Parallel Cover Trees and their Applications

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Zachary Napier
UC Riverside
znapi001@ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Letong Wang
UC Riverside
lwang323@ucr.edu

ABSTRACT

The cover tree is the canonical data structure that efficiently maintains a dynamic set of points on a metric space and supports nearest and k-nearest neighbor searches. For most real-world datasets with reasonable distributions (constant expansion rate and bounded aspect ratio mathematically), single-point insertion, single-point deletion, and nearest neighbor search (NNS) only cost logarithmically to the size of the point set. Unfortunately, due to the complication and the use of depth-first traversal order in the cover tree algorithms, we were unaware of any parallel approaches for these cover tree algorithms.

This paper shows highly parallel and work-efficient cover tree algorithms that can handle batch insertions (and thus construction) and batch deletions. Assuming constant expansion rate and bounded aspect ratio, inserting or deleting m points into a cover tree with n points takes $O(m \log n)$ expected work and polylogarithmic span with high probability. Our algorithms rely on some novel algorithmic insights. We model the insertion and deletion process as a graph and use a maximal independent set (MIS) to generate tree nodes without conflicts. We use three key ideas to guarantee work-efficiency: the prefix-doubling scheme, a careful design to limit the graph size on which we apply MIS, and a strategy to propagate information among different levels in the cover tree. We also use path-copying to make our parallel cover tree a persistent data structure, which is useful in several applications.

Using our parallel cover trees, we show work-efficient (or near-work-efficient) and highly parallel solutions for a list of problems in computational geometry and machine learning, including Euclidean minimum spanning tree (EMST), single-linkage clustering, bichromatic closest pair (BCP), density-based clustering and its hierarchical version, and others. To the best of our knowledge, many of them are the first solutions to achieve work-efficiency and polylogarithmic span assuming constant expansion rate and bounded aspect ratio.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; **Data structures design and analysis**.

KEYWORDS

cover tree, parallel algorithms, parallel data structures, nearest neighbor search, euclidean minimum spanning tree, single-linkage clustering

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538581

ACM Reference Format:
Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490148.3538581

1 INTRODUCTION

Nearest neighbor search (NNS) on a metric space is one of the most widely-used primitives in algorithm design, which has applications in computational geometry, computer graphics, machine learning, computer vision, and many other areas. Finding (or even approximating) nearest neighbor for general metrics requires linear time[1]. However, the metrics in real-world applications and of practical interest usually do exhibit nice properties that can be exploited. Some widely-studied properties include the low expansion rate, which indicates that the density of points in the metric space changes smoothly, and bounded aspect ratio (defined in Sec. 2),

A canonical data structure that exploits such a property is the cover tree [8], which is usually considered as the "standard" solution for NNS on metric space, both theoretically and practically. Theoretically, the point-insert, point-delete, and NNS query only take logarithmic time, assuming the metric space has constant expansion rate and bounded aspect ratio. Practically, highly-optimized software for cover trees from [8, 39] demonstrates good performance on a large variety of real-world instances. Cover tree is simpler than several related data structures around the time [23, 41, 44], but it is only simple *conceptually*. Although cover trees were proposed in 2006, many tricky details and corrections are discussed and made in later works [26, 31, 43]. In fact, it was only until recently that Elkin and Kurlin [31] corrected the single-update and query bounds, which will be reviewed in Sec. 3.

A cover tree T organizes a set S of points in some metric space [8]. It consists of a number of levels. Every level consists of nodes, each corresponding to a point in S. Note that a point can correspond to multiple nodes across different levels. The cover tree is defined to maintain three key invariants (formal definition in Sec. 3): (1) **Nesting**: tree nodes at level i is a subset of nodes at level $i-1$. (2) **Covering tree**: for all i, each tree node at level $i-1$ must be covered by some tree nodes in level i within distance 2^i (one corresponding node at level i will be the parent of that at level $i-1$). (3) **Separation**: any two tree nodes at level i are separated by distance 2^i. We show an illustration in Fig. 1. A cover tree for a set S of points is not unique. As long as the three invariants hold, the logarithmic bound for point insertion, deletion, and NNS query holds (assuming low expansion rate and bounded aspect ratio).

Given the wide applications of cover trees, parallelizing it is of interest both in theory and in practice. There are two major

[1]A simple example is a uniform metric space [5] where all pairwise distances are similar so that we can take no structural advantage [8].

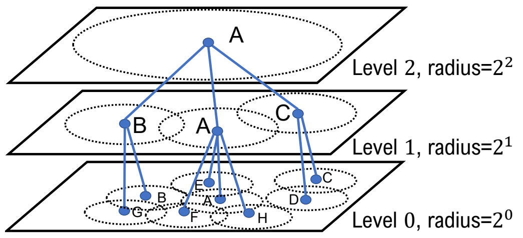

Figure 1: An example cover tree with 8 points in 3 levels. No points are in other points' circles at the same level (separation distance), and each point is in at least one point's circle at a higher level (covering distance).

challenges in parallelizing it. The first one is the complication in the algorithms mentioned above, especially in the analysis. The second and main reason is that the existing cover tree algorithms are also inherently sequential—both the insertion and the deletion are in a depth-first order (see Alg. 1 and 2: the algorithms need to traverse as deep as possible, and modify the cover tree during backtracking). Hence, the result of a single-insert/delete can drastically change the position of another operation. An example is presented in Fig. 2. The two inserted nodes, even when they will be inserted to different branches in the tree, can interact and conflict with each other. Hence it is highly non-trivial to parallelize the original algorithm to handle parallel insertions on two or more points. To solve such conflicts, we need some proximity information for each point (e.g., the points close to it), but such information is not known before the point is inserted into the cover tree. This chicken-and-egg issue adds extra difficulty to parallelizing the cover tree. To the best of our knowledge, we are only aware of two papers that "claimed" that they parallelized the cover tree [39, 57]. Sharma and Joshi's algorithm [57] missed many details and it remains unclear what their algorithms and the cost bounds are. Izbicki and Shelton's version [39] relaxed the separation property in cover trees. Hence, their tree is more similar to a quadtree, and the cost of an NNS can be linear as opposed to logarithmic in the original cover tree [8]. As a result, it has been open for 15 years on parallelizing the cover tree proposed in 2006, while maintaining logarithmic query and update costs.

Contribution in this paper. We show highly parallel and work-efficient cover tree algorithms that can handle batch insertions (and thus construction) in Alg. 4 and batch deletions in Alg. 5. In particular, we show algorithms with the following bounds on work (number of operations) and span (longest dependent operations, formally defined in Sec. 2):

Theorem 1.1. *Assuming constant expansion rate and bounded aspect ratio, inserting or deleting m points into a cover tree with n points takes $O(m \log n)$ expected work and polylogarithmic span with high probability.*

This also indicates that constructing a cover tree of size n takes $O(n \log n)$ expected work and polylogarithmic span with high probability. The more precise versions of the results are given in Thm. 4.9, 4.10 and 4.12. Our algorithms are work-efficient (modulo randomization) as they spend the same operations as the sequential algorithm. NNS queries are naturally parallelized since they do not modify the cover tree, but cluster queries, as discussed in many applications in Sec. 5, require batch-deletion on the persistent parallel cover tree.

We note that designing the parallel algorithms for batch-insertion and batch-deletion is highly non-trivial, both in algorithm design and in analyzing the correctness and cost bounds. Our key idea is that, instead of inserting or deleting the points using depth-first traversal, we consider all points to be inserted or deleted at each level as a whole, and process the levels either top-down (for insertion) or bottom-up (for deletion). Take the batch insertion as an example. To process a set of points S to be inserted at level i, the main challenge is to identify a subset S' of them that can be added to this level, such that (1) they are well-separated and (2) all other points in S will be covered either by the existing points at this level or points in S'. Our key insight is to model the points as a graph G to illustrate the pairwise conflict relationship (i.e., two points cannot both be put at this level because they are too close to each other). Then any maximal independent set (MIS) on this graph gives a feasible set of points that can be inserted at this level, while other points are guaranteed to be covered by some of the selected points at this level. Using this MIS approach gives a valid cover tree, but its efficiency is not straightforward—considering constructing a cover tree by batch-inserting n points to an empty tree. Constructing the graph G already takes at least $O(n^2)$ work, but the sequential algorithm only uses $O(n \log n)$ work. To make our algorithm work-efficient and highly parallel, in addition, we need to consider: (1) the prefix doubling approach such that the "additional information" that each sub-batch adds to the cover tree is limited (to bound the number of edges in G); (2) to minimize the number of point pairs to check when constructing edges in G and avoid checking pairwisely in the batch; and (3) to efficiently propagate information between the levels. With detailed solutions and justifications given in Sec. 4, we can show that with our design, the additional work to resolve the conflicts caused by parallel insertions (constructing G and running MIS on G) is asymptotically bounded by the work of the sequential insertion, in expectation. Also, using parallel MIS [4, 12] with polylogarithmic span enables polylogarithmic span for the entire algorithm since a cover tree has $\Theta(\log n)$ levels assuming low expansion rate and bounded aspect ratio. Putting all pieces together, our batch-insertion algorithm is work-efficient and highly parallel, with the cost bounds given in Thm. 4.9 and 4.10. Batch-deletion can be solved similarly, but it does not require the prefix-doubling scheme.

Given the wide applications of NNS in computational geometry and machine learning, our new algorithms for cover tree give the first work-efficient and highly parallel solutions to a list of problems given in Sec. 5, again assuming low expansion rate and bounded aspect ratio. While for many of the applications, we can just replace the sequential cover tree with our parallel one, we highlight our algorithm for Euclidian minimum spanning tree (EMST) and single-linkage clustering. To support parallel cluster queries in these applications, we need to make our cover tree a functional data structure. For single-linkage clustering, which is one of the most widely-used methods for hierarchical agglomerative clustering, we combine our parallel EMST algorithm with a recent algorithm by Wang et al. [66], and achieve the first nearly-work-efficient parallel algorithm with polylogarithmic span.

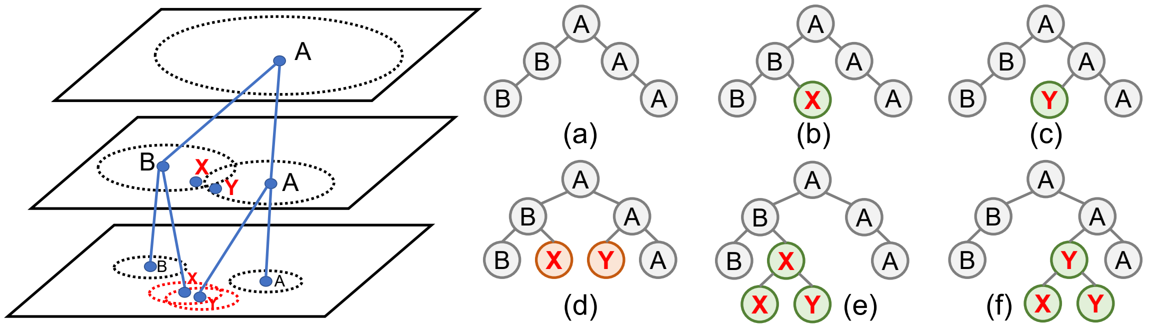

Figure 2: An example of the challenge for parallel insertion. Points A and B are already in the tree and we now want to insert points X and Y. Figure (a) shows the original cover tree before the insertions, and figures (b) and (c) give the tree if either X or Y is inserted, respectively. However, if both insertions are applied in parallel and independently, then the tree shown in figure (d) becomes invalid since points X and Y are not separated in this level. In the sequential algorithm, figure (e) shows the final cover tree when we first insert X and then Y, and figure (f) gives the tree after inserting point Y then X. Both versions are valid cover trees, but a correct parallel algorithm needs to identify the potential conflict between point pairs like X and Y, and yield a correct output tree (either figure (e) or (f)). Our solution requires a few key observations and an efficient parallel MIS (maximal independent set) algorithm as a subroutine, and is explained in more details in Sec. 4.1.

2 PRELIMINARIES

We use the term $O(f(n))$ **with high probability (*whp*)** in n to indicate the bound $O(kf(n))$ holds with probability at least $1-1/n^k$ for any $k \geq 1$. With clear context we drop "in n". We use $\log n$ as a short form of $1+\log_2(n+1)$.

Low-Expansion Metric Space. A metric (X, d_X) is defined on a set X and with a distance function $d : X \times X \to \mathbb{R}^*$ that satisfies properties: (1) $d_X(x,y) = 0 \Leftrightarrow x = y$ for $x, y \in X$, (2) $d_X(x,y) = d_X(y,x)$ for $x, y \in X$, and (3) $d_X(x,y) \leq d_X(x,z) + d_X(z,y)$ for $x, y, z \in X$. With clear context, we drop the superscript X.

Define $B_X(p,r) = \{x \in X \mid d(p,x) \leq r\}$ as the closed ball centered at point p and containing all points in X at a distance of at most r from p. With clear context, we drop X. We say a metric has (ρ, c)-***expansion*** [41] *iff* for all $p \in X$ and $r > 0$,

$$|B(p,r)| \geq \rho \implies |B(p,2r)| \leq c \cdot |B(p,r)|.$$

The parameter c is referred to as the expansion rate of the metric space, and we say a metric has a low or constant expansion rate if $c = O(1)$. Usually we assume ρ is $O(\log |X|)$, which guarantees constant expansions for most real-world datasets. Intuitively, low expansion means a smooth distribution of the points, and rules out the case where as the ball grows, we encounter a few points, then a long distance with no points, then suddenly a tremendous number of points. This case is also less likely in most real-world datasets.

Theorem 2.1 (Sampling Theorem [41]). *Given a metric with (ρ, c)-expansion, a uniformly random subset X' with $|X'| = m$ will have* $(\max(c\rho, O(\log m)), 2c)$*-expansion with high probability to m.*

We note that if we want to improve the probability to the size of n, then the analysis in [41] implies that we just change $O(\log m)$ in the parameter to $O(\log n)$.

This theorem shows that for low-expansion metric space, a sample of this space is also low-expansion. This is crucial for our randomized batch-insertion algorithm in Sec. 4.1.

The query cost for cover trees relies on the expansion rate including or excluding the query point(s). For simplicity, we still use c to denote the expansion rate after such modifications, if it is increased. In particular, for single-point query and bichromatic closest pairs, the query expansion rate includes the query point; for Euclidean MST and single-linkage clustering, the query expansion rate excludes the query points.

Bounded Aspect Ratio. Bounded aspect ratio is another common assumption for real-world datasets. Aspect ratio is defined as $\Delta = \frac{\max\{d(x,y) \mid x,y \in X\}}{\min\{d(x,y) \mid x,y \in X\}}$. A common assumption is that the aspect ratio is bounded—if the input size is n, then the aspect ratio is n^κ for some constant $\kappa > 0$. For instance, the ratio of the Earth's radius to a sand's radius is only about $10^{10} \approx 2^{33}$. Most of the real-world applications have aspect ratios smaller than this value. Hence, for big-data applications (usually $n \geq 10^6$), it is reasonable to assume $\Delta = n^\kappa$ for some constant $\kappa > 0$, and $\log \Delta = O(\log n)$. The height of a cover tree $\mathcal{H}(T)$ is $\lceil 1 + \log_2 \Delta \rceil$. For theoretical accuracy, we leave $\mathcal{H}(T)$ as a parameter when analyzing existing and our new cover tree algorithms, while in practical applications we can assume $\mathcal{H}(T) = O(\log_2 \Delta) = O(\log n)$.

For simplicity, throughout this paper, we assume $\min d(x,y) = 1$, so aspect ratio is simplified as $\Delta = \max d(x,y)$. Note that this is only for the ease of description (e.g., the leaf level in a cover tree is always level 0), and none of the previous cover tree algorithms or our new ones rely on this assumption.

Functional Trees / Persistent Trees. Functional data structures are data structures that are immutable (i.e., no operations modify existing data), and thus any update operation will create a new version instead of updating in place. For tree-based data structures, recent work [10, 11, 27, 28, 61, 62] showed that using path-copying is a both theoretically and practically efficient approach to support functional trees. In particular, any update (including insertion or deletion) will copy the full path from the root to the node (or nodes, for multi-point queries) to be updated. In this way, we can maintain copies of all history versions of the tree structure, which will be useful for many applications in Sec. 5. An illustration of a persistent tree using path-copying is presented in Fig. 3.

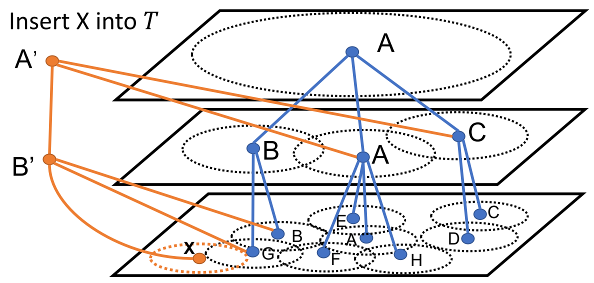

Figure 3: A functional cover tree using path copying. The blue nodes and lines show the original cover tree T. When inserting a new point X, it follows the path from $A \longrightarrow B$ and will be finally inserted as a child of B. A functional insertion will copy all nodes on the path (i.e., A to A' and B to B'), and insert them to the copied nodes. The tree nodes in the original cover tree are immutable. Finally, following the root of A', we can find the cover tree T' with the new point X inserted, and the original cover tree T is still accessible by reading the root A.

Computational Model and Notations. We use the work-span model for fork-join parallelism with binary forking to analyze parallel algorithms [11, 25], which is recently used in many papers on parallel algorithms (a short list: [2, 6, 13, 14, 21, 22, 24, 29]).

We assume a set of threads that share a common memory. Each thread supports standard RAM instructions, and a `fork` instruction that forks two new child threads. When a thread performs a `fork`, the two child threads all start by running the next instruction, and the original thread is suspended until all children terminate. A computation starts with a single root thread and finishes when that root thread finishes. An algorithm's ***work*** is the total number of instructions and the ***span*** (depth) is the length of the longest sequence of dependent instructions in the computation. We can execute the computation efficiently using a randomized work-stealing scheduler both in theory and in practice [11, 17, 25].

Some subcomponents (MIS and semisort, see details below) used in this paper need constant-cost atomic operation TestAndSet(p), which reads and attempts to set the boolean value pointed to by p to *true*. It returns *true* if successful and *false* otherwise.

Parallel Maximal Independent Set. For a graph $G = (V, E)$, an independent set is a set of vertices $V' \subseteq V$, such that for any $u, v \in V'$, $(u, v) \notin E$. A maximal indepent set (MIS) is an independent set V' where $\forall v \in V, v \notin V', V' \cup \{v\}$ is not an independent set. In parallel, the problem can be solved in $O(|E|)$ work and $O(\log^3 |V|)$ span *whp*. Our recent work [58] improved the span to $O(\log |V| \log d_{max})$ *whp* where d_{max} is the maximum degree of any vertex in the graph G. In this paper, we will use parallel MIS as a subcomponent.

Semisort. Given a sequence of key-value pairs, a ***semisort*** algorithm reorders the element in the sequence such that elements with the same key are contiguous [36]. In parallel, semisort can be solved in $O(n)$ expected work and $O(\log n)$ span *whp* [11].

3 COVER TREES

This section overviews the cover tree structure [8] and its various properties shown in [8, 31, 43].

A cover tree consists of a number of levels that are indexed by the integer i, which decreases as the levels are descended. Every level consists of nodes, each of which corresponds to a unique point in the data set S. Note that each point can correspond to multiple nodes across different levels.

Let C_i denote the set of points in S associated with the nodes at level i. The cover tree is defined to maintain three key invariants:

(1) **Nesting**. $C_i \subset C_{i-1}$. This means that a point that is associated with a node at one level is also associated with a node at every level below it.
(2) **Covering tree**. For every $p \in C_{i-1}$, there exists a $q \in C_i$ such that $d(p, q) \leq 2^i$ and the node at level i associated with q is a parent of the node at level $i - 1$ associated with p.
(3) **Separation**. For all distinct $p, q \in C_i$, $d(p, q) > 2^i$.

For simplicity, we assume $d(p, q) \geq 1$ for all $p, q \in S$, so all critical levels in the cover tree has $i \geq 0$.

Finally, we differentiate a few different versions using compression. The plain version stores the cover tree in $\lceil 1 + \log_2 \Delta \rceil$ levels, so each point can show up at multiple levels. The tree height is then $\mathcal{H}(T) = \lceil 1 + \log_2 \Delta \rceil$. To compress the tree, one can either compress a tree node with one child (so the tree has at most $2n - 1$ nodes) [8], or consider all tree nodes corresponding to the same point as a supernode [31]. We use the plain version throughout this paper since parallelizing cover trees is already very challenging. Meanwhile, we believe that our techniques apply to the compressed versions, and we leave that as future work.

3.1 Cover Tree Properties

Below are some important lemmas for cover trees that will be used in designing algorithms for parallel cover trees.

Lemma 3.1. *For each point p, the number of points at level i which falls into ball $B(p, 2^{i+\kappa})$ is $c^{3+\kappa}$ for any non-negative integer κ.*

Proof. Let $Q = B(p, 2^{i+\kappa}) \cap C_i$ (C_i is the set of points at level i) be the set of points described in Lem. 3.1. For all $q \in Q$, since they are all at level i, the balls $B(q, 2^{i-1})$ must be disjoint. The idea is then to bound the number of such disjoint balls around p. Now consider all points within ball $B(p, 2^{i+\kappa+1})$, which is a superset of Q. We will then discuss the number of disjoint balls $B(q, 2^{i-1})$ one can pack inside $B(p, 2^{i+\kappa+1})$. For any $q \in Q$, since $q \in B(p, 2^{i+\kappa})$, we have $d(p, q) \leq 2^{i+\kappa}$, and thus $B(p, 2^{i+\kappa+1}) \subseteq B(q, 2^{i+\kappa+2})$. Therefore,

$$|B(p, 2^{i+\kappa+1})| \leq |B(q, 2^{i+\kappa+2})| \leq c^{3+\kappa}|B(q, 2^{i-1})|$$

Note that all balls $B(q, 2^{i-1})$ must be contained in ball $B(p, 2^{i+\kappa+1})$. This proves that the total number of such points q can be no more than $|B(p, 2^{i+\kappa+1})|/|B(q, 2^{i-1})| \leq c^{3+\kappa}$. □

We note that this lemma can be viewed as a simplified form of the packing lemma in [31]. Since this form is exactly what is needed to analyze our parallel cover tree, we provide this analysis here, and hopefully it provides some insights to the readers on how the expansion rate affects the properties of the cover tree. Based on this lemma, it is easy to bound the number of children for any tree node in a cover tree.

Corollary 3.2. *The number of children of any tree node is $\leq c^4$ in a cover tree.*

Algorithm 1: Single-Point Insert(p, Q_k, k).

```
Input: The point $p$ to be inserted, a cover set $Q_k$ and a level $k$.
Output: A cover tree that includes the new point $p$.

$Q \leftarrow \{\text{Children}(q) \mid q \in Q_k\}$
if $d(p,Q) > 2^k$ then return false
else
    $Q_{k-1} \leftarrow \{q \in Q \mid d(p,q) \leq 2^k\}$
    if Insert($p, Q_{k-1}, k-1$) = false then
        if $d(p,Q_k) \leq 2^k$ then
            $q \leftarrow$ any point in $Q_k$ satisfying $d(p,q) \leq 2^k$
            Insert $p$ into $q$'s children
            return true and exit
        else
            return false
```

For a tree node at level i, all children are at level $i-1$ and must be in $B(p, 2^i)$. Plugging in $\kappa = 1$ gives the stated bound. This is a simple use case of Lem. 3.1, and in Sec. 4, we will extensively use it to bound the work and span for the parallel cover tree algorithms.

Lemma 3.3 (Query cost [31]). *A nearest neighbor query takes $O(c^{10}\mathcal{H}(T))$ work, and a k-nearest neighbor query takes $O(c^7(k+c^3)\log k \cdot \mathcal{H}(T))$ work.*

The analyses are given by Elkin and Kurlin in [31]. The nearest neighbor query algorithm loops over all tree levels in a top-down manner. At each level, it visits at most c^6 tree nodes. Then the algorithm will check all their children, multiplying another c^4. The k-NN search can be analyzed similarly.

3.2 Sequential Cover Tree Algorithms

We now review a few useful sequential primitives on cover trees. Elkin and Kurlin [31] recently pointed out some fatal issues in the analysis of the original cover tree paper. Here, we formally analyze the sequential algorithms first. We do not consider the analysis in this section as a contribution of this paper, but we need the results to correctly bound our parallel algorithms.

Single-point insertion. We present the single insertion algorithm on the cover tree in Alg. 1, originally from [43].

It iterates over the levels of the tree from top to bottom, and at each level it has a set Q_i of nodes that p could possibly be a descendent of. Specifically, any node within a distance of 2^{i+1} from p is a candidate. In the first iteration, Q_i contains only the root node. We construct a set Q_{i-1} for the next level down by taking all the children of nodes in Q_i and filtering out those that are not candidates. We then make a recursive call with Q_{i-1}, which represents an attempt to insert p as a descendent of one of Q_{i-1}. If that fails, then nodes for p at any levels below i fulfill the separation condition, so if a node at this level in Q_i covers p, we can insert p as its child and exit. If no such node covers p, then we return false.

Theorem 3.4 (Single insertion work). *A single point insertion uses $O(c^5\mathcal{H}(T))$ work.*

Proof. The insertion algorithm traverses all levels in the tree, and visits all tree nodes q_i at level $k-1$ if $d(p,q_i) \leq 2^k$, and their children q'_j. Note that all balls $B(q'_j, 2^{k-1})$ must be in $B(p, 2^{k+1})$, so based on Lem. 3.1, there can only be c^5 of such points. Hence, the insertion algorithm uses $O(c^5\mathcal{H}(T))$ work. □

Algorithm 2: Single-Point Delete(p, $\{Q_k, Q_{k+1}, ..., Q_\infty\}$, k).

```
Input: The point $p$ to be deleted, a set of cover sets $\{Q_k, Q_{k+1}, ..., Q_\infty\}$, and the current level $k$.
Output: The modified cover tree that excludes the point $p$.

$Q \leftarrow \{\text{Children}(q) : q \in Q_k\}$
$Q_{k-1} \leftarrow \{q \in Q \mid d(p,q) \leq 2^k\}$
Delete($p$, $\{Q_{k-1}, Q_k, ..., Q_\infty\}$, $k-1$)
if $d(p,Q) = 0$ then
    Remove $p$ from $C_{k-1}$ and from the children of Parent($p$)
    for $p' \in$ Children($p$) do
        $k' \leftarrow k-1$
        while $d(p', Q_{k'}) > 2^{k'}$ do
            Insert $p'$ into $C_{k'}$ (and $Q_{k'}$)
            $k' \leftarrow k'+1$
        $q' \leftarrow$ any point in $Q_{k'}$ satisfying $d(p',q') \leq 2^{k'}$
        Make $q'$ as the parent of $p'$
```

Here this bound seems tighter than the bound by Elkin and Kurlin [31], given that their bound is $O(c^8\mathcal{H}(T))$. However, we note that the definition of tree height in [31] is different from us—their tree height only counts for non-empty levels while our tree height includes all levels. Hence, either bound can be better, decided by the input distribution.

Single-point deletion. The deletion algorithm again starts from the top of the tree and goes down one level at a time, and again at each iteration it starts with a set of nodes at this level Q_i that are within two times the covering distance of p. The recursive call will remove p from every level below this level, and also find new parents for the orphaned nodes. After the recursive call returns, we only need to remove p from this level and find parents for the newly orphaned nodes. For each orphaned node, it must be covered by some node in Q, so we just check each Q in order, starting from this level and going up the levels until we find the first level that covers us. The only exception is when the root is deleted, and we can process that separately. Then once we find the new parent of the orphaned node, we will update the pointers.

Lemma 3.5 (Single deletion work). *A single point deletion uses $O(c^8\mathcal{H}(T))$ work.*

Proof. Beygelzimer et al. [8] showed that in the deletion algorithm, only one point from each level can promote more than two levels (Line 8–10). At each level, the deleted point p has at most c^4 children, which is compared to at most c^4 tree nodes in $Q_{k'}$ for at most twice. Given that the tree height is $\mathcal{H}(T)$, multiplying the three terms gives the stated single deletion work bound. □

Traversing the cover tree. Another commonly used sequential algorithm on cover trees is to traverse the tree w.r.t. a point p, in order to extract all tree nodes covering p (based on the corresponding radius at each level). The set of these nodes is simply the concatenation of all the sets Q' in each iteration of Line 4. The algorithm is given in Alg. 3. Similar to single insertion/deletion, we keep track of

Algorithm 3: Traverse(T, p)

```
Input: The cover tree $T$ and the point $p$ to be checked.
Output: A list of tree nodes $N$ that contains tree nodes $q_i$
        on level $k$ such that $d(p, q_i) < 2^{k+1}$.

1 $N \leftarrow \{virtual\text{-}root\}$
2 $Q \leftarrow \{T.root\}$
3 for $k$ from root level to leaf level do
4     $Q' \leftarrow \{q \in Q \mid d(p, q) < 2^{k+1}\}$
5     $N \leftarrow N \cup Q'$
6     $Q \leftarrow \{\text{Children}(q) \mid q \in Q'\}$
7 return $N$
```

the tree nodes that are sufficiently close to the point, in a top-down manner. The cost for traversal is the same as insertion since they touch the same set of tree nodes.

We note that the traversal cost is cheaper than an NNS on the cover tree (Lem. 3.3). The reason is that NNS can visit nodes that Alg. 3 does not visit. For instance, the tree node for the nearest neighbor of a query point does not necessarily cover the query point. Hence, a tighter work bound for Alg. 3 can be obtained.

4 THE PARALLEL COVER TREE ALGORITHM

In this section, we discuss the parallel cover tree algorithms. We note that the queries on cover trees are already parallel since they do not change the data structure, so multiple queries can directly be applied simultaneously. Now we will introduce our algorithms for batch insertion and batch deletion that are work-efficient and have polylogarithmic span, assuming constant expansion rate and bounded aspect ratio.

4.1 The Batch-Insertion Algorithm

The challenge of a parallel insertion algorithm is illustrated in Fig. 2. In short, we want to identify all potential point pairs that would violate the separation property if we inserted them independently, while maintaining the work-efficiency. For instance, checking all point pairs in a batch of size m will lead to $O(m^2)$ work, which is suboptimal when $m = \omega(\log n)$ since the sequential insertion takes $O(m \log n)$ work. Hence, we need two key components in our insertion algorithm (Alg. 4)—one is the maximal independent set that enables parallel insertions and resolves the conflicts, and the other is prefix doubling that guarantees work-efficiency.

To tackle the possible conflicts between point pairs, we note the following fact:

Lemma 4.1. *For every point pair p_i and p_j that are both inserted at level i as single insertions and violate the separation property, for p_i's parent p_f, we have $d(p_f, p_j) < 3 \cdot 2^i$.*

Proof. Since p_i and p_j violate the separation property, we know $d(p_i, p_j) \leq 2^i$. Since p_i is p_f's child, $d(p_i, p_f) < 2^{i+1}$. Combining with the triangle inequality, we have $d(p_f, p_j) \leq d(p_i, p_j) + d(p_i, p_f) < 3 \cdot 2^i$. □

Therefore, our insertion algorithm identifies these conflict pairs by first using Alg. 3 on Line 7 to traverse the existing tree for each point $p_i \in S$ and record the tree node q_j with $d(p_i, q_j) < 2^{k+1}$

Notation	Definition
General:	
T	The original cover tree
S	The batch that contains points to be inserted or deleted
p_i	Referring to a point in S
q_i	Referring to an tree node in T
Specific for batch-insert:	
S_i	The i-th inserted batch based on prefix doubling
$\bar{S}_i$	$T \cup S_0 \cup \cdots \cup S_i$
P_i	The parent tree node $p_i \in S$ should be inserted if done sequentially in isolation
l_i	The level $p_i \in S$ should be inserted if done sequentially in isolation
L_k	$L_k \leftarrow \{p_i \mid l_i = k\}$
Π_{q_i}	The "conflict set" for tree node $q_i \in T$ that is $\{p_j \in S \mid d(q_i, p_j) < 2^{k+1}\}$, where q_j is at the k-th level
Specific for batch-delete:	
A_i	Point p_i's ancestor at the level being processed
L_k	All tree nodes that should be deleted at level k
X	The current set of uncovered (orphaned) tree nodes due to deletions
Π_{q_i}	The "conflict set" for tree node $q_i \in T$ that is $\{q_j \in X \mid d(q_i, q_j) < 2^{k+1}\}$, where q_j is at the k-th level

Table 1: Notations used in parallel cover trees and analysis.

where q_j is at the k-th level. At the same time, Alg. 3 also computes the level l_i and the parent tree node P_i when a single point $p_i \in S$ is inserted into the cover tree in the sequential algorithm.

Once we have l_i and all pairs (q_j, p_i), we can semisort them and get L_k that contains all points to be inserted at level k (Line 9), and Π_{q_j} that is the set of inserted points covered by tree node q_j with distance 2^{k+1} (Line 8), where k is the level that tree node q_j is in.

After all the preprocessing, we come to the interesting part of this algorithm: inserting all nodes at each level in top-down order. An illustration for this step is given in Fig. 4. Consider we are processing level k now. For each point $p_i \in L_k$ to be inserted, for all tree node q_j at the k-th level, we know $d(p_i, q_j) > 2^k$ since otherwise p_i will be inserted in q_j's subtree in a deeper level. Meanwhile, we know there exists at least one tree node q_j with $d(p_i, q_j) \leq 2^{k+1}$ since otherwise p_i can be inserted at a higher level. Hence, all points in L_k must be in the annuli with distance 2^k and 2^{k+1} centered at tree nodes at level k, as shown in the grey region in Fig. 4.

We then build the graph G to decide the feasible points to be added at level k. We in parallel enumerate each point $p_i \in L_k$ and check all possible conflict pairs (Line 12). As indicated by Lem. 4.1, we only need to enumerate the points in $\Pi_{P_i} \cap L_k$ (P_i is p_i's parent if p_i is inserted as a single point) since Π_{P_i} captures all points p_j with $d(p_i, p_j) < 3 \cdot 2^k$ (Line 13). We then on Line 14 check if $d(p_i, p_j) \leq 2^k$ (violating the separation property), and if so, we add an edge between these two points in G (Line 15). Such an example graph is shown on the top-right in Fig. 4.

Once the graph G is constructed, we will run a maximal independent set (MIS) algorithm on G and let I to be output that contains

Algorithm 4: BatchInsert(T, S).

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

1  Randomly shuffle points in $S$ and partition them into groups
     of $S_0, S_1, \dots S_{\log|S|-1}$, s.t. $|S_0| = 1$ and $|S_i| = 2^{i-1}$ for $i > 0$
2  for $i \leftarrow 0$ to $\log|S| - 1$ do
3      BatchInsertHelper($T$, $S_i$)
4  Function BatchInsertHelper($T$, $S$)
5      Let $C \leftarrow \emptyset$ be a set of pairs of $(q_j, p_i)$.
6      parallel foreach $p_i \in S$ do
7          Run Traverse($T$, $p_i$). For each node $q_j \in T$ with
             $d(p_i, q_j) < 2^{k+1}$ where $q_j$ is at the $k$-th level, add
             the pair $(q_j, p_i)$ to $C$. Set $P_i$ and $l_i$ to the node that
             $p_i$ sould be a child of and $l_i$ to the level $p_i$ should
             be inserted if done sequentially in isolation.
8      Semisort $C$ by $q_j$ and set $\Pi_{q_j} \leftarrow \{p_i \mid (q_j, p_i) \in C\}$
9      Semisort nodes in $S$ based on $l_i$, and let
         $L_k \leftarrow \{p_i \mid l_i = k\}$
10     for $k$ from the root level to leaf level do
11         Initialize a graph $G$ as $(L_k, \emptyset)$
12         parallel foreach $p_i \in L_k$ do
13             parallel foreach $p_j \in \Pi_{P_i} \cap L_k$ do
14                 if $d(p_i, p_j) \leq 2^k$ then
15                     Create an edge between $p_i$ and $p_j$
16         Compute the MIS of $G$ and let $I$ be the selected
             vertices
17         Insert the point $p_i \in I$ to the cover tree based on $P_i$
             from level $k$ to level 0 (leaf level)
18         parallel foreach $p_i \in I$ do
19             parallel foreach $p_j \in \Pi_{P_i}$ do
20                 Let $k' = \lceil \log_2 d(p_i, p_j) \rceil$
21                 if $k' - 1 < k$ then
22                     for $\bar{k}$ from $k' - 1$ to $k - 1$ do
23                         Add $p_j$ to $\Pi_{q_j}$, where $q_j$ is a node at
                             level $\bar{k}$ corresponding to $p_i$
24                 if $k' < l_j$ then
25                     $P_j \leftarrow$ the tree node for $p_i$ at level $k'$
26                     Remove $p_j$ from $L_{l_j}$
27                     Add $p_j$ to $L_{k'}$
28                     $l_j \leftarrow k'$
```

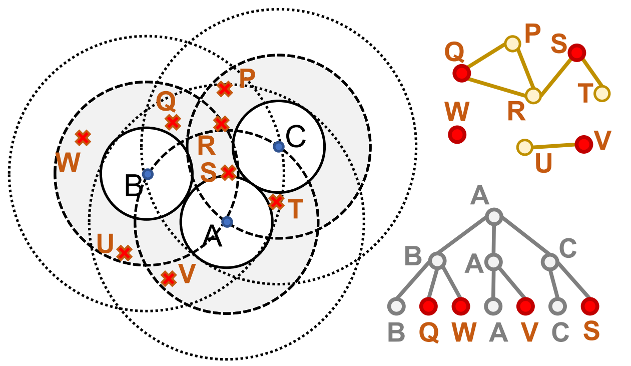

Figure 4: The parallel insertion process in a certain level. In this example, points A, B, and C have already been inserted and are siblings in this level, and the other points are in the inserted batch. Here the solid circles identify the separating distance d, long dash circles are the covering distance $2d$, and the short dash circles indicate the distance of $3d$. All inserted nodes in this level must be in the annuli marked in gray (otherwise they either will not be covered by A, B, or C, or will go to lower levels). We check all point pairs in each short dash circle with distance $3d$, and add an edge if their distance is no more than d. A graph on the top-right corresponds to the points P to W. We run an MIS on this graph, and assume points Q, S, V, and W are selected. These four nodes will be inserted to the tree as shown on the bottom-right, and other points P, R, T, and U will be distributed to either of their selected neighbors, and wait to be inserted in the next round.

the selected vertices (Line 16). In Fig. 4, $I = \{Q, S, V, W\}$. All points in I can be inserted into the cover tree at level k (Line 17), while at least one of its selected neighbors will cover every other point.

After we build the tree nodes for the points in I, we have two more tasks. For each selected vertex in I, we generate the conflict sets for all inserted tree nodes corresponding to this point, executed on Line 23 based on the definition of the conflict sets. It is easy to see that all points in these conflict sets generated by the point p_i must be in Π_{P_i}. The second task is that we need to check if the newly inserted points invalidate the current insert positions P_j of some uninserted points p_j in S. Consider a point p_j that is very close to a newly inserted point p_i (say $d(p_i, p_j) = 1$). In this case, the original position for p_j can violate the separation property if it is not in the leaf level. We need to check this and adjust it to a valid insertion position if needed (on Line 25), and we refer to this as the redistribution step. We will later show that the work for these two steps are bounded by the cost to construct the graph G and the sequential insertion cost.

We will start by showing the correctness of this algorithm.

Lemma 4.2. *All possible conflict point pairs (i.e., violating separation) are captured by* Π_{P_i} *on Line 13.*

Proof. Let p_i be a point we are inserting and let p_j be any other point we are inserting, and let P_i and P_j be the nodes that would be the parents of p_i and p_j respectively if we inserted each by itself. Suppose two inserted node p_i and p_j under P_i and P_j violate the separation property at some level l and $d(p_i, p_j) \leq 2^l$. Let l_i be the level of P_i, so $l_i > l$. Then $d(P_i, p_j) \leq d(P_i, p_i) + d(p_i, p_j) \leq 2^{l_i} + 2^l \leq 2^{l_i} + 2^{l_i - 1} = 3 \cdot 2^{l_i - 1} < 2^{l_i + 1}$. Hence, p_j is also in Π_{P_i}. □

We then inductively show that by the end of an iteration of Line 10, T is valid a cover tree (e.g., all Π_{q_i} and P_i for each uninserted point in S satisfy the definitions in Tab. 1 for the T), then T is also valid after running another iteration of this loop. In these lemmas, T refers to the tree at the end of this iteration, and T' refers to the tree at the end of the previous iteration.

Lemma 4.3. *At end of an iteration of Line 10, T is a valid cover tree.*

Proof. First, we show that the covering property is satisfied. Consider a point p_i that is in T at the end of this round. If it is already in T', then the covering property is satisfied since neither its level or parent were changed. Otherwise, we inserted p_i in this round under P_i. By definition, P_i covers p_i, so the covering property is satisfied.

Next we show that the separation property is satisfied for each pair of nodes in each level. Consider two tree nodes q_i and q_j in the same level in T. If they are already in T', then the separation property is already satisfied. If $q_i \notin T'$ and $q_j \in T'$, then the separation property is satisfied, because if we inserted q_i into T' in the sequential algorithm, it would insert into the location separated from q_j, since otherwise we either identify this when running the traverse algorithm on Line 7, or q_i will go to q_j's subtree on Line 23 in the round that the corresponding point of q_j is inserted. The separation property is also satisfied when neither q_i nor q_j are in T', because the MIS selected I to be a set of points such that no two points are closer than 2^k.

The covering and separation properties are satisfied, so T is a valid cover tree. □

Lemma 4.4. *At end of an iteration of Line 10, each Π_{q_i} is correct.*

Proof. For every newly inserted node q_i, on Line 23 we set $\Pi_{q_i} = \{p_j \in \Pi_{P_i} \mid d(p_i, p_j) \leq 2^{k'+1}\}$, where k' is q_i's level. By definition, $\Pi_{q_i} \subseteq \{p_j \in S \mid d(p_i, p_j) \leq 2^{k'+1}\}$. Since $\Pi_{P_i} \subseteq S$, we have already checked all points in Π_{P_i}. Let $p_j \in S \setminus \Pi_{P_i}$. From the definition of a conflict set, $d(p_j, P_i) > 2^{k+2}$. Using the triangle inequality, $d(p_j, p_i) \geq d(P_i, p_j) - d(P_i, p_i) > 2^{k+2} - 2^{k+1} = 2^{k+1} > 2^{k'+1}$. Therefore $\{p_j \in S \mid d(p_i, p_j) \leq 2^{k'+1}\} \subseteq \Pi_{q_i}$ and Π_{q_i} is correct. □

Lemma 4.5. *At end of an iteration of Line 10, P_j and l_j for each uninserted point p_j is correct.*

Proof. Let P'_j and l'_j be the values of P_j and l_j at the end of the previous iteration. First note that, if $l_j \leq l'_j$, the separation property is satisfied between p_j's nodes and all nodes in T'.

If $p_j \notin \Pi_{P_i}$ for any $p_i \in I$, then we leave $P_j = P'_j$ and $l_j = l'_j$. $p_j \notin \Pi_{P_i}$ means $d(p_j, P_i) > 2^{k+2}$. By the triangle inequality, $d(p_j, p_i) \geq d(p_i, P_i) - d(p_j, P_i) > 2^{k+2} - 2^{k+1} = 2^{k+1}$. $2^{k+1} > 2^{l_j}$ since $l_j \leq k$, so p_j is separated from all nodes for all p_i. $l_j \leq l'_j$, so it is separated from all nodes in T'. Lastly $P_j = P'_j$ trivially satisfies the covering property.

If $p_j \in \Pi_{P_i}$ for any $p_i \in I$, then either $d(p_j, p_i) \leq 2^{l'_j}$ for some $p_i \in I$ or $d(p_j, p_i) > 2^{l'_j}$ for all $p_i \in I$. Each of the two cases are discussed in the next two paragraphs.

If $d(p_j, p_i) > 2^{l'_j}$ for all $p_i \in I$, we leave $P_j = P'_j$ and $l_j = l'_j$. $d(p_j, p_i) > 2^{l'_j} = 2^{l_j}$ means separation is satisfied between all nodes for p_j and all nodes for all $p_i \in I$. The remaining nodes in T are the nodes that were already in T', and because $l_j \leq l'_j$, all nodes for p_j are separated from all nodes in T'. P'_j satisfied the covering property, so P_j satisfies the covering property. Therefore P_j and l_j are correct.

If $d(p_j, p_i) \leq 2^{l'_j}$ for some $p_i \in I$, then we pick $l_j = \lceil \log_2 d(p_j, p_i) \rceil$ and set P_j to p_i's node at level $l_j + 1$. This means $2^{l_j} < d(p_j, p_i) \leq 2^{l_j+1}$, so P_j covers p_j and that p_j is separated from all nodes for p_i. For all $p_h \in I \setminus \{p_i\}$, the triangle inequality gives $d(p_j, p_h) \geq d(p_i, p_h) - d(p_j, p_i) \geq 2^k - 2^{k-1} = 2^{k-1} > 2^{l_j}$, so separation is satisifed with all the other newly inserted nodes. Lastly, $l_j \leq l'_j$, so separation is satisfied with all nodes in T'. Therefore P_j and l_j are correct. □

Lemma 4.6. *The new cover tree is a valid cover tree.*

Proof. This follows by induction using Lem. 4.3 to 4.5 and by assuming the correctness of the traverse algorithm. At the start of the algorithm, T is given as a valid cover tree. The loop around Line 7 computes each P_i and Π_{q_i} correctly, so the invariant holds at the beginning of the first iteration. By induction on the previous three lemmas, the invariant holds at the beginning and end of any iteration. Nothing happens after the last iteration, so T is a valid cover tree at the end of the algorithm. □

However, just doing the above-mentioned steps does not guarantee work-efficiency. Considering a tree T with just one root node and $|S| = n$. Then for some tree node p_i in level k cover set $\Pi_{p_i} \cap L_k$ may contain $O(n)$ nodes, and checking them pairwise on Line 14 incurs $O(n^2)$ work. To reduce the work in these incremental construction algorithms, a common approach is prefix doubling [14–16, 59]. Here we can do the same: partition the batch S into $\log_2 |S|$ groups, and insert each group in turn. This guarantees that the current cover tree always has at least the same number of (randomly chosen) points as the group to insert. As a result, each of the cover set ($\Pi_{p_i} \cap L_k$ on Line 13) has limited size: constant on average and logarithmic *whp*.

Lemma 4.7. *For each point p, the number of p's neighbors in G (Line 16) is $O(1)$ in expectation and $O(\log n)$* whp.

Proof. Now consider all neighbor points of p, and denote them as $N(p)$. They must all be in the current batch S_i (Line 3) to be inserted, and $N(p) \in B_{S_i}(p, 2^k)$ (k is the current level). Note that not all points in $B_{S_i}(p, 2^k)$ are in $N(p)$ since the points inserted in the previous $i-1$ batches may "capture" some of the points in $B_{S_i}(p, 2^k)$ (Line 25) so they will be inserted to lower levels. This will only help reducing the neighbor set size (i.e., making $N(p)$ smaller).

We now analyze the size of $N(p)$ in expectation and with high probability. Let $\bar{S}_i = S_0 \cup \cdots \cup S_i$, and we have $B_{S_i}(p, 2^k) \subseteq B_{\bar{S}_i}(p, 2^k)$. Let $N'(p) \subseteq B_{\bar{S}_i}(p, 2^k)$ be the set of points that are not captured by points in the original tree T (excluding points in $\bar{S}_i$ with parents at or below the k-th level). The argument is that if any of the points in $N'(p)$ are in the previous $i-1$ batches (i.e., $N'(p) \neq N(p)$), then p cannot be inserted at level k, but lower than that. This is based on the separation property of the cover tree, and indicates $N'(p) = N(p)$ if p is processed at the k-th level. Given how S_i is sampled, we show that $|N(p)|$ is small. Each point in $N'(p)$ has at most half the probability of being in S_i. We first consider the high probability guarantee for $s = |N(p)|$. For each point $q \in N'(p)$, $\Pr[q \in N(p)] \leq 1/2$. If $s > c \log n$, then

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

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 $m \le n$.*

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.

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
```

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 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],

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 $T$
Construct the cover tree $\mathcal{D}$ on $P$
$S \leftarrow \{\{p_1\}, \{p_2\}, \ldots, \{p_n\}\}$
$T \leftarrow \emptyset$
while $|S| > 1$ do
    parallel foreach $C_i \in S$ do
        $\langle p_i, q_i \rangle \leftarrow$ Cluster-Query($C_i$)
    Let $T' = \cup_i \{\langle p_i, q_i \rangle\}$ and $T \leftarrow T \cup T'$
    Merge the clusters using the tree edges in $T'$ and update $S$
return $T$

function Cluster-Query($C$)
    $\mathcal{D}' \leftarrow \mathcal{D}$.B-Delete($C$)
    parallel foreach $p_i \in C$ do
        $q_i \leftarrow \mathcal{D}'$.Query($p_i$)
        $d_i \leftarrow d(q_i, p_i)$
    $i^* = \arg\min_i d_i$ // Using a parallel reduce
    return $\langle p_{i^*}, q_{i^*} \rangle$
```

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.

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 $|B(p,\epsilon)| \ge \mathsf{minPts}$. 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 $\mathsf{minPts}$is a constant, and in practice, we usually pick $\mathsf{minPts} = 10$.

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.