *BMC bioinformatics*, 11(S1):S15, 2010.

[51] B. Moseley, S. Vassilvtiskii, and Y. Wang. Hierarchical clustering in general metric spaces using approximate nearest neighbors. In *International Conference on Artificial Intelligence and Statistics*, pages 2440–2448. PMLR, 2021.

[52] G. Narasimhan and M. Zachariasen. Geometric minimum spanning trees via well-separated pair decompositions. *J. Experimental Algorithmics*, 6:6–es, 2001.

[53] R. Paredes and E. Chávez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *String Processing and Information Retrieval*, pages 127–138, 2005.

[54] J. P. Praene, B. Malet-Damour, M. H. Radanielina, L. Fontaine, and G. Riviere. Gis-based approach to identify climatic zoning: A hierarchical clustering on principal component analysis. *Building and Environment*, 164:106330, 2019.

[55] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *International Conference on Pattern Recognition (ICPR)*, 2002.

[56] M. I. Shamos and D. Hoey. Closest-point problems. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 151–162. IEEE, 1975.

[57] M. Sharma and R. Joshi. Design and implementation of cover tree algorithm on cuda-compatible gpu. *International Journal of Computer Applications*, 975:8887, 2010.

[58] Z. Shen, Z. Wan, Y. Gu, and Y. Sun. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

[59] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015.

[60] S. Subramaniam and S. Pope. A mixing model for turbulent reactive flows based on euclidean minimum spanning trees. *Combustion and Flame*, 115(4):487–514,

1998.

[61] Y. Sun and G. E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

[62] Y. Sun, D. Ferizovic, and G. E. Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

[63] J. B. Tenenbaum, V. d. Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[64] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing*, pages 81–86, 2008.

[65] Y. Wang, Y. Gu, and J. Shun. Theoretically-efficient and practical parallel dbscan. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 2555–2571, 2020.

[66] Y. Wang, S. Yu, Y. Gu, and J. Shun. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1982–1995, 2021.

[67] P. Willett. Recent trends in hierarchic document clustering: a critical review. *Information processing & management*, 24(5):577–597, 1988.

[68] A. C.-C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM J. on Computing*, 11(4):721–736, 1982.

[69] W. Zhou. A practical scalable shared-memory parallel algorithm for computing minimum spanning trees. Master's thesis, Karlsruhe Institute of Technology, 2017.