Parallel kd-tree with Batch Updates

Ziyang Men
zmen002@ucr.edu
UC Riverside

Zheqi Shen
zshen055@ucr.edu
UC Riverside

Yan Gu
ygu@cs.ucr.edu
UC Riverside

Yihan Sun
yihans@cs.ucr.edu
UC Riverside

Abstract

In this paper, we present Pkd-tree, a parallel kd-tree data structure that is efficient in both theory and practice. The kd-tree is one of the most widely-used data structure to manage multi-dimensional data. However, we observed that existing parallel implementations have challenges in supporting efficient updates and/or queries with high parallelism. Such difficulty is cause by two aspects. First, the special structure for kd-trees does not support simple rebalance primitives such as rotations. Thus, enabling dynamic updates on kd-trees can be challenging. Second, the design space of kd-trees requires a co-design of construction, update, and queries. Some existing approaches incorporate parallel updates by sacrificing query performance. To make the data structure practical and flexible, it is essential to optimize construction, updates, and queries cohesively.

We present Pkd-tree (Parallel kd-tree), which supports parallel tree construction, batch update (insertion and deletion), and various queries including k-nearest neighbor search, range query, and range count. We proved that our algorithms have strong theoretical bound in work (sequential time complexity), span (parallelism), and I/O cost. Our key techniques include 1) reconstruction-based update algorithms that guarantees the tree to be weight-balanced, and 2) an efficient construction algorithm that optimizes work, span, and I/O simultaneously. Our reconstruction-based algorithms use the construction algorithm as a major building block, so our new techniques apply for both construction and updates. With a combination of new algorithmic insights and careful engineering effort, we achieved a highly optimized implementation of Pkd-tree.

We tested Pkd-tree with various synthetic and real-world input distributions, including both uniform and highly skewed data. We compare Pkd-tree with state-of-the-art parallel kd-tree implementations. On synthetic and real-world data, with competitive or better query performance, Pkd-tree is always the fastest in construction and updates in all cases among all kd-tree implementations.

1 Introduction

The kd-tree is one of the most widely-used data structure for managing multi-dimensional data. kd-trees maintains a set of points in D dimensions[1], and supports a variety of queries including k-nearest neighbor (k-NN), orthogonal range count and range search, etc. Compared to other counterparts, the kd-tree has its unique advantages, such as linear space, theoretically-efficient construction, simple algorithmic ideas, being comparison-based (and thus resistant to skewed data), scales to reasonably-large dimensions (being efficient up to $D \approx 10$) and supporting a wide range of query types. Due to these advantages, kd-tree is usually the choice of data structure in many applications. Indeed, after its invention by Bentley in 1975 [8], kd-tree has been widely used in real-world applications and cited by over ten thousand times across multiple areas including databases [20, 34, 38, 49], data science [26, 53, 69, 74], machine learning [24, 46, 47, 62], clustering algorithms [45, 48, 52, 60, 65], computational geometry [18, 37, 50, 66], etc.

Due to the ever-growing data volume, it is imperative to consider parallelism in kd-trees. However, we observe a *significant gap* between the *wide usage* of kd-trees, and a *lack of efficient parallel implementation* to support high-performance kd-trees. The two parallel libraries we are aware of, CGAL [29] and ParGeo [71], both have difficulties to scale to today's large-scale data size. CGAL does not support parallel updates. Even for its sequential point update, it fully rebuilds the tree for rebalancing, which is inefficient. ParGeo supports parallel update, and avoids fully rebuilding the tree upon update by using *logarithmic method*, i.e., it maintains $O(\log n)$ perfectly balanced trees with different sizes, and an update reorganizes the trees by merging some of them in parallel. Accordingly, a query processes all $O(\log n)$ trees and combine the results, which can be significantly more expensive than a single kd-tree. As shown in Table 1, for dataset with 100M 2-dimensional points, it is about 17–23× slower in k-NN queries and 2.5× slower in range queries. Due to these issues, neither CGAL nor ParGeo outperforms Zd-tree [13], a recent parallel quad/octree structure based on space-filling curves: CGAL is 99–1511× slower in update, and ParGeo is 11–14× slower in queries. However, due to the essence of quad/octree and space-filling curves, Zd-tree has its own limitations compared to kd-trees, such as inefficiency for $D > 3$ dimensions, and the difficulty to deal with non-integer coordinates or skewed data distributions.

In this paper, we provide ***high-performance parallel implementation for kd-trees that supports efficient construction, updates, and queries.*** To do this, we have to deal with several inherent challenges, which are also what make state-of-the-art solutions fall short of achieving high performance.

The first important challenge is to support efficient dynamic updates on kd-trees. kd-trees differ from other classic trees and do not support rebalance primitives for updates, such as overflow/underflow (as in B-trees), and rotations (as in binary search trees). Sequentially, a few solutions have been proposed (see Sec. 8), but it is not clear how they incorporate parallelism. The recent attempt in ParGeo adopted the logarithmic method with strong bounds, but incurs significant overhead in query. It is therefore worth asking 1) what is the best way to achieve dynamism for kd-trees in parallel, and 2) how to achieve a tradeoff to give overall best performance

[1]Based on the original terminology, kd-tree deals with k-dimensional data. To avoid overloading k in different scenarios such as the "kNN" query (i.e., finding k nearest neighbors of a given point), we use D as the number of dimensions in this paper.

and theoretical bounds in both updates and queries.

The second challenge comes from the inherent complication of the *design space* for kd-trees. As a classic data structure, kd-tree is widely-used in multiple applications with different design goals. The performance evaluation is thus monolithic, encompassing construction, updates, and a *variety* of queries. To achieve the best performance, one also have to consider factors such as work-efficiency (i.e., low time complexity), I/O-efficiency, and parallelism. Most existing work focused on one or a subset of such goals, and thus overlooked the other factors and did not incorporate them in their implementation collectively. A good implementation should require co-design of all these factors, but it is highly challenging to do so. It is therefore worth asking 1) how to incorporate all these factors (work, parallelism, I/O) to design a full-featured interface (construction, update, queries) of kd-trees, and 2) how much each factor affects the performance.

In this paper, we answer these questions by proposing ***Pkd-tree (Parallel kd-tree)***, a parallel kd-tree implementation supporting efficient construction, batch-update and multiple query types. This requires both new algorithmic insights to maintain good theoretical guarantees, and careful engineering effort to achieve high performance. One key question we investigated is, whether the perfect balancing is needed in kd-tree (i.e., $\log_2 n + O(1)$ tree height), or a reasonable relaxed criterion is good enough (e.g., $O(\log n)$ tree height). We conducted experimental studies and showed that the kd-tree's query performance is reasonably consistent with minor imbalance (see Sec. 7.4). Motivated by this observation, we adopted a *randomized weight-balanced* approach, and such relaxation allows for much higher performance in *construction, update and query* compared to state-of-the-art implementations. We also provide careful *theoretical analysis* to guarantee that under our relaxation, the algorithms are *efficient in work* (i.e., low time complexity), *span* (i.e., good parallelism)[2], and *I/O cost*.

To enable efficient parallel kd-tree algorithms both in theory and in practice, Pkd-tree is a careful co-design of construction and update algorithms. For construction, to reduce the I/O cost to find splitting hyperplane and move all data points per level, Pkd-tree employs a careful ***sampling scheme*** to determine the hyperplanes for λ levels, and applies an efficient parallel algorithm to ***sieve all points down λ levels by one round of data movement***. In experiments, we observe that sampling and multi-level construction improve performance by 1.6× and 2.2×, respectively.

For batch updates (insertion and deletion), we design a ***lazy strategy*** that tolerates the difference of sibling subtree sizes by a controllable factor of α before invoking the rebalancing scheme, which identifies the unbalanced substructures and performs a ***local reconstruction*** on the affected subtrees. The parameter α controls the level of balance, enabling a tradeoff between update and query costs—a stronger balance condition allows for better query performance (due to shallower tree height), at the cost of more expensive updates (due to more frequent rebalancing). We experimentally show that even a reasonably large α only mildly affects the query performance, while leading to much better update time.

While the high-level ideas of sampling [1, 5, 10, 11], multi-level construction [1, 58] and reconstruction-based balancing scheme [22, 32, 44, 54] have all been studied in existing work (for both kd-trees and other data structures), the unique challenge here is to integrate these elements cohesively in theory, in order to facilitate a unified, monolithic implementation for kd-tree. Indeed, we are not aware of any existing work on kd-trees that are theoretically-efficient in all measurements of work, span and I/O for either construction ***or*** update. We present a more detailed description of the existing work in Sec. 8. Such a synergy of construction and update algorithms is more important in Pkd-tree, since its update relies on efficient reconstruction. Our design achieves this by 1) designing the sieving step as a building block by borrowing ideas from recent parallel sorting algorithms [28], which provides work, span and I/O bounds for both construction and updates, and 2) configuring the parameters for sampling, multi-level construction and rebalancing collectively. In theory, we present the parameter configuration in Lem. 3.1 and Lem. 3.2, and analyze the tradeoff (due to relaxed balancing) in the cost bounds of update and query in Thm. 4.1 and Sec. 5, respectively. In practice, we provide the highly-optimized implementation that support parallel construction, update, and three query types: k-nearest query (k-NN), range count and range search. We also carefully surveyed existing optimizations in previous theoretical and experimental work, across different settings (sequential, parallel, distributed), and apply what we believe are the most relevant ones to Pkd-trees (overviewed in Sec. 6).

With the new proposed algorithms, as illustrated in Table 1, Pkd-tree is faster than the best parallel kd-tree baseline by 2.1× on construction, 3.3× on update, 1.6× on k-NN query and 1.5× on range query. For high-dimension space and large-scale data, Pkd-tree shows better performance (see Table 3 as a summary of the results). We also design experiments to understand the performance gain of the proposed techniques in both construction and updates, discussed in Sec. 7.3. Pkd-tree can process very-large scale data. It constructs one billion points in 3D in about 4 seconds, and in 9D in about 10s. Even in 3D, none of the baseline kd-trees were able to perform all point 10-NN query on 1 billion points due to space- and time-inefficiency, while Pkd-tree only takes 11 seconds. We believe Pkd-tree is the first parallel dynamic kd-tree implementation that scales to billions of points with high performance.

Our contributions include:

- Algorithms and implementations for parallel kdtree construction, updates, queries;
- Parallel library (open-source) supporting full interface for kd-tree with high performance; and
- In-depth experimental study on parallel kd-trees.

2 Preliminaries

We present a table of notations used in this paper in Table 2. We use ***with high probability*** (*whp*) in terms of n to mean probability at least $1 - n^{-c}$ for any constant $c > 0$. With clear context, we omit "in terms of n". We use $\log n$ as a short term of $\log_2 n$.

Computational Model. We analyze our algorithm using the ***work-span model*** in classic fork-join model with ***binary-forking*** [6, 15, 19]. We assume multiple threads that share memory. Each thread operates as a sequential RAM augmented with a fork instruction, which spawns two child threads that run in parallel. The parent

[2]The *work* of a parallel algorithm is the total number of operations (i.e., sequential time complexity), and its *span* is the longest dependence chain. They are formally defined in Sec. 2.

	Uniform					Varden				
Operation	**Ours**	**Zd-tree**	**Log-tree**	**BHL-tree**	**CGAL**	**Ours**	**Zd-tree**	**Log-tree**	**BHL-tree**	**CGAL**
Build (Dataset Size = 100M)	<u>.246</u>	.523	3.92	3.24	98.8	<u>.255</u>	.501	3.71	2.96	40.6
Batch Insert (Batch Size = 10M)	<u>.044</u>	.145	1.09	3.30	119	<u>.050</u>	.059	.980	2.99	89.2
Batch Delete (Batch Size = 10M)	<u>.050</u>	.128	.240	2.83	28.5	.076	<u>.056</u>	.310	2.66	5.88
10 Nearest Neighbor for All Points	<u>1.05</u>	1.68	24.2	3.93	7.99	<u>1.04</u>	1.62	17.9	2.25	3.68
Range Query (10^4 queries)	<u>.145</u>	n.a.	.358	.220	127	<u>.143</u>	n.a.	.359	.220	136

Table 1: Running time (in seconds) for Pkd-tree and other baselines on 100M points in 2D. "Zd-tree" is a parallel quad/oct tree in [13], "Log-tree" is a parallel kd-tree using logarithmic method from [71], "BHL-tree" is a single kd-tree used in Log-tree [71]. "CGAL" is a widely-used geometry library [71]. More details are given in Sec. 7. The output size for "Range query" is controlled in range of 10^4–10^8. The best performance for each test is underlined.

thread resumes execution upon the completion of both child threads. We can use a logarithmic number of steps of forking to simulate a parallel for-loop. The execution of a computation task can be visualized as a directed acyclic graph (DAG). The ***work*** (W) of a parallel algorithm is the total number of operation within its DAG, and the ***span (depth)*** (S) depicts the longest path in the DAG. Using a randomized work-stealing scheduler, a computation with work W and span S can be executed in $W/P+O(S)$ time *whp* (in W) with P processors [6, 19, 35].

We use the classical I/O model [4, 31] to measure the cost for memory access. In the I/O model, the memory is divided into two hierarchy levels. The CPU is connected to the small-memory (a.k.a. the cache) of finite size M, which is connected to a large-memory (the main memory) with infinite size. Both memory are organized as line of blocks and each block has size B, thus in total M/B cachelines in the small-memory. CPU can only access the data in small-memory with free cost, and there is a unit cost to transfer data from large-memory to small-memory, assuming the optimal offline cache replacement policy. The I/O cost of an algorithm is the measure of the cost for data transfer during execution of the algorithm.

kd-Trees. We study points in D-dimension Euclidean space, and the distance between two points is their Euclidean distance. Given point p, we refer $p[i]$ $(1 \le i \le D)$ to the value of i-th dimension of p. A partition hyperplane can be represented by a pair $\langle d, x\rangle$, where d $(1 \le d \le D)$ is the ***splitting dimension*** and $x \in \mathbb{R}$ is the ***splitting coordinate***. We refer to such a pair s as a ***splitter***.

The ***kd-tree***, or the k-dimensional tree, is a spatial-partitioning data structure based on a binary tree. To avoid overloading the commonly-used parameter k in k-NN query, we use D to refer to the number of dimensions of the dataset. Each interior (non-leaf) node in a kd-tree signifies an axis-aligned splitting hyperplane $\langle d, x\rangle$. Points to the left of the hyperplane are stored in the left subtree and the remaining are in the right subtree. Each subtree is split recursively until the number of points drops below a given parameter ϕ (a small constant), and such nodes are leaf nodes and directly store the points they contain. Common approaches for choosing the dimension of the splitting hyperplane include cycling among the D dimensions, choosing as the widest dimension, etc. The cutting coordinate x is usually the median of the points on the d-th dimension, yielding two balanced subtrees. Given n points in D dimensions, a balanced kd-tree has a height of $\log_2 n + O(1)$ and can be constructed in $O(n \log n)$ work using $O(n)$ space.

kd-trees are versatile in answering different types of queries such as k-NN queries and range-based queries. In this paper, we mainly focus on *k-NN queries* (finding the k nearest points to a query point in the kd-tree), rectangle *range queries* (reporting all points within an axis-parallel bounding box) and rectangle *range count queries* (reporting the number of points within an axis-parallel bounding box).

$\boldsymbol{T}$	a (sub-)kd-tree, also the set of points in the tree		
$\boldsymbol{\phi}$	upper bound for size of leaf wraps		
$\boldsymbol{k}$	required number of nearest neighbors in a query		
$\boldsymbol{\lambda}$	number of levels in tree sketch (i.e., that are built at a time)		
$\boldsymbol{\mathcal{T}}$	shorthand of $\mathcal{T}_\lambda$, tree skeleton at T with maximum levels λ		
$\boldsymbol{P}$	input point set (for updates, P is the batch to be updated)		
$\boldsymbol{T.lc}$	left child of T	$\boldsymbol{T.rc}$	right child of T
$\boldsymbol{n}$	tree size	$\boldsymbol{m}$	batch size for batch updates
$\boldsymbol{D}$	number of dimensions	$\boldsymbol{d}$	a certain dimension
$\boldsymbol{S}$	samples from P	$\boldsymbol{s}$	size of the S
$\boldsymbol{\sigma}$	oversampling rate	$\boldsymbol{\alpha}$	balancing parameter
$\boldsymbol{M}$	small-memory (cache) size	$\boldsymbol{B}$	cacheline size

Table 2: Notations used in this paper.

We use the (subtree) root pointer T to denote a (sub-)kd-tree. With clear context, we also use T to represent the set of all points in T. Every interior node in T maintains two pointers $T.lc$ and $T.rc$ to its left and right children respectively. As we mentioned, Pkd-tree is *weight-balanced.* Given the balancing parameter $\alpha \in [0, 0.5]$, we say a kd-tree is (weight-)balanced if $0.5 - \alpha \le |T.lc|/|T| \le 0.5 + \alpha$, and *imbalanced* otherwise. Essentially, this means that the two subtrees can be off from perfectly balanced by a factor of α.

3 Our Tree Construction Algorithm

We start with the parallel kd-tree construction algorithm, which is also a building block in our parallel batch-update algorithm. Constructing a kd-tree means to distribute the points into nested sub-spaces recursively. Sequentially, once the splitting dimension is decided, we can find the median of all points and partition them into the left and right subtrees. The two subtrees will be processed recursively until the subproblem size is within the leaf-wrap threshold ϕ. It directly implies a parallel construction algorithm with $O(n \log n)$ work and $O(\log^2 n)$ span using the standard parallel partition algorithm ($O(n)$ work and $O(\log n)$ span). However, it requires $O(\log n)$ rounds of data movement and is not I/O-efficient.

Instead of partitioning all points into the left and right subtrees and pushing the points to the next level in the recursive calls, the high-level idea of our approach is to build λ levels at a time by one round of data movement. To avoid the data movement for finding the splitting coordinates, our algorithm also uses samples to decide all splitters for λ levels. We then distribute all points into the corresponding subtrees (2^λ of them) and recurse.

The main challenge here are 1) to move each point exactly once to their final destination in an I/O-efficient and parallel manner, and 2) to use only a subset of the points (samples) to decide the splitters and make the tree nearly balanced. We will elaborate on our approach for parallel and I/O-efficient kd-tree construction in Sec. 3.1. Then in Sec. 3.2 we show the analysis to guarantee the tree height and cost bounds under our sampling scheme.

3.1 Parallel and I/O-efficient kd-tree Construction

We present our algorithm in Alg. 1 and an illustration in Figure 1. The algorithm $T = \text{BuildTree}(P)$ returns a kd-tree T on the input points in array P. As mentioned, our main idea is to use samples to decide all splitters in λ levels. We define the *skeleton* at T as the substructure consisting of all splitters (and thus interior nodes) in the first λ levels, and denote it as $\mathcal{T}_\lambda$. We use the samples to build the skeleton. In particular, we will uniformly take $2^\lambda \cdot \sigma$ samples from P, where σ is the *over sampling rate*. In Sec. 3.2 we will show how to choose the parameter σ to achieve good theoretical guarantee. Let S be the set of sample points, we will build the skeleton as a kd-tree on S. As we will show in Sec. 3.2, we will keep S small and fit in cache, so that the skeleton can be built by the naïve parallel kd-tree construction algorithm at the beginning of Sec. 3.

The skeleton depicts the first λ levels of the tree, and splits the space into 2^λ subspaces, corresponding to the external nodes (leaves) of the skeleton. We call each such external node a ***bucket*** in this skeleton. The problem then boils down to sieving the points into the corresponding bucket, so that we can further deal with each bucket recursively in parallel. We label all buckets from 0 to $2^\lambda - 1$. We first note that the target bucket for each point can be easily looked up in $O(\lambda)$ cost by searching in the skeleton. Sequentially, one can simply move all points one by one to their target bucket, by maintaining a pointer to the (current) last element in each bucket. In parallel, the key challenge is to independently determine the "offset" of each point, so the points can be moved to their target buckets in parallel without introducing locks or data races.

We borrow the idea from I/O-efficient parallel sorting algorithm [28], which also involves redistributing elements into $\omega(1)$ buckets. Our goal is to reorder array P and make all points in the same bucket to be contiguous, so that next recursion receives consecutive input slice. At a high-level, this algorithm divides the array into chunks of size l, and process them in parallel. We will first count the number of elements in chunk i that falls into bucket j in $A[i][j]$. Note that there is no data race since within each bucket we count all points sequentially. Then we compute the exclusive prefix sum of matrix A in column major and get matrix B. This can be done in parallel and I/O-efficiently by an I/O-efficient matrix transpose [16] and a standard parallel prefix-sum [36]. As such, $B[i][j]$ implies the offset when writing a point in chunk i that belongs to bucket j. We present an illustration for this process in Figure 1. Then we process all buckets again in parallel, and move each point to their final destination by using the offsets provided in B as the starting pointer. There is still no data race here, since all points that "share" the same offset must be in the same chunk. They will be processed sequentially.

After all points in the same bucket are placed consecutively, we can recursively build kd-trees for each bucket in parallel. Such recursion stops when the number of points is a smaller than the leaf wrap ϕ (32 in our implementation), where we construct a leaf node with a flat array for all points and return.

Algorithm 1: Parallel kd-tree construction

```
Input: A sequence of points $P$.
Output: A $k$d-tree $T$ on points in $P$.
Parameter: $\lambda$: the height of a tree skeleton.
           $\phi$: the leaf wrap of the $k$d-tree.
Function BuildTree($P$)
    $n \leftarrow |P|$                                   // Size of the input points
    if $|P| < \phi$ then return A leaf node containing $P$        // Leaf wrap
    $S \leftarrow$ Uniformly sample $2^\lambda \cdot \sigma$ points on $P$ with replacement
    Build tree skeleton $\mathcal{T}$ by constructing the first $\lambda$ levels of a $k$d-tree on $S$
    // Sieve each point to their corresponding bucket (external node) in $\mathcal{T}$.
       This is performed by reordering all points in $P$ to make points in the
       same bucket consecutive.
    $R[\,] \leftarrow$ Sieve($P, \mathcal{T}$)  // $R[i]$: the sequence slice for all points in bucket $i$
    parallel-foreach external node $i$ do
        $t \leftarrow$ BuildTree($R[i]$)        // Recursively build a $k$d-tree on $R[i]$
        Replace the external node with $t$
    return The root of $\mathcal{T}$
// Sieve points in $P$ to the buckets (external nodes) in $\mathcal{T}$
Function Sieve($P, \mathcal{T}$)
    (Conceptually) divide $P$ evenly into chunks of size $l$
    parallel-foreach chunk $i$ do
        for point $p$ in chunk $i$ do
            $j \leftarrow$ the bucket id for $p$ by looking up $p$ in $\mathcal{T}$
            $A[i][j] \leftarrow A[i][j] + 1$
    Get the column-major prefix sum of $A[i][j]$ as matrix $B$
    parallel-foreach bucket $j$ do
        Let $s_j \leftarrow B[0][j]$ be the offset of bucket $j$
    parallel-foreach chunk $i$ do
        for point $p$ in chunk $i$ do
            $j \leftarrow$ the bucket id for $p$ by looking up $p$ in $\mathcal{T}$
            $P'[B[i][j]] \leftarrow p$
            $B[i][j] \leftarrow B[i][j] + 1$
    Copy $P'$ to $P$
    parallel-foreach bucket $j$ do
        $R[j] \leftarrow$ the slice $P[s_j..s_{j+1} - 1]$  // for the last bucket, $s_{j+1} = |P|$
    return $R[\,]$
```

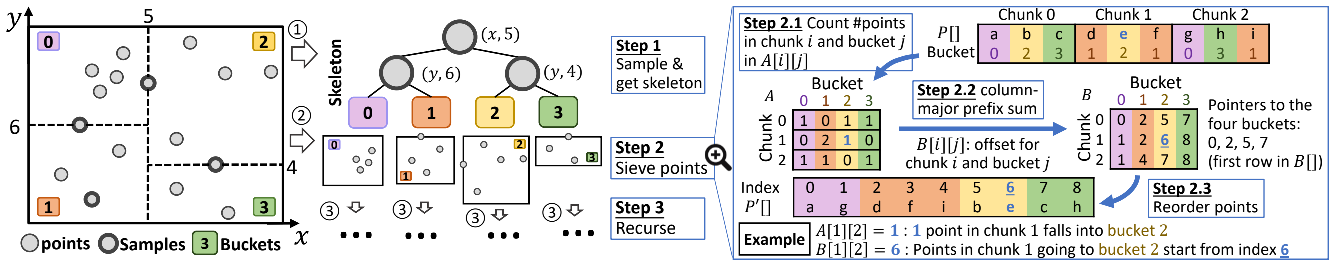
3.2 Theoretical Analysis

We now formally analyze our new construction algorithm and show its theoretical efficiency. We start with a useful lemma about sampling. Similar results about sampling have also been shown previously [1, 17, 57]. We put it here for completeness.

Lemma 3.1. *For a Pkd-tree T with size n', for any $\epsilon < 1$, setting $\sigma = (12c\log n)/\epsilon^2$ guarantees that the size of a child subtree is within the range of $(1/2 \pm \epsilon/4) \cdot n'$ with probability at least $1 - 2/n^c$.*

Here we need to distinguish a subtree size n' and the overall tree size n for a stronger high probability guarantee, so we can apply union bounds in the later analysis.

Proof. In Alg. 1, note that our sampling scheme guarantees that each leaf has at least σ sampled points. Therefore, every time we find a pivot, it is the median of at least 2σ sampled points. Here let s ($\geq 2\sigma$) be the number of samples for this Pkd-tree T, and S' contains the smallest $(1/2 - \epsilon/4) \cdot n'$ points in the cutting dimension. We want to show that the chance we have more than $s/2$ samples in S' (so that the left side has no more than $(1/2 - \epsilon/4) \cdot n'$ points) is small. Since all samples are picked randomly, we denote indicator variable $X_i = 1$ if i-th sample is in S' and 0 otherwise. Let $X = \sum_{i \in S'} X_i$ and $\mu = \mathbb{E}[X] = (1/2 - \epsilon/4)s$. Let $\delta = \epsilon/(2 - \epsilon)$, and we have $(1 + \delta)\mu = (1 + \frac{\epsilon}{2-\epsilon})(\frac{1}{2} - \frac{\epsilon}{4})s = \frac{s}{2}$. Using the form of Chernoff Bound $P[X \geq (1 + \delta)\mu] \leq \exp(-\delta^2\mu/(2 + \delta))$, we have:



y
x
5
6
4
0
1
2
3
①
②
③
Skeleton
(x, 5)
(y, 6)
(y, 4)
points
Samples
Buckets
Step 1 Sample & get skeleton
Step 2 Sieve points
Step 3 Recurse
Step 2.1 Count #points in chunk i and bucket j in A[i][j]
P[]
Bucket
Chunk 0
Chunk 1
Chunk 2
A
B
Chunk
Step 2.2 column-major prefix sum
B[i][j]: offset for chunk i and bucket j
Pointers to the four buckets: 0, 2, 5, 7 (first row in B[])
Index
P′[]
Step 2.3 Reorder points
Example
A[1][2] = 1 : 1 point in chunk 1 falls into bucket 2
B[1][2] = 6 : Points in chunk 1 going to bucket 2 start from index 6

Figure 1: An illustration of our kd-tree construction algorithm, with a detailed overview on the *sieving step*.

$$\mathbb{P}\left[X \geq \frac{s}{2}\right] \leq \exp\left(-\frac{\delta^2\mu}{2+\delta}\right) = \exp\left(-\frac{\epsilon^2 s}{16-4\epsilon}\right)$$
$$\leq \exp\left(-\frac{12c\log n}{16-4\epsilon}\right) = \frac{1}{n^{c\cdot\frac{12\log_2 e}{16-4\epsilon}}} \leq \frac{1}{n^c}.$$

The right subtree has the same low probability to be unbalanced, so taking the union bound gives the stated bound. □

Lemma 3.2 (Tree height). *The total height of a Pkd-tree with size n is $O(\log n)$ for $\sigma = \Omega(\log n)$, or $\log n + O(1)$ for $\sigma = \Omega(\log^3 n)$, both* whp.

Proof. To prove the first part, we will use $\epsilon = 1$ in Lem. 3.1. In this case, for $\sigma = 12c\log n$, one subtree can have at most 3/4 of the size of the parent with probability $1 - 1/n^c$, which means that the tree size shrinks by a quarter every level. This indicates the tree heights to be $O(\log n)$ *whp* for any constant $c > 0$.

We now show the second part of this lemma. For leaf wrap $\phi \geq 4$, the tree has height 1 for $n \leq 4$. By plugging in $\epsilon = O(1)/\log n$, the tree height h can be solved as $\log n + O(1)$. Similar to the above, here in the worst case, the children's subtree size shrinks by at least $1/2 + \epsilon/4$. For simplicity, we let $\epsilon = 4/\log n$ so $1/2 + \epsilon/4 = 1/2 + 1/\log n$. Hence, the tree height satisfies $(1/2 + 1/\log n)^h = 1/n$, so $h = -\log n/\log(1/2 + 1/\log n)$. This solves to $\delta = h - \log n = -\log n/\log(1/2 + 1/\log n) - \log n = O(1)$ for $n > 4$. This analysis is quite complicated—due to the space limit, we put it in the appendix A.1. The high-level idea is to replace $t = \log n$, so $\delta = f(t) = -t/(\log(1/2 - 1/t)) - t = t/(1 + \log t - \log(t-2)) - t$. We show that $f(t)$ is decreasing for $t \geq 2$ by proving $f'(t) < 0$ for $t \geq 2$. Since we have multiple logarithmic functions in the denominator, we can compute the second and third derivatives and need a few algebraic techniques to remove them. □

Later in this paper, we will experimentally show that maintaining the perfect balancing of kd-tree (tree height of $\log_2 n + O(1)$) is not necessary for the most kd-tree's use cases. Hence, throughout the rest of the analysis, we will use $\sigma = \Theta(\log n)$ and assume the tree height as $O(\log n)$.

With these lemmas, we now show that Alg. 1 is theoretically efficient if we plug in the appropriate parameters. Particularly, we set 1) skeleton height $\lambda = \epsilon \log M$ for some constant $\epsilon < 1/2$, so that the recursive depth of Alg. 1 is $O(\log n)/\lambda$ *whp*; and 2) chunk size $l = 2^\lambda$, so array A and B have size $O(|P|)$, and operations on them will have $O(1)$ cost amortized to each input point. We also assume $M = \Omega(\text{polylog}(n))$, which is true for realistic settings.

Theorem 3.3 (Construction cost). *With the parameters specified above, Alg. 1 constructs a Pkd-tree of size n in optimal $O(n\log n)$ work and $O(\text{Sort}(n)) = O((n/B)\log_M n)$ I/O cost, and has $O(M^\epsilon \log_M n)$ span for constant $0 < \epsilon < 1/2$, all with high probability. Here M is the small-memory size and B us the block size.*

Proof. We start with the work bound. Although the entire algorithm has several steps, each input point is operated for $O(1)$ times in each recursive level, except for lines 15 and 22. For these two lines, looking up the bucket id has $O(\lambda)$ work. Since the total recursive depth of Alg. 1 is $O(\log n)/\lambda$ *whp*, the work is $O(\lambda \cdot (\log n)/\lambda) = O(\log n)$ *whp* per input point, leading to total $O(n\log n)$ work *whp*.

We now analyze the span of Alg. 1. The algorithm starts with sampling $2^\lambda \cdot \sigma$ points and build a tree skeleton with λ levels. Taking samples and building the skeleton can be trivially parallelized in $O(\log^2 n)$ span (using the algorithm at the beginning of Sec. 3). In the sieving step, each chunk has $l = 2^\lambda$ elements that are processed sequentially, and all chunks are processed in parallel. This gives $O(2^\lambda)$ span. The column-major prefix sum on line 17 can be computed in $O(\log n)$ span [16], and all other parts also have $O(\log n)$ span. The total span for one level of recursion is therefore $O(l + \log^2 n) = O(M^\epsilon)$, assuming $M = \Omega(polylog(n))$. Since Alg. 1 have $O(\log n)/\lambda$ recursive levels *whp*, the span for Alg. 1 is $O(M^\epsilon \log_M n)$ *whp*.

We finally analyze the I/O cost. Based on the parameter choosing, the sampling skeleton construction fully fit in cache. In each sieving step, by selecting $l = 2^\lambda = M^\epsilon \leq \sqrt{M}$, the array $A[i][\cdot]$ and $B[i][\cdot]$ fits in cache, so the loop bodies lines 14 and 21 will access the input points in serial, incurring $O(n/B)$ I/O cost. All other parts also have $O(n/B)$ I/O cost, including the column-major prefix sum on line 17 [16]. Hence, the total I/O cost for Alg. 1 is $O(n/B)$ per recursive level, multiplied by $O((\log n)/\lambda) = O(\log n/\log M) = O(\log_M n)$ levels *whp*, which is $O(n/B \cdot \log_M n)$. □

The work and I/O cost in Thm. 3.3 are the same as sorting (modulo randomization) [41] and hence optimal. The span bound can also be optimized to $O(\log n \log_M n)$ *whp*, using a similar approach in [16], with the details given in the proof of the theorem below.

Theorem 3.4 (Improved span). *A Pkd-tree of size n can be built in optimal $O(n\log n)$ work and $O(\text{Sort}(n)) = O((n/B)\log_M n)$ I/O cost, and has $O(\log n \log_M n)$ span, all with high probability.*

Proof. The $O(2^\lambda) = O(M^\epsilon)$ span per recursive level is caused by the two sequential loops on lines 14 and 21. These two loops can be parallelized by a sorting-then-merging process as in [16]. The high-level idea is to first sort (instead of just count) the entries in the loop on line 14 based on the leaf labels. Once sorted, we can easily get the count of the points in each leaf. Then on line 21, once the array is sorted, points can be distributed in parallel. We refer the readers to [16] for more details. The span bound for each level is $O(\log n)$ [15, 16], so in total it will be $O(\log n \log_M n)$ *whp*. □

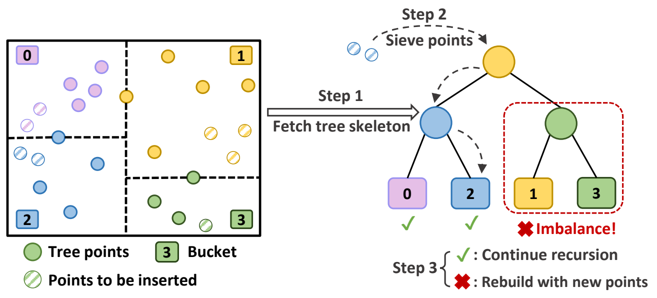


Figure 2: Illustation of our batch insertion to a kd-tree. We first sieve points into corresponding bucket and rebuild the (sub)-tree if necessary.

In practice we still use the previous version because $O(M^\epsilon \log_M n)$ span can enable sufficient parallelism, and the additional sorting to get the improved span may lead to performance overhead.

4 Parallel Algorithms for Batch Updates

In this section, we present our parallel update (insertions and deletions) algorithms for kd-trees. Here we consider the batch-parallel setting that inserts or deletes a batch of points P to the current Pkd-tree T. Pkd-trees does not always keeps the tree perfectly balanced as in existing works [29, 44, 63]—our key idea to achieve both high performance and good theoretical guarantee is to allow the tree to be *weight-balanced* and to *partially reconstruct* the tree on upon imbalance. Figure 2 illustrates the high-level idea. In particular, we allow the sizes of the two subtrees be off by at most a factor of α, i.e., the size of a subtree can range from $(0.5-\alpha)$ to $(0.5+\alpha)$ times the size of its parent. Such a relaxation allows most updates to be performed lazily, until at least a significant fraction of a subtree has been modified. If such a case happens, we partially rebuild the tree (i.e., only on those unbalanced subtrees) using Alg. 1, of which cost can be amortized to the updated points in previous batches. This idea of lazy updates with reconstruction has been studied sequentially on various of trees for point updates [32, 54]. The key challenge here is to adapt this idea to the batch-parallel setting while maintaining theoretical and practical efficiency. Theoretically, we show efficient work and I/O bounds, and high parallelism of our new batch-update algorithm. In practice, we conduct in-depth study of the performance with the relaxation of balancing criteria. In Sec. 7.4, we show that the query performance of Pkd-tree is mostly unaffected for $\alpha \le 0.4$ (the two subtree sizes differ by up to 9×). By allowing slight skewness, Pkd-tree significantly outperforms all existing counterparts on updates.

4.1 Batch Insertion and Deletion Algorithms

Our parallel batch update algorithms takes as input a kd-tree T and a set of points P, and insert the points in P to (or delete P from) T. The methodology for our insertion and deletion functions are almost identical. In general, we partition the points in P that belongs to each subtree using the same sieving algorithm in construction, and recursively handle each subproblem in parallel. However, before the actual update is performed, we will use the current subtree size and the bucket sizes (i.e., the number of insertions/deletions falling into this subtree) to determine if the subtree may become unbalanced after update. If so, we will directly flatten all points in the current subtree with the points to be inserted, and

Algorithm 2: Batch insertion

```
Input: A sequence of points $P$ and a $k$d-tree $T$.
Output: A $k$d-tree with $P$ inserted.
Parameter: $\lambda$: the maximum height of a fetched tree skeleton.
           $\phi$: the leave wrap of $T$.
// Insert points $P$ into $k$d-tree $T$
1  Function BatchInsert($T, P$)
2      if $P = \emptyset$ then return $T$
3      if $T$ is leaf then return BuildTree($T \cup P$)        // Insert into a leaf
4      $\mathcal{T} \leftarrow$ the skeleton at $T$
5      Sieve points to the corresponding bucket in $\mathcal{T}$ using Alg. 1. Let $R[i]$ be
         the sequence of all points sieved to bucket $i$.
6      $t \leftarrow$ The root of skeleton $\mathcal{T}$
7      return InsertToSkeleton($t, R[0..2^\lambda)$)
// Insert the buckets $R[l]$ to $R[r-1]$ to a node $t$ in the skeleton
8  Function InsertToSkeleton($t, R[l..r)$)
9      if $t$ is an external node in the skeleton then
10         $x \leftarrow$ the subtree at $t$
11         return BatchInsert($x, R[l]$)
12     else
13         if after insertion, the two subtrees at $t$ are weight-balanced then
14             $m \leftarrow$ number of buckets in $t.lc$
15             In Parallel:
16                 $t.lc \leftarrow$ InsertToSkeleton($t.lc, R[l, m)$)
17                 $t.rc \leftarrow$ InsertToSkeleton($t.rc, R[m, r)$)
18             return $t$
19         else return BuildTree($t \cup (\bigcup_{i=l}^{r-1} R[i])$)   // Rebuild the subtree
```

directly call the construction algorithm to create a (almost) perfectly balanced tree. Note that in our case it cannot be perfectly balanced due to our sampling-based construction algorithm, but in Sec. 4.2 we show our update algorithms are still theoretically efficient.

We first show our insertion algorithm in Alg. 2. We note that the only difference in the deletion algorithm is the base case on **??** and line 3 (i.e., removing points instead of adding them). For simplicity, we also assumes that $P \cap T = \emptyset$ for an insertion and $P \subseteq T$ for a deletion for now. Later we show how to slightly modify the algorithm to deal with the case where P and T partially overlap.

Alg. 2 inserts a set of points P into a kd-tree node T. It first skips the trivial case that $P = \emptyset$ at line 2. Then if T is a leaf, the algorithm will construct a tree based on $P \cup T$. Otherwise (T is an interior node), we will first grab the skeleton $\mathcal{T}$ at T. Similar to the construction algorithm, we will first sieve all points in P based on the skeleton $\mathcal{T}$, and deal with all subproblems recursively in parallel. With the set of points in each bucket and the original subtree size, we can compute the sizes of each subtree in $\mathcal{T}$ after insertion, and thus determine whether any of these subtrees are unbalanced. If we encounter node t in $\mathcal{T}$ that will become unbalanced after insertion, we will directly reconstruct the subtree using all original points in t and the points in P that belongs to this subspace. We do not need to further process the subtrees of t in this case. Otherwise, we recursively deal with the subtrees and their corresponding buckets.

Dealing with partially overlapped P and T. In practice, P and T may partially overlap, i.e., inserting points already in T and removing points not in T. In this case, directly using $|T| + |P|$ or $|T| - |P|$ to measure the tree size after updating cannot accurately determine imbalance of the tree. There are several approaches to get around this. The simplest one is to maintain a set S for all points in T, and always first check if $p \in P$ is already in T (for insertion) or not in T (for deletion) before applying the updates. We can use hash tables [64] or BSTs [14, 68] to maintain S that are highly parallel and without affecting the asymptotic bounds. Our implementation use the second approach by traversing the tree in two rounds. In the first round, we record the actual size for each subtree after insertion/deleltion. Then in the second round we perform the actual update, and rebalance based on the new sizes computed.

4.2 Theoretical Analysis

Note that although our batch update algorithms are conceptually simple, we now show that it has good theoretical guarantees, and later in Sec. 7 we will show its high practical performance. Since the update algorithms use the construction algorithm as a subroutine, we need to synergetically set up the parameters for both algorithms—we select $\sigma = (12c \log n)/\alpha^2$ for some constant $c > 0$ to ensure a low amortized cost in Thm. 4.1. Here we assume α is a constant and $\sigma = \Theta(\log n)$. For simplicity, here we assume the batch size m is smaller than the tree size n; otherwise, for batch insertion we just replace all n in the bounds to $m + n$ and there is no change for batch deletion.

Theorem 4.1 (Updates). *A batch update (insertions or deletions) of a batch of size m on a Pkd-tree of size n has $O(\log n \log_M n)$ span* whp*; the amortized work and I/O cost per element in the batch is $O(\log^2 n)$ and $O(\log(n/m) + (\log n \log_M n)/B)$* whp*, respectively.*

Due to the space limit, we defer the full proof in appendix A.2. The overall structure of this analysis is similar to Thm. 3.3, with the additional information that traversing m leaves in a binary tree of size n touches $O(m \log(n/m))$ tree nodes [14]. Again in practice, we use the sieve approach in Alg. 1, which leads to $O(\sqrt{M} \log n)$ span and supports sufficient parallelism.

The cost for Pkd-tree can be higher than using the logarithmic method (see introduction in Sec. 1)—e.g., the work per point is $O(\log^2 n)$ instead of $O(\log n)$. However, we note that the bound for Pkd-tree is not tight. Unless in the adversarial case, the update cost per point is more likely to be $O(\log n)$ when subtree rebuild is less frequent. In Sec. 7, we will use various experiments to show the update is faster than the logarithmic method practically. Meanwhile, since Pkd-tree only keeps a single tree rather than $O(\log n)$ trees, the query performance on Pkd-tree is significantly superior.

In addition, Pkd-tree can support stronger balancing criterion for $\alpha = o(1)$ (although in Sec. 7 we experimentally show this is necessarily for most applications). In this case, the amortized work and I/O cost per point will increase to $O((\log^2 n)/\alpha)$ and $O(\log(n/m) + (\log n \log_M n)/B\alpha)$ *whp*, respectively.

5 Queries on Pkd-trees

We note that unlike kd-trees based on logarithmic method (i.e., maintaining $O(\log n)$ trees), Pkd-tree always maintains a single tree, so any query that applies to the classic static kd-tree works on Pkd-tree without any modifications. We can, of course, carefully study the implementation details for each query, that applies to Pkd-trees and the classic kd-trees, which will be discussed in Sec. 6. In this section, we review the analysis for queries on kd-trees.

Lemma 5.1 ([17]). *An orthogonal range count costs $O(2^{h(D-1)/D})$ work on a kd-tree of height h, and the orthogonal range query has $O(2^{h(D-1)/D} + k)$ work, where k is the output size.*

This indicates that when we pick $\alpha = 1/O(\log n)$, the tree height is $\log n + O(1)$ (Lem. 3.2), and the range count will cost $O(n^{(D-1)/D})$, matching the best cost on a kd-tree. Interestingly, however, in Sec. 7.1, we show that this bound is not tight in most cases in practice. Slightly increasing the tree height h will only marginally affect the actual query cost.

The k-nearest neighbor (k-NN) query is another typical application using kd-trees. Many theoretical analyses on the classic (single) kd-trees also apply to Pkd-trees. For instance, one can show logarithmic expected query cost for kd-trees on uniformly distributed inputs [30, 67] or on approximate range queries [7, 27, 51]. These analyses only requires asymptotic tree heights, so the bounds also apply to Pkd-trees *whp*.

6 Implementation Details

In this section, we discuss additional implementation details that contributes to the practical efficiency of Pkd-tree.

Avoid the extra copies. For simplicity, in Alg. 1, we assume we copy the array of points in P' back to P on line 25 after we distribute the points. In practice, this copy can be avoided by rolling P and P' in each recursive call. This significantly saves unnecessary memory access in the algorithm.

Parameter Choosing. Our theoretical analysis in Sec. 3.2 suggests $\lambda = c \log M$ for some constant $c < 1$. In practice, we observed that using $\lambda = 4$ to 10 generally gives good performance. We use $\lambda = 6$ for Pkd-tree. We use $\phi = 32$ for leaf warp size, and over sampling rate $\sigma = 32$. We use balancing parameter $\alpha = 0.3$, and further explain our choice in Sec. 7.4.

Handling of Duplicates. One issue we observed in some existing kd-tree implementations (e.g., CGAL) is the inefficiency in dealing with duplicate points. Because many points may fall onto the split hyperplane and a default approach will put all of them on one side of the tree, the tree height (and thus the query performance) may degenerate significantly. To handle this, Pkd-tree uses a special ***heavy leaf***. When all points in a node are duplicates, we use a heavy leaf to store the coordinate and the count.

Minimize the Memory Usage. A key effort in Pkd-tree is to minimizing the memory usage. Reducing the memory footprint is crucial in at least two aspects. First, it allows Pkd-tree to handle larger inputs. Second, smaller memory footprint generally means lower I/Os and better cache utilization, leading to better performance.

There are a few approaches in the design of Pkd-tree reduce memory usage. The first is leaf wrapping as mentioned, which creates a flat tree leaf when the subtree size drops below a certain threshold (32 in Pkd-tree). Second, we try to keep each interior node as small as possible to fit more tree nodes in the cache. The only additional information we keep for each subtree is the size, which is needed in our weight-balance scheme and is used in range count queries. Therefore, unlike Zd-tree, ParGeo, and CGAL, Pkd-tree does not store a *bounding box* of each tree node, which is the smallest box containing all points in this subtree. This box is used in queries to prune the subtree: when the query does not overlap with the box, the entire subtree can be skipped. In Pkd-tree, instead of storing the bounding box, the query will compute the subspace of each subtree based on the parent's subspace and the splitter hyperplane. This is not as tight as the bounding box, but in our experiments, we observed that avoiding explicitly storing bounding boxes still gives better overall performance for Pkd-tree.

Maintaining Candidates in k-NN Queries. During k-NN search, updating the candidates is a write-extensive process that takes large portion of the search time. We use a standard approach as in CGAL, which uses a priority queue using a max-heap of size k. When the queue is full, elements are inserted only if it is smaller than the root of the heap.

7 Experiments

Setup. We conducted experiments on a machine equipped with 96 cores (192 hyper-threads) with four-way Intel Xeon Gold 6252 CPU and 1.5 TiB RAM. We use `C++` for all implementation and ParlayLib [12] for parallelism. We tested operations including tree construction, batch insertion/deletion, all-point k-NN query (report the k nearest neighbors for all points in the dataset), range report and range count queries. For each experiment, we ran it for 4 times and take the average of the last 3. We use $\lambda = 6$ as explained in Sec. 6. We set $\alpha = 0.3$, and discuss the choice of parameter in Sec. 7.4.

Baselines. We compare our implementation with four existing implementations, described as follows.

- **Zd-tree** [13]. A parallel quad/octree based on Morton order (Z-order curve). Zd-tree maps each point to an integer by interleaving the coordinates' bits and uses integer sort as preprocessing. Due to precision issues, Zd-tree's implementation only supports inputs in 2 or 3 dimensions.
- **BHL-tree** [71]. The single-tree version of the parallel kd-tree from ParGeo, using the binary heap layout. It supports parallel construction, but a batch update simply rebuilds the whole tree.
- **Log-tree** [71]. The version based on logarithmic method from ParGeo. Log-tree keeps $O(\log n)$ BHL-trees with sizes 2^0, 2^1,2^2, etc. A batch update is performed by merging and rebuilding a subset of the trees. However, queries have to be performed on all $O(\log n)$ trees and combine the results.
- **CGAL** [29]. The kd-tree in the open-source computational geometry library CGAL. CGAL integrates parallelism using the threading building blocks (TBB). CGAL supports parallel tree construction, k-NN, and range query. The batch insertion triggers a rebuild of the whole tree together with the inserted points, and the batch deletion is implemented by removing the points one by one. CGAL's implementation has known scalability issues [13], but fairness, we use 192 hyperthreads when testing CGAL.

Datasets. We tested both synthetic and real-world datasets. We will introduce the real-world datasets in Sec. 7.2. For synthetic datasets, we use two distributions: `Varden` [33] and `Uniform`. `Varden` is generated by randomly walk on the local neighborhood, but with restart to jump to a random place with probability, so that generating skewed distribution. `Uniform` draws points within a box uniformly at random. Both `Uniform` and `Varden` use 64-bit integer coordinates. We tested dataset sizes of 10^8 (100M) and 10^9 (1000M). For simplicity, we shorthand each test by the dimension, distribution, and size, e.g., "3D-V-100M" stands for 3-dimensional points from `Varden` with size 100 million, and so as for other parameter settings.

7.1 Basic Operation Evaluation

Overall Performance. We summarize the performance for Pkd-tree and other baselines in Table 3. We tested construction with 100M and 1000M points, batch insertions and deletions with a batch size of 10% of the total tree size, as well as two queries: all-point 10-NN query, and range report query with output size between 10^4 and 10^8. None of the other baselines support range count query, so we test range count queries for Pkd-tree separately, and show the result later in this section and in Table 4. Pkd-tree scales to large datasets and relatively high dimensions—only Pkd-tree can stably perform all operations on 1 billion points in up to 9 dimensions.

For construction, Pkd-tree is the fastest in all tests, which is 1.4–2.1× faster than Zd-tree, 8.2–13.7× than Log-tree, 9.0–12.0× than BHL-tree, and 52.1–406× than CGAL. All Pkd-tree, Zd-tree, Log-tree, and BHL-tree scale well with the increasing of processors (see the scalability curve in Figure 8). Among them, Pkd-tree is faster than Log-tree and BHL-tree due to the better I/O efficiency of Alg. 1. It is interesting to see that, Pkd-tree is even faster than Zd-tree. We will study the high performance of Pkd-tree construction in Sec. 7.3 in details. As mentioned, CGAL has a known scalability issue [13] and cannot utilize 36 or more threads (also see the scalability curve in Figure 8). For this reason, CGAL is much slower than others and we exclude it in 1000M tests because it took too long to finish.

For batch updates, Pkd-tree is always the fastest except for four instances, where Zd-tree is the fastest in three cases and Log-tree is the fastest in one. Note that the Zd-tree as a quad/octree based on space-filling curve, should be faster than kd-trees on updates, with the disadvantages on query performance and others. However, our highly-optimized Pkd-tree has even better query performance than Zd-tree in most cases, due to good theoretical bounds and careful implementation.

Finally, in both k-NN and range queries, Pkd-tree is also always the fastest except for one case in k-NN query (Pkd-tree is only 2% slower than CGAL), and three cases in range query, where Pkd-tree can be 1.1–2.2× slower then BHL-tree, all in high dimensions. As mentioned in Sec. 6, Pkd-tree avoids explicitly storing the bounding boxes to optimize memory usage, but computes the subspaces for each subtree on-the-fly in queries. This approach trade off (slower) query performance for (faster) construction and updates. Even so, our query performance is still competitive, and is the fastest for 24 out of 28 instances, which is based on our careful engineering discussed in Sec. 6. Also, another interesting finding is that almost all kd-trees showed better performance on `Varden` than `Uniform` in k-NN queries. Since `Varden` is a skewed distribution, the output ranges for most points are small, yielding faster pruning for non-overlapping subtrees.

In the following, we present in-depth performance study for updates and queries. We use synthetic datasets with size 100M in 3 dimensions as benchmark for the rest of evaluations.

Batch Update. To carefully study the performance of batch updates, we design experiments to vary the batch sizes, and show the results of all algorithms in Figure 3. The batch insertion is to first construct a tree and then insert another batch from the same distribution into the tree; conversely, the batch deletion is to delete batch of points from the tree.

Our Pkd-tree shows the best performance for batch update on

		Uniform-100M					Varden-100M					Uniform-1000M				Varden-1000M			
Op.	Dims	Ours	Zd-tree	Log-tree	BHL-tree	CGAL	Ours	Zd-tree	Log-tree	BHL-tree	CGAL	Ours	Zd-tree	Log-tree	BHL-tree	Ours	Zd-tree	Log-tree	BHL-tree
Build	2	.246	.523	3.92	3.24	98.8	.255	.501	3.71	2.96	40.6	3.20	5.59	36.2	31.8	3.65	5.31	34.4	29.8
	3	.293	.555	4.44	3.70	94.4	.316	.530	4.42	3.66	33.6	3.75	6.06	43.7	39.8	4.77	6.89	43.1	38.8
	5	.432	n.a.	5.90	5.10	103	.480	n.a.	5.89	5.11	38.3	5.69	n.a.	59.8	58.3	6.46	n.a.	58.7	58.4
	9	.720	n.a.	9.19	8.66	120	.821	n.a.	8.99	8.64	42.8	9.77	n.a.	95.2	103	11.2	n.a.	92.2	103
Batch insert (10%)	2	.044	.145	1.09	3.30	119	.050	.059	.980	2.99	89.2	.480	1.65	30.7	39.3	.462	1.08	27.8	36.8
	3	.050	.148	1.29	3.81	123	.094	.066	1.26	3.77	62.7	.527	1.70	39.2	49.1	.566	.772	37.1	47.6
	5	.063	n.a.	1.79	5.57	145	.136	n.a.	1.76	5.51	64.3	.664	n.a.	55.0	69.4	1.46	n.a.	54.9	69.3
	9	.097	n.a.	2.93	9.62	170	.221	n.a.	2.84	9.58	68.0	.985	n.a.	161	122	2.21	n.a.	174	121
Batch delete (10%)	2	.050	.128	.240	2.83	28.5	.076	.056	.310	2.66	5.88	.530	1.50	1.50	32.5	.660	.992	1.96	30.0
	3	.058	.149	.370	3.47	35.4	.097	.042	.410	3.42	4.00	.617	1.72	4.20	41.8	1.05	.567	4.92	41.7
	5	.081	n.a.	.510	4.86	29.6	.143	n.a.	.550	4.87	3.30	.870	n.a.	2.50	74.3	1.59	n.a.	2.53	75.7
	9	.130	n.a.	.760	8.45	30.2	.230	n.a.	.850	8.51	3.51	1.27	n.a.	3.40	134	2.52	n.a.	3.17	135
10-NN (all)	2	1.05	1.68	24.2	3.93	7.99	1.04	1.62	17.9	2.25	3.68	11.3	18.9	s.f.	s.f.	11.4	18.4	s.f.	s.f.
	3	2.12	3.09	33.2	8.33	16.9	1.98	3.67	20.3	3.32	1.95	22.4	33.0	s.f.	s.f.	22.0	30.6	s.f.	s.f.
	5	10.5	n.a.	t.o.	59.6	90.6	3.31	n.a.	t.o.	5.68	3.89	114	n.a.	s.f.	s.f.	35.7	n.a.	s.f.	s.f.
	9	631	n.a.	t.o.	3040	1181	5.87	n.a.	t.o.	80.3	33.4	8433	n.a.	s.f.	s.f.	56.5	n.a.	s.f.	s.f.
Range query (10K,100M]	2	.145	n.a.	.358	.220	127	.143	n.a.	.359	.220	136	.389	n.a.	2.16	1.70	.381	n.a.	2.03	1.60
	3	.274	n.a.	.665	.414	108	.257	n.a.	.599	.407	131	.704	n.a.	3.41	2.50	.715	n.a.	3.44	2.42
	5	.841	n.a.	1.64	1.08	104	.850	n.a.	1.14	.749	113	2.17	n.a.	7.45	5.08	2.29	n.a.	6.11	4.36
	9	4.95	n.a.	7.63	5.57	79.0	2.95	n.a.	2.01	1.33	115	12.8	n.a.	29.3	20.9	12.8	n.a.	10.8	7.73

Table 3: Running time (in seconds) for Pkd-tree and other baselines. Lower is better. Baselines are introduced in Sec. 7. The fastest runtime for each benchmark is underlined. "n.a.": not available. "s.f.": segmentation fault. "t.o.": time out (more than 3 hours).

`Uniform`, while it is slightly slower than Zd-tree for batch insertion and small-size batch deletion using `Varden`. As mentioned above, Zd-tree uses space-filling curves to handle each multi-dimensional point as integers, which is more efficient to handle updates but does not apply to dimensions higher than 3. Even so, Pkd-tree is still competitive and the running time is within 1.4–2.6× of Zd-tree for batch insertion. As for batch deletion, the number of remaining points in the tree reduces as the batch size increases, leading to faster partial rebuilding and less total time for the Pkd-tree.

Both BHL-tree and CGAL fully rebuild the tree on insertions. Therefore, they show a flat curve of running time when increasing the batch size and are much slower than Pkd-tree and Zd-tree based on rebalancing. Log-tree's performance sits in the middle. It avoids fully rebuilding the tree, but merging a subset of the trees. One may notice that there is a jump for Log-tree in batch insertions. The reason is that for small batch, the Log-tree only needs to merge a few small trees. However, inserting 30% new points triggers a new tree to be built (total size exceeds the next power of 2), making the performance similar to a fully rebuild as in BHL-tree.

k-NN. We tested all-point k-NN queries on `Uniform` and `Varden` for $k \in \{1, 10, 100\}$ and show the result in Figure 4. All queries run in parallel. Pkd-tree is always among the fastest. Most other baselines are still competitive within 1.04–17.5×. The only exception is Log-tree: querying on all $O(\log n)$ kd-trees significantly slows down the performance. It also incurs high space usage, and were unable to run 100-NN query due to out-of-memory issue.

Range Query. To evaluate range queries, we create three types of queries: *small* (output size in $(0, 10^2]$), *medium* (output size in $(10^2, 10^4]$) and *large* (output size more than 10^4). For each query type, we report the average time of 10^4 distinct queries in parallel. The results are in Figure 5. Overall, Pkd-tree has the best performance. Most other baselines (except for CGAL) are competitive.

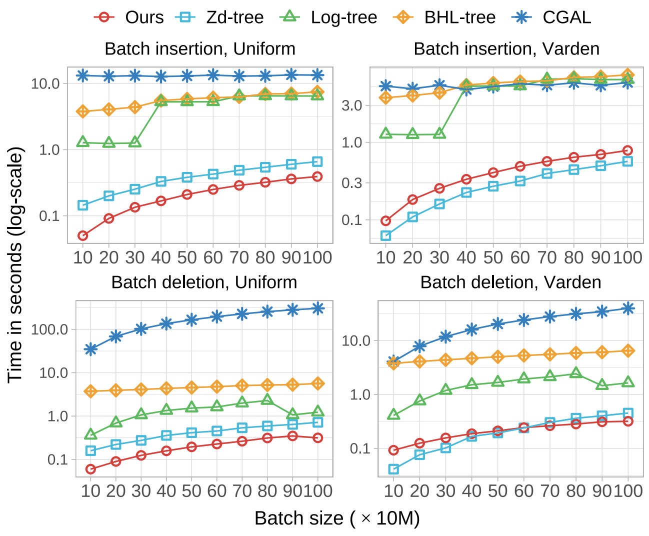


Figure 3: Time required for batch update on points from `Varden` and `Uniform` with size 100M and dimension 3. The batch size is the number of points in the batch. The time is measured in seconds and transformed into logarithmic scale.

Range Count. A range count query is similar to a range query but only reports the number of points in the queried range. Range count is a crucial subroutine in many applications [3, 55, 60, 72, 75]. However, no baseline provides a range count interface, and users have to call a regular range query and report the output size. Pkd-tree answers range count queries efficiently by using the subtree sizes stored in each tree node: when a subtree is totally in the query range, we skip the subtree and add its size to the return value. Table 4 presents the results. Comparing with the running time on range queries, a range count query is faster than reporting

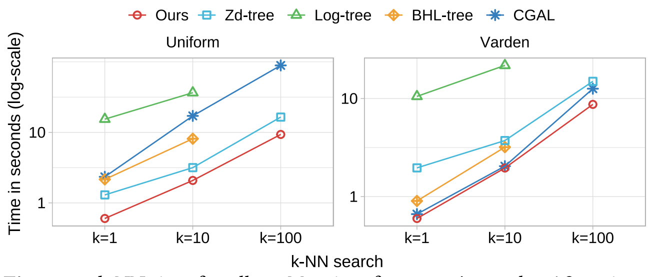


Figure 4: k-NN time for all 100M points from `Varden` and `Uniform` in 3 dimensions. The k is set to be 1, 10 and 100. The time is measured in seconds and plots in log-scale. There is no data for Log-tree and BHL-tree when $k = 100$ due to out-of-memory.

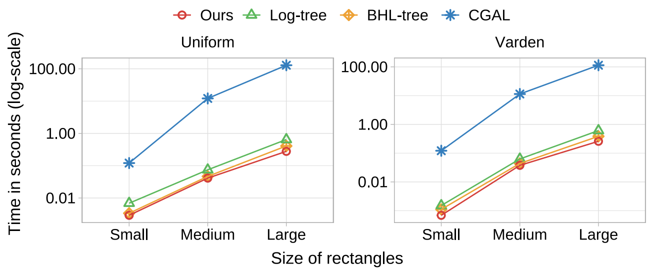


Figure 5: Orthogonal range query for three rectangle types in parallel. The benchmark contains 100M points with dimension 3. Each type of rectangle contains 10K candidates. The time is measured in seconds and transformed in log-scale. Zd-tree has no implementation of range query.

all elements in the range, e.g, for 3D-U-100M and 3D-V-100M, the range count is 0.7-3.0× faster and 0.7-3.8× faster than range query, respectively.

	`Uniform`			`Varden`		
Dims	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
2	0.001	0.008	0.014	0.001	0.008	0.013
3	0.004	0.032	0.093	0.001	0.026	0.069
5	0.009	0.212	0.698	0.028	0.180	0.627
9	0.272	2.18	5.12	0.010	0.792	2.95

Table 4: Orthogonal range count query on our Pkd-tree for three types of rectangle in parallel. The benchmark contains 100M points with dimension 3. Each type includes 10K random chosen rectangles.

7.2 Real-World Datasets

We evaluate the time for tree construction and k-NN, where $k \in \{1, 10, 100\}$, on real-world datasets below. Our benchmarks include very large datasets `COSMOS` (`CM`) [61] and `OpenStreetMap` (`OSM`) [39] that contains up to 2.7 billion points, and high-dimensional datasets `HT` [42] and `HouseHold` (`HH`) [40] with up to 10 dimensions. We also test the benchmark `GeoLifeNoScale` (`GLNS`) [76] that contains highly duplicated points. All coordinates are 64-bit real numbers. Table 5 demonstrates the results.

Pkd-tree shows the best performance in all but two instances. CGAL is 2.15× faster than Pkd-tree better performance in 10-NN on `HT`, but fails to build three out of five datasets: it does not support `OSM` due to high memory usage, and cannot run on `HH` and `GLNS` due to the inability to handle heavy duplicates. Zd-tree performed well in construction on `GLNS` (1.9× faster than Pkd-tree) but extremely poorly in queries for the same reason: with heavy duplicates, their

	Points	**Dims**	**Op.**	**Ours**	**Zd-tree**	**Log-tree**	**BHL-tree**	**CGAL**
HT	928K	10	Build	.008	n.a.	.678	.061	.472
			1-NN	.008	-	2.53	.015	.015
			10-NN	.043	-	2.81	.059	.020
HH	2.04M	7	Build	.054	n.a.	.716	.102	t.o.
			1-NN	.058	-	1.26	1.60	-
			10-NN	.229	-	2.60	3.19	-
GLNS	24M	3	Build	.256	.136	1.34	.792	s.f.
			1-NN	.274	631	3.74	1.31	-
			10-NN	.775	666	14.4	9.37	-
CM	321M	3	Build	1.54	1.75	16.7	13.3	184
			1-NN	2.79	4.49	25.9	5.24	5.94
			10-NN	9.09	10.1	s.f.	s.f.	33.0
OSM	2770M	2	Build	27.3	o.o.m.	o.o.m.	100	o.o.m.
			1-NN	31.3	-	-	s.f.	-
			10-NN	42.8	-	-	s.f.	-

Table 5: Measuring tree construction and k-NN time on read-world datasets for Pkd-tree and baselines. The "Points" is the number of points in the datasets and "Dims" is the dimension for the points. Details about baselines are discussed in Sec. 7. The fastest runtime for each benchmark is underlined. "n.a.": not available. "s.f.": segmentation fault. "t.o.": time out (more than 3 hours). "o.o.m.": out of memory.

approach based on space-filling curve may map many points to the same leaf, resulting in an unbounded leaf size. This makes construction simpler since fewer tree nodes are created, but each query may need to check all points in a relevant leaf one by one and becomes much slower. BHL-tree and Log-tree also suffer from high memory usage and cannot process for k-NN queries on `CM` and `GLNS`. The static single kd-tree, BHL-tree, is the most competitive to Pkd-tree on real-world datasets, but it is still 1.8–8.6× slower in construction and 1.4–28× slower in query than Pkd-tree.

7.3 Technique Analysis for Tree Construction

In the tree construction Alg. 1, we mainly employed two techniques to improve I/O efficiency: 1) build λ levels at a time to save total data movement, and 2) use sampling to determine the splitters to save memory accesses. We measure the performance of above techniques to tree construction by testing four versions of Pkd-tree with different levels of optimizations, and present the results in Figure 6: 1) the final version with sampling and $\lambda = 6$ (red bar), 2) using sampling with $\lambda = 1$, i.e., to construct one level at a time (blue bar), 3) finding the exact median in parallel to determine the splitter (green bar), and 4) finding the exact median sequentially to determine the splitter (yellow bar). We tested both `Uniform` and `Varden`, using 1 thread and 192 threads, respectively. Comparing the blue bar with green bar (or the blue bar with the yellow bar for the sequential setting), we can observe that sampling always improves the performance by 2.5×. However, comparing the red and blue bar, it indicates that building multiple levels together significantly improves the parallel performance, but does not affect the sequential performance much. This emphasizes the importance of I/O-efficiency in the parallel setting: when more threads run and access memory in parallel, memory bandwidth will become the main performance bottleneck, and optimizing I/O is crucial to improve performance.

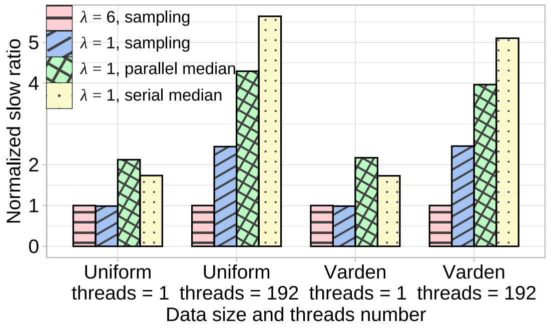


Figure 6: Measuring the effectiveness of techniques in tree construction, categorized by datasets (n = 100M, d = 3) and number of threads used. The vertical axis is the running time for different technique combinations normalized to the default one (use tree skeleton with λ = 6 and sampling).

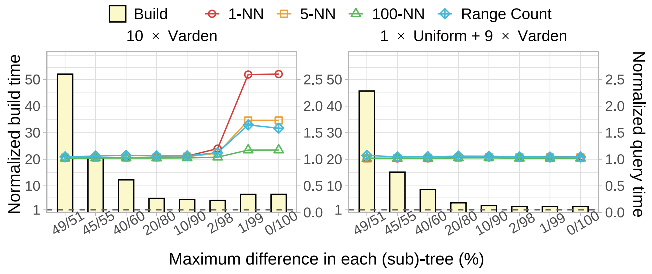


Figure 7: Construction and query time with varying balancing parameter α. The benchmark contains 10^8 insertions for 3D points in 100 batches and queries on the final tree. The bars and the left axis show the time to build the tree by 100 batch-insertions, normalized to the time on directly building the tree from the 10^8 points. The lines and the right axis show the query time on the final tree, normalized to the query time on the tree constructed directly from the 10^8 points.

7.4 Balancing Parameter Revisited

As mentioned previously, the main idea of Pkd-tree is to relax the strong balancing criteria to achieve much better performance in construction and updates. This may increase the tree height and affect the query performance, and the tradeoff is controlled by the parameter α. In this section, we systematically study the choice of α and its impact to the performance.

To do this, we create adversarial input such that belated rebalancing may result in large tree height. We generate skewed batch sequences and insert them into an empty tree by 100 batches, after which we perform queries on the tree to see how the imbalance affect query performance.We test α in a full range from 0.01 (almost always rebalance to perfect) to 0.50 (no rebalancing, siblings can be arbitrarily different). We tested multiple distributions and show two of the most representative ones in the paper:

- TYPE I: concatenation of ten distinct instances from 3D-V-10M.
- TYPE II: concatenation of one instance from 3D-U-10M and another one from 3D-V-90M.

Generally, given a fixed value of α, the tree constructed on TYPE I is more imbalance than TYPE II. Figure 7 presents the construction and k-NN time wrt. balancing parameter α. The "build time" is to construct a tree by inserting 100 batches incrementally. We normalize the construction time to if we directly build the tree T^* once using all points in the batches. The "query time" is to perform queries on the final constructed tree. We normalize it to if we

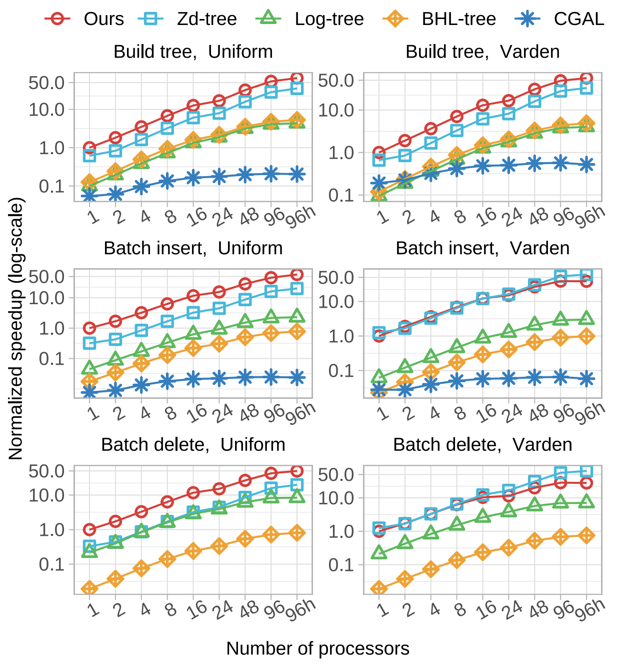


Figure 8: Self-speedup of basic operations over `Uniform` and `Varden` for our implementation and other baselines on different numbers of processors. Scalability of all tested systems. The curves are relative running time on different number of threads normalized to Pkd-tree running on one thread. Higher is better. The benchmark contains 100M points in 3D. "96h": 96 cores with hyper-threads. There is no data for CGAL in batch delete since it deletes points sequentially.

perform the same query on T^* obtained by construction. We use $x/y = (0.5-\alpha)/(0.5+\alpha)$ in the figure to indicate the degree of balanced controlled by α, which means that two sibling subtrees can differ by at most $x : y$. For both input sequences, the overall trend for construction time is decreasing when less balance is required, since rebalancing is triggered less frequently. There is a slight rebound when the subtrees are too imbalanced. This is because the cost of traversing the tree in batch insertion also increases when the tree grows skew. For queries, an imbalanced tree poses more negative influence on k-NN search with smaller value of k than the larger one. This is because when k becomes large, the main cost will be dominated by writing k elements in the input, and the imbalancing in tree will not be as significant.

Overall, the query performance seem to be reasonably resistant to tree imbalance: as long as the ratio of sibling tree sizes is within 2/98, the query performance is hardly affected. As we mentioned, we also tried several other adversarial distributions, and TYPE I is the one where query is affected the most. In most of the cases (e.g., TYPE II), the slowdown in query is within 1%. However, allowing relaxed balancing criteria greatly improved the update performance: if we allow 20/80 imbalance on subtrees, then 100 batch updates can be 9.9× more efficient than if the ratio of subtrees sizes have to be within 49/51. Based on these observations, we choose α = 0.3 in our implementation (20/80) to achieve a tradeoff between the balance of the tree and update performance.

			Layout& Update		I/O Optimizations	Notes&More Optimizations
1975	**Bentley [8]**	Seq.	Single tree;	No balance	-	• Original kd-tree paper
1978	**Bentley [9]**	Seq.	Log-tree; Tree merging	Perfect	-	• Proposed logarithmic method
1980	**kdb-tree [59]**	Seq.	B-tree; Overflow/underflow	Not shown	• B-tree layout	-
1983	**Overmars [54]**	Seq.	Single tree; Partial rebuild	Relaxed (weight balanced)	-	-
1997	**CGAL [29]**	Par.	Single tree; Full rebuild (sequential deletion)	Perfect	• Leaf wrap	• "CGAL" tested in Sec. 7 • Dimension choosing • Open-source library
2003	**Bkd-tree [56]**	Seq.	Log-tree; Tree merging	Perfect	• I/O-efficient construction • I/O-efficient point update	
2003	**Arge et al. [2]**	Seq.	Log-tree; Tree merging	Perfect	• I/O-efficient construction • I/O-efficient point update	
2016	**Agarwal et al. [1]**	Dist.	Single tree; Static	Relaxed (randomized)	-	• Sampling • Multi-level construction
2021	**ikd-tree [22]**	Seq.	Single tree; Partial rebuild (lazy deletion)	Relaxed (weight balanced)	-	• Lazy deletion
2021	**ParGeo [71]**	Par.	Log-tree; Tree merging (lazy deletion)	Perfect	• Leaf wrap • vEB layout	• "Log-tree" tested in Sec. 7 • Open-source library
2021	**ParGeo [71]**	Par.	Single tree; Full rebuild	Perfect	• Leaf wrap • Binary heap layout	• "BHL-tree" tested in Sec. 7 • Open-source library
2021	**zd-tree* [13]**	Par.	Quad/octree; Rotations	Balanced BST	• Leaf wrap	
	This paper	Par.	Partial rebuild	Relaxed (weight balanced, randomized)	• Leaf wrap • I/O-efficient construction • I/O-efficient batch update	• Sampling • Multi-level construction • Dimension choosing • Open-source library

Table 6: Summary of Related work. "Log-tree": logarithmic method (using $O(\log n)$ kd-trees). "Seq.": sequential; "Par.": parallel; "Dist.": distributed.

7.5 Parallel Scalability

We test the scalability for tree construction and batch update of our implementation and other baselines on both 3D-U-100M and 3D-V-100M. We normalize all running time to Pkd-tree on one core, and show the scalability in Figure 8. For `Uniform`, our implementation reaches approximately 66× self-relative speedup on build, 58× on batch insert and 50× on batch delete using 192 hyper-threads, which is about 1.9× higher than Zd-tree, 12× than BHL-tree 15× than Log-tree and 320× than CGAL. For `Varden`, Pkd-tree retains 56× self-relative speedup on tree construction, which is 1.7× faster than the best baseline Zd-tree. Meanwhile, our self-speedup for batch updates on `Varden` is much lower than `Uniform`. It reaches 38.6× for batch insertion and only 28× for deletion, which are 1.6× and 2.1× slower than Zd-tree respectively. The reason is that the distribution for `Varden` is extremely skewed and the number of partial rebuilding in Pkd-tree is much higher than `Uniform`, which poses a large overhead for batch update. However, Pkd-tree still outperforms Log-tree and CGAL by a large margin.

8 Related Work

In this section, we review the literature of the kd-tree, with a summary of the most relevant ones in Table 6. The original algorithm proposed by Bentley et al. [8, 30] does not include a rebalancing scheme, and assumes either static data, or inserting keys in a random order. Since then, researchers have been developing rebalancing schemes for kd-trees, mainly in the two categories: *logarithmic method* ("log-tree" in Table 6), and *partial rebuild*. The logarithmic method was first proposed by Bentley in [9], and has been followed up by later work, including optimizing the I/O costs [2, 56] and parallelism [71]. However, as shown in Table 3, maintaining $O(\log n)$ trees in the logarithmic method hampers the query efficiency significantly. Another issue is that insertions and deletions are asymmetry and need to be handled by inherently different approaches, which is more complicated. An alternative idea is to maintain a single tree and partially rebuild the unbalanced subtrees, which was proposed by Overmars [54]. Many papers followed up this idea, such as the KDB-tree [59], scapegoat k-d tree [32], ikd-tree [22], and the divided k-d tree [70]. Among them, only KDB-tree is I/O-optimized. None of them considered parallelism.

There have been many attempts to optimize the I/O cost for kd-trees. Earlier ones simply consider flatten the binary structure into a B-tree-like multiple-way tree [56, 59]. While the B-tree structure fits better for external memory, using it on the main memory slows down the query performance, and remains unclear how it coordinates with parallelism. Agarwal et al. [2] gave a smart grid-based construction algorithm, which achieves optimal $O(\text{Sort}(n))$ I/O cost. They also dynamize the tree using the logarithmic method. Later, Wang et al. [71] parallelized it (with some simplifications for practicality) and integrated it in the ParGeo library, which is compared to our Pkd-tree in this paper.

There exists parallel kd-tree algorithms, but most of them are not I/O-optimized or do not support a full interface (other than ParGeo). Static kd-tree construction is widely studied, using techniques such as randomization and space-filling curves [1, 21, 23, 25, 43, 58, 63, 73]. Agarwal et al. [1] showed an approach in the MPC model that optimizes the number of rounds in a distributed setting, but it does not show I/O and span bounds, and has no implementations. These approaches do not directly support updates. Although CGAL [29] have the interface for updates, it simply rebuilds the tree after updates, and the performance overhead can be significant.

In this paper, we also compare Pkd-trees to Zd-trees [13], which is a recent parallel quad/oct-tree implementation based on the Morton order. Compared to kd-trees, quad/oct-trees are simpler on construction and updates, but generally slower on queries; they also have limitations in dealing with $D > 3$ dimensions, non-integer coordinates or skewed data distributions.

9 Conclusion

We present a parallel kd-tree that achieves optimal theoretical work, span and I/O bound for tree construction and batch update. The tree has $\log n + O(1)$ depth, thus preserves the optimal query bound as well. We also provide a high-performance implementation of parallel kd-tree that outperforms all baselines we tested in terms of tree construction, batch update and various queries. We quantitatively study the effectiveness of different techniques employed in tree construction and experimentally show the resistance of kd-tree to query performance under imbalanced condition. The future work may focus on developing new balancing scheme for kd-tree in batch dynamic scene or try to achieve a better balance between the cost for rebuilding and higher tree quality.

References

[1] Pankaj Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. 2016. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Principles of Database Systems (PODS)*. 429–440.

[2] Pankaj K Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. 2003. Cache-oblivious data structures for orthogonal range searching. In *Proceedings of the nineteenth annual symposium on Computational geometry*. 237–245.

[3] Pankaj K Agarwal, Jeff Erickson, et al. 1999. Geometric range searching and its relatives. *Contemp. Math.* 223 (1999), 1–56.

[4] Alok Aggarwal and S Vitter, Jeffrey. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[5] I Al-Furajh, Srinivas Aluru, Sanjay Goil, and Sanjay Ranka. 2000. Parallel construction of multidimensional binary search trees. *IEEE Transactions on Parallel and Distributed Systems* 11, 2 (2000), 136–148.

[6] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[7] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)* 45, 6 (1998), 891–923.

[8] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[9] Jon Louis Bentley. 1979. Decomposable searching problems. *Inform. Process. Lett.* 8, 5 (1979), 244–251.

[10] Jon Louis Bentley. 1990. Experiments on traveling salesman heuristics. In *Proceedings of the first annual ACM-SIAM symposium on discrete algorithms*. 91–99.

[11] Jon Louis Bentley. 1990. K-d trees for semidynamic point sets. In *Proceedings of the sixth annual symposium on Computational geometry*. 187–197.

[12] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[13] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[14] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[15] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[16] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[17] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[18] Benjamin Blonder, Cecina Babich Morrow, Brian Maitner, David J Harris, Christine Lamanna, Cyrille Violle, Brian J Enquist, and Andrew J Kerkhoff. 2018. New approaches for delineating n-dimensional hypervolumes. *Methods in Ecology and Evolution* 9, 2 (2018), 305–319.

[19] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[20] Christian Böhm, Stefan Berchtold, and Daniel A Keim. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)* 33, 3 (2001), 322–373.

[21] Russell A Brown. 2014. Building a balanced kd tree in o (kn log n) time. *arXiv preprint arXiv:1410.5420* (2014).

[22] Yixi Cai, Wei Xu, and Fu Zhang. 2021. ikd-tree: An incremental kd tree for robotic applications. *arXiv preprint arXiv:2102.10808* (2021).

[23] Yu Cao, Xiaojiang Zhang, Boheng Duan, Wenjing Zhao, and Huizan Wang. 2020. An improved method to build the KD tree based on presorted results. In *International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 71–75.

[24] Yifei Chen, Yi Li, Rajiv Narayan, Aravind Subramanian, and Xiaohui Xie. 2016. Gene expression inference with deep learning. *Bioinformatics* 32, 12 (2016), 1832–1839.

[25] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L Bocchino Jr, Sarita V Adve, and John C Hart. 2010. Parallel SAH kD tree construction.. In *High performance graphics*. Citeseer, 77–86.

[26] Zhenyun Deng, Xiaoshu Zhu, Debo Cheng, Ming Zong, and Shichao Zhang. 2016. Efficient kNN classification algorithm for big data. *Neurocomputing* 195 (2016), 143–148.

[27] Matthew Dickerson, Christian A Duncan, and Michael T Goodrich. 2000. KD trees are better when cut on the longest side. In *European Symposium on Algorithms (ESA)*. Springer, 179–190.

[28] Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[29] Andreas Fabri and Sylvain Pion. 2009. CGAL: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 538–539.

[30] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)* 3, 3 (1977), 209–226.

[31] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.

[32] Igal Galperin and Ronald Rivest. 1993. Scapegoat Trees.. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 93. 165–174.

[33] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–45.

[34] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

[35] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[36] Yan Gu, Omar Obeya, and Julian Shun. 2021. Parallel In-Place Algorithms: Theory and Practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 114–128.

[37] Yulan Guo, Ferdous Sohel, Mohammed Bennamoun, Min Lu, and Jianwei Wan. 2013. Rotational projection statistics for 3D local surface description and object recognition. *International journal of computer vision* 105 (2013), 63–86.

[38] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3 (1994), 357–399.

[39] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive computing* 7, 4 (2008), 12–18.

[40] Georges Hebrail and Alice Berard. 2012. Individual household electric power consumption. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58K54.

[41] Jia-Wei Hong and H. T. Kung. 1981. I/O Complexity: The Red-Blue Pebble Game. In *ACM Symposium on Theory of Computing (STOC)*.

[42] Ramon Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai F Rulkov, and Irene Rodriguez-Lujan. 2016. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems* 157 (2016), 169–176.

[43] Warren Hunt, William R Mark, and Gordon Stoll. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing*. IEEE, 81–88.

[44] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2017. A progressive kd tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*. IEEE, 1–5.

[45] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence* 24, 7 (2002), 881–892.

[46] Jiaxin Li, Ben M Chen, and Gim Hee Lee. 2018. So-net: Self-organizing network for point cloud analysis. In *Proceedings of the IEEE conference on computer vision*

and pattern recognition. 9397–9406.

[47] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, 3835–3845.

[48] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. 2003. The global k-means clustering algorithm. *Pattern recognition* 36, 2 (2003), 451–461.

[49] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[50] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[51] Songrit Maneewongvatana and David M Mount. 1999. It's okay to be skinny, if your friends are fat. In *Workshop on Computational Geometry*, Vol. 2. Citeseer, 1–8.

[52] Leland McInnes and John Healy. 2017. Accelerated hierarchical density based clustering. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 33–42.

[53] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.

[54] Mark H Overmars. 1983. *The design of dynamic data structures*. Vol. 156. Springer Science & Business Media.

[55] Md Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[56] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases (SSTD)*. Springer, 46–65.

[57] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. on Computing* 18, 3 (1989), 594–607.

[58] Maximilian Reif and Thomas Neumann. 2022. A scalable and generic approach to range joins. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3018–3030.

[59] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 10–18.

[60] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[61] Nick Scoville, H Aussel, Marcella Brusa, Peter Capak, C Marcella Carollo, M Elvis, M Giavalisco, L Guzzo, G Hasinger, C Impey, et al. 2007. The cosmic evolution survey (COSMOS): overview. *The Astrophysical Journal Supplement Series* 172, 1 (2007), 1.

[62] Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. 2005. *Nearest-neighbor methods in learning and vision: theory and practice*. Vol. 3. MIT press Cambridge, MA, USA:.

[63] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, Vol. 26. Wiley Online Library, 395–404.

[64] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.

[65] Jonathan A Silva, Elaine R Faria, Rodrigo C Barros, Eduardo R Hruschka, André CPLF de Carvalho, and João Gama. 2013. Data stream clustering: A survey. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–31.

[66] Mark William Smith, Jonathan L Carrivick, and Duncan J Quincey. 2016. Structure from motion photogrammetry in physical geography. *Progress in physical geography* 40, 2 (2016), 247–275.

[67] Robert F Sproull. 1991. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica* 6 (1991), 579–589.

[68] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[69] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. Visualizing large-scale and high-dimensional data. In *Proceedings of the 25th international conference on world wide web*. 287–297.

[70] Marc J van Kreveld and Mark H Overmars. 1991. Divided kd trees. *Algorithmica* 6 (1991), 840–858.

[71] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 450–452.

[72] Kun-Lung Wu and Philip S Yu. 1998. Range-based bitmap indexing for high cardinality attributes with skew. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*. IEEE, 61–66.

[73] Hiroki Yamasaki, Atsushi Nunome, and Hiroaki Hirata. 2018. Parallelizing the Construction of a k-Dimensional Tree. In *2018 IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD)*. IEEE, 23–30.

[74] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. 2016. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. *Journal of medical systems* 40 (2016), 1–8.

[75] Jun Zhang, Xiaokui Xiao, and Xing Xie. 2016. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proceedings of the 2016 international conference on management of data*. 155–170.

[76] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.

A Appendix

A.1 Proof for Tree Height

LEMMA A.1. *Function* $f(n) = -\log n/\log(1/2+1/\log n) - \log n = O(1)$ *for* $n \geq 8$.

Proof. Let $t = \log n$, we have:

$$\begin{aligned} f(t) &= -\frac{t}{\log(t+2) - \log 2t} - t \\ &= -\left(\frac{t}{\log(t+2) - \log t - 1} + t\right) \\ &= -h(t) \end{aligned}$$

Clearly, $f(t) = O(1)$ when $t = 3$. We then show $f(t) > 0$ holds for $t \geq 3$, which is :

$$\begin{aligned} \log\frac{t+2}{2t} > \log\frac{1}{2} &\implies \frac{1}{\log(t+2) - \log 2t} + 1 < 0 \\ &\implies h(t) > 0 \end{aligned}$$

as desired. The remaining is to show $f(t)$ is decreasing, which equivalents to show $h(t)$ is increasing over $t > 3$. The derivative of $h(t)$ is

$$\begin{aligned} h'(t) &= \frac{\log(t+2) - \log t - 1 - t\left(\frac{c}{t+2} - \frac{c}{t}\right)}{(\log(t+2) - \log t - 1)^2} + 1 \\ &= \frac{\log\frac{t+2}{t} - 1 - \frac{c\cdot t}{t+2} + c}{\left(\log\frac{t+2}{t} - 1\right)^2} + 1 \end{aligned}$$

where $c = 1/\ln 2$. Let $k = (t+2)/t$. We wish to show that $h'(k) > 0$ holds for $k \longrightarrow 1^+$, namely,

$$\begin{aligned} &\frac{\log k - 1 - c/k + c}{(\log k - 1)^2} + 1 > 0 \\ \iff &\log^2 k - \log k - c/k + c > 0 \\ \iff &g(k) > 0 \end{aligned}$$

Since $g(1^+) > 0$, therefore, it is sufficient to show $g'(k) > 0$ holds for k, i.e.,

$$\begin{aligned} &2c^2 \cdot \ln k/k - c/k + c/k^2 > 0 \\ \iff &2c \cdot k\ln k - k > -1 \\ \iff &k(2c\ln k - 1) > -1 \end{aligned}$$

The function w.r.t k in LHS is increasing and equals to -1 (the RHS) when $k = 1$. Proof follows then. □

A.2 Proof for batch update

THEOREM A.2 (UPDATES). *A batch update (insertions or deletions) of a batch of size m on a Pkd-tree of size n has* $O(\log n \log_M n)$ *span* whp*; the amortized work and I/O cost per element in the batch is* $O(\log^2 n)$ *and* $O(\log(n/m) + (\log n \log_M n)/B)$ whp*, respectively.*

Proof. We will start with the span bound. According to Thm. 3.4, the sieve process and trees rebuilding (rebalancing) all have $O(\log n \log_M n)$ span *whp*. Note that the tree rebuilding (line 19) can only be triggered once on any tree path. The span for other parts is $O(\log n)$—the INSERTTOSKELETON function can be recursively call for $\lambda = O(\log n)$ levels each with constant cost. In total, the span is the same as the construction algorithm, since in the extreme case, the entire tree can be rebuilt.

Then we show the work bound. The cost to traverse the Pkd-tree and find the corresponding leaves to update is $O(\log n)$ per point **Ziyang:** *whp?*, proportional to the tree height. Once a rebuild is triggered (on line 19), the cost is $O(n' \log n')$ where n' is the subtree size. After that (or the original construction), each subtree will contains $(1/2\pm\sqrt{(12c\log n)/\sigma}/4)n' = (1/2\pm\alpha/4)n'$ points *whp* (Lem. 3.1). We need to insert at least another $3\alpha n'/4$ points for this subtree to be sufficiently imbalance that triggers the next rebuilding of this subtree. The amortized cost per point in this subtree is hence $O(\log n'/\alpha) = O(\log n')$ on this tree node assuming α is a constant. Note that Pkd-tree has the tree height of $O(\log n)$, so overall amortized work per inserted/deleted point is $O(\log^2 n)$.

We can analyze the I/O bound similarly. We first show the rebuilding cost. For a subtree of size n', the cost is $O((n'/B)\log_M n')$ (Thm. 3.3). The amortized cost per updated point, using the same analysis above, is $O((1/B)\log_M n')$. Again since the tree height is $O(\log n)$, the overall amortized work per inserted/deleted point is $O((\log n \log_M n)/B)$. Then, we consider the cost to traverse the tree and find the corresponding leaves to update. Finding m leaves in a tree of size n will touch $O(m\log(n/m))$ tree nodes [14], so the amortized I/O per point is $O(\log(n/m))$. Putting both cost together gives the stated I/O bound. □