# Parallel $k$d-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 $k$d-tree implementation that is efficient both theoretically and practically. The $k$d-tree is one of the most widely used data structures to manage multi-dimensional data. However, we observed that existing parallel implementations have challenges in supporting efficient updates and/or queries with high parallelism. The challenge arises mainly due to two reasons. First, the unique structure for $k$d-trees does not support simple rebalancing primitives such as rotations, making dynamic updates challenging. Second, the design space of $k$d-trees encompasses construction, update, and queries. Some existing approaches may incorporate parallel updates by sacrificing query performance. For the data structure to be practical and flexible, it is essential to holistically optimize construction, updates, and queries.

We propose Pkd-tree (Parallel $k$d-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 bounds in work (sequential time complexity), span (parallelism), and I/O cost. Our key techniques include 1) reconstruction-based update algorithms that guarantee the tree to be weight-balanced, and 2) an efficient construction algorithm that optimizes work, span, and I/O simultaneously. We design general techniques for both construction and updates. With a combination of new algorithmic insights and careful engineering effort, we achieved a highly optimized implementation of the 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 $k$d-tree implementations. In all tests, with competitive or better query performance, Pkd-tree is always the fastest in construction and updates among all $k$d-tree implementations. We released our code.

## 1 Introduction

The $k$d-tree is one of the most widely-used data structures for managing multi-dimensional data. A $k$d-tree maintains a set of points in $D$ dimensions[1], and supports various queries such as $k$-nearest neighbor ($k$-NN), orthogonal range count and range search. Compared to other counterparts, the $k$d-tree has its unique advantages, such as linear space, simple algorithms, being comparison-based (and thus resistant to skewed data), scaling to reasonably-large dimensions (being efficient up to $D \approx 10$) and supporting a wide range of query types. Due to these advantages, $k$d-tree is usually the choice of data structure in many applications. Indeed, after its invention by Bentley in 1975 [7], $k$d-tree has been widely used and cited by over ten thousand times across multiple areas such as databases [19, 33, 37, 48], data science [26, 53, 68, 73], machine learning [24, 45, 46, 62], clustering algorithms [44, 47, 50, 60, 65], and computational geometry [17, 36, 49, 66].

Due to the ever-growing data volume, it is imperative to consider parallelism in $k$d-trees. However, we observe a *significant gap* between the *wide usage* of $k$d-trees, and a *lack of efficient parallel implementation* to support high-performance $k$d-trees. The two parallel libraries we are aware of, CGAL [28] and ParGeo [70], both have difficulties scaling to today's large-scale data size. CGAL does not support parallel updates. Even in its sequential 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 $k$d-tree. As shown in Tab. 1, on 100M points in 2D, it is 17–23× slower in $k$-NN queries and 2.5× slower in range queries than a single $k$d-tree. Due to these issues, neither CGAL nor ParGeo outperforms Zd-tree [12], a recent parallel quad/octree 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 $k$d-trees, such as inefficiency for $D > 3$ dimensions, and the difficulty to deal with non-integer coordinates or highly skewed data.

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

The first challenge is to support efficient dynamic updates on $k$d-trees. $k$d-trees differ from other classic trees and do not support rebalancing primitives for updates, such as overflow/underflow (as in B-trees), or rotations (as in binary search trees). Sequentially, a few solutions have been proposed (see Sec. 7), 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 queries. It is therefore worth asking 1) what is the best way to achieve dynamism for $k$d-trees in parallel, and 2) how to achieve a tradeoff to give overall best performance and theoretical bounds in both updates and queries.

The second challenge comes from the inherent complication of the *design space* for $k$d-trees. As a classic data structure, the $k$d-tree is widely-used in various 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 has 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

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

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

**Table 1: Running time (in seconds) for Pkd-tree and other baselines on 100M points in 2D. Lower is better.** "Zd-tree" is a parallel quad/octree in [12], "Log-tree" is a parallel $k$d-tree using logarithmic method from [70], "BHL-tree" is a single $k$d-tree used in Log-tree [70]. "CGAL" is a geometry library [70]. More details are given in Sec. 6. The output size for "Range query" is controlled in range of $10^4$–$10^8$. The best performance for each test is underlined. The "n.a." stands for "not available".

their implementation collectively. A good implementation should require a 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 $k$d-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 $k$d-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 being perfectly balanced is needed in $k$d-tree (i.e., $\log_2 n + O(1)$ tree height), or whether a reasonably relaxed criterion is good enough (e.g., $O(\log n)$ tree height). We conducted experimental studies and showed that the query performance of $k$d-tree is reasonably consistent with minor imbalance (see Sec. 6.4). Motivated by this observation, we adopted a *randomized weight-balanced* approach, and such relaxation allows for much better 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 $k$d-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 $\lambda$ levels, and applies an efficient parallel algorithm to ***sieve all points down $\lambda$ levels by one round of data movement***. In experiments, we observe that sampling and multi-level construction improve performance by 1.6× and 2.4×, 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 $\alpha$ before invoking the rebalancing scheme, which identifies the unbalanced substructures and performs a ***local reconstruction*** on the affected subtrees. The parameter $\alpha$ controls the degree of balance, enabling a trade-off 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 $\alpha$ only mildly affects the query performance, while leading to much better update time.

[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.

While the high-level ideas of sampling [1, 5, 9, 10], multi-level construction [1, 58] and reconstruction-based balancing scheme [21, 31, 43, 54] have all been studied in previous work (for both $k$d-trees and other data structures), the unique challenge here is to integrate these elements cohesively in theory, such that they facilitate a unified implementation for $k$d-tree. Indeed, we are not aware of any existing work on $k$d-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. 7. Such a synergy of construction and update algorithms is crucial in Pkd-tree, since its update relies on efficient reconstruction. Pkd-tree achieves this by 1) designing the sieving step as a building block by borrowing ideas from recent parallel sorting algorithms [27], 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 and the cost bounds based on the parameters in Lem. 3.1 and 3.2 and Thm. 3.3 and 4.1. In practice, we provide a highly-optimized implementation that supports parallel construction, update, and three query types: $k$-nearest neighbor query ($k$-NN), range count and range search.

As illustrated in Tab. 1, Pkd-tree achieves much better performance than previous $k$d-trees due to the newly proposed techniques. On 100M points in 2D, Pkd-tree is at least 11.6× faster than all $k$d-tree baselines on construction, 13.4× on batch insertion, 4.1× on batch deletion, 1.7× on $k$-NN, and 1.5× on range query. Pkd-tree is also faster than Zd-tree in construction, queries, and most of batch update operations. Results in higher dimension and larger data are summarized in Tab. 3. We also design experiments to understand the performance gain of the proposed techniques in both construction and updates, discussed in Sec. 6.3. On our machine, Pkd-tree can process 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 $k$d-trees were able to perform all point 10-NN query on 1 billion points due to space- and/or time-inefficiency, while Pkd-tree only uses 11 seconds. We believe Pkd-tree is the first parallel dynamic $k$d-tree implementation that scales to billions of points with high performance. Our code is publicly available in [52]. For page limit, some proofs are deferred to the full version of this paper [51].

Our contributions include:

- Parallel construction and update algorithms for parallel $k$d-tree with theoretical efficiency;
- Parallel library (open-source) with high performance; and
- In-depth experimental study on parallel $k$d-trees.

## 2 Preliminaries

We present a table of notations used in this paper in Tab. 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 $\log_2(1 + n)$.

**Computational Model.** We analyze our algorithms using the ***work-span model*** in the classic fork-join model with ***binary-forking*** [6, 14, 18]. We assume multiple threads that share memory. Each thread is a sequential RAM augmented with a fork instruction, which spawns two child threads that run in parallel. The parent thread resumes execution upon the completion of both child threads. A parallel for-loop can be simulated by a logarithmic number of steps of forking. A computation can be viewed as a directed acyclic graph (DAG). The ***work*** **($W$)** of a parallel algorithm is the total number of operations within its DAG, and the ***span*** **($S$)** 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 $W/\rho + O(S)$ time *whp* (in $W$) with $\rho$ processors [6, 18, 34].

We use the classic I/O model [4, 30] to measure the cost of memory access. In the I/O model, the memory is divided into two 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) of infinite size. Both memories are organized as blocks with size $B$, thus in total $M/B$ blocks in the small-memory. The 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 number of data transfers during the algorithm.

***k*d-Trees.** We study points in Euclidean space in $D$ dimensions, and the distance between two points is their Euclidean distance. 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 binary tree data structure. 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 $k$d-tree signifies an axis-aligned splitting hyperplane $\langle d, x\rangle$. Points to the left of the hyperplane (those with their $d$-th dimension coordinates smaller than $x$) 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 $\phi$ (a small constant), which become 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 [7], choosing the dimension with the widest stretch [29], etc. Our Pkd-tree also uses the widest dimension as the cutting dimension. 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 $k$d-tree has a height of $\log_2 n + O(1)$ and can be constructed in $O(n \log n)$ work using $O(n)$ space.

$k$d-trees can answer different types of queries. We focus on *$k$-NN queries* (finding the $k$ nearest points in the $k$d-tree to a query point), 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).

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

**Table 2: Notations used in this paper.**

We use the (subtree) root pointer $T$ to denote a (sub-)$k$d-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 $k$d-tree is (weight-)balanced if $0.5 - \alpha \le |T.lc|/|T| \le 0.5 + \alpha$, and *unbalanced* otherwise. Essentially, this means that the two subtrees can be off from perfectly balanced by a factor of $\alpha$.

## 3 Our Tree Construction Algorithm

We start with our parallel $k$d-tree construction algorithm, which is also a building block in our parallel batch-update algorithm. Constructing a $k$d-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 $\phi$. 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 when the input is larger than cache size. We will refer to this algorithm as the *plain parallel $k$d-tree construction* algorithm and use it in base cases.

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 $\lambda$ 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 $\lambda$ levels. It then distributes all points into the corresponding subtrees ($2^\lambda$ of them) and recurse.

The main challenge here are 1) to move each point exactly once to its 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 $k$d-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 *k*d-tree Construction

We present our algorithm in Alg. 1 and an illustration in Fig. 1. The algorithm $T = \textsc{BuildTree}(P)$ builds a $k$d-tree $T$ on the input points in array $P$. As mentioned, our main idea is to use samples to decide all splitters in $\lambda$ levels. We define the *skeleton* at $T$, denoted as $\mathcal{T}$, as the substructure consisting of all splitters (and thus interior nodes) in the first $\lambda$ levels. We use the samples to build the skeleton.

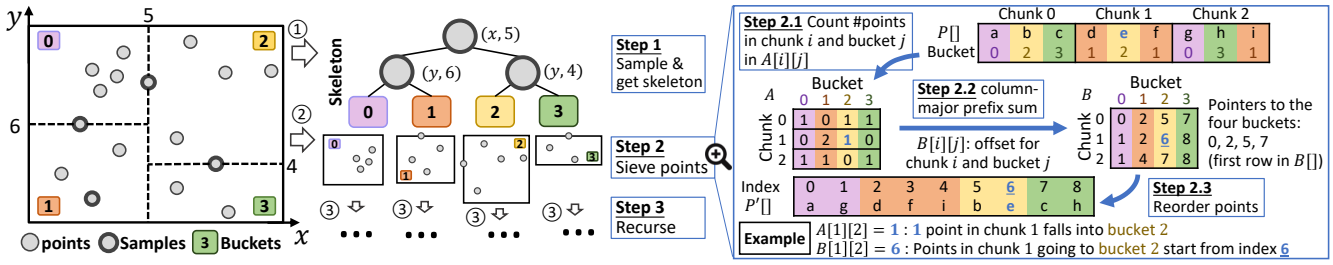


**Figure 1: An illustration of our $k$d-tree construction algorithm**, with a detailed overview on the *sieving step*.

In particular, we will uniformly take $2^\lambda \cdot \sigma$ samples from $P$, where $\sigma$ is the *over sampling rate*. In Sec. 3.2 we will show how to choose the parameter $\sigma$ to achieve a good theoretical guarantee. Let $S$ be the set of sample points. The skeleton will be the first $\lambda$ levels of the $k$d-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 plain parallel $k$d-tree construction algorithm at the beginning of Sec. 3.

The skeleton depicts the first $\lambda$ levels of the tree, and splits the space into $2^\lambda$ subspaces, corresponding to the external nodes (leaves) of the skeleton. We call each such external node a ***bucket*** in this skeleton. We label all buckets from 0 to $2^\lambda - 1$. 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 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 the I/O-efficient parallel sorting algorithm [27], 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 the next recursion receives a consecutive input slice. At a high level, this algorithm divides the array into chunks of size $l$, and processes them in parallel. We will first count the number of elements in chunk $i$ that fall 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 the offset matrix $B$. This can be done by the parallel I/O-efficient matrix transpose [15] and a standard parallel prefix-sum [35]. 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 Fig. 1. Then we process all buckets again in parallel, and move each point to their final destination by using the offsets provided from matrix $B$ 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 $k$d-trees for each bucket in parallel. The recursion stops when the number of points is smaller than $2^\lambda \cdot \sigma$. We then switch to the base case and use the plain parallel $k$d-tree construction to build the subtree. We will later show that by setting proper values for $\lambda$ and $\sigma$, the base case fits into cache and using the plain parallel construction will not incur extra I/O overhead.

**Algorithm 1:** Parallel $k$d-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.

**Function** BuildTree($P$)
  **if** $|P| < 2^\lambda \cdot \sigma$ **then** Use the plain parallel construction and **return**
  $S \leftarrow$ 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, 16, 57]. We put it here for completeness.

Lemma 3.1. *For a Pkd-tree $T$ with size $n'$, for any $\epsilon < 1$, setting $\sigma = (6c \log n)/\epsilon^2$ 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.* Alg. 1 guarantees that each leaf in a skeleton has at least $\sigma$ sampled points. Therefore, every time we find a splitter, it is the

median of at least $2\sigma$ sampled points. Let $s$ ($\geq 2\sigma$) be the number of samples for this Pkd-tree $T$, and $\Lambda \subset T$ 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 $\Lambda$ (i.e., the left side has fewer than $(1/2-\epsilon/4)\cdot n'$ points) is small. Since all samples are picked randomly, we denote indicator variable $X_i$, where $X_i = 1$ if the $i$-th sample is in $\Lambda$ and 0 otherwise. Let $X = \sum X_i$ for $i = 1..|\Lambda|$, and $\mu = \mathbb{E}[X] = (1/2-\epsilon/4)s$. Let $\delta = \epsilon/(2-\epsilon)$. Then $(1+\delta)\mu = (1+\frac{\epsilon}{2-\epsilon})(\frac{1}{2}-\frac{\epsilon}{4})s = \frac{s}{2}$. Using the form of Chernoff Bound $\Pr[X \geq (1+\delta)\mu] \leq \exp(-\delta^2\mu/(2+\delta))$, we have:

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

The right subtree has the same low probability to be unbalanced, so taking the union bound gives the state 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 $\sigma = 6c\log n$, 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 that the tree heights is $O(\log n)$ *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 $n \leq 4$. We will show that using $\epsilon = 4/\log n$ (i.e., $\sigma = O(\log^3 n)$), the tree height $h$ is $\log n + O(1)$. Similar to the above, here in the worst case, for a subtree of size $n'$, the children's subtree size is at most $(1/2+\epsilon/4)\cdot n' = (1/2 + 1/\log n)\cdot n'$. Hence, the tree height satisfies $(1/2+1/\log n)^h = 1/n$, so $h = -\log n/\log(1/2+1/\log n)$. Here, let $\delta = h - \log n = -\log n/\log(1/2+1/\log n) - \log n$. It solves to $\delta = O(1)$ for $n > 4$. Although complicated, the analysis primarily employs some algebraic methods. Due to the space limit, we put it in 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 computed the second and third derivatives and used a few algebraic techniques to remove them. □

Later, we will experimentally show that maintaining strong balancing criteria (tree height of $\log_2 n + O(1)$) is not necessary for most $k$d-tree's use cases. Hence, in 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 in work, span, and I/O, if we plug in the appropriate parameters. Recall that $M$ is the small memory size. We will set 1) skeleton height $\lambda = \epsilon\log M$ for some constant $\epsilon < 1/2$; and 2) chunk size $l = 2^\lambda$, so array $A$ and $B$ have size $O(2^\lambda \times |P|/l) = O(|P|)$, and operations on them will have $O(1)$ cost per input point on average. 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 $B$ is 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 14 and 21. 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 building a tree skeleton with $\lambda$ levels. Taking samples and building the skeleton on them can be trivially parallelized in $O(\lambda\log n)$ span (using the plain 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 16 can be computed in $O(\log n)$ span [15], 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 $M = \Omega(\textit{polylog}(n))$. 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 samples fully fits in the 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 on lines 13 and 20 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 16 [15]. 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) [4] and hence optimal. The span bound can also be optimized to $O(\log^2 n)$ *whp*, using a similar approach in [15], 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 $O(\log^2 n)$ 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 13 and 20. These two loops can be parallelized by a sorting-then-merging process as in [15]. The high-level idea is to first sort (instead of just count) the entries in the loop on line 13 based on the leaf labels. Once sorted, we can easily get the count of the points in each leaf. Then on line 20, once the array is sorted, points can be distributed in parallel. We refer the readers to [15] for more details. The span bound for each level is $O(\log n)$ [14, 15] for the sieving step. For the rest of the part, the span is $O(\lambda\log n)$ caused by skeleton construction. Altogether, the span per level is $O(\lambda\log n)$, and there are $O(\log n/\lambda)$ recursion levels *whp*. Therefore, the total span is $O(\log^2 n)$ *whp*. □

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 Pkd-tree. Here we consider the batch-parallel setting that inserts or deletes a batch of points $P$ to the

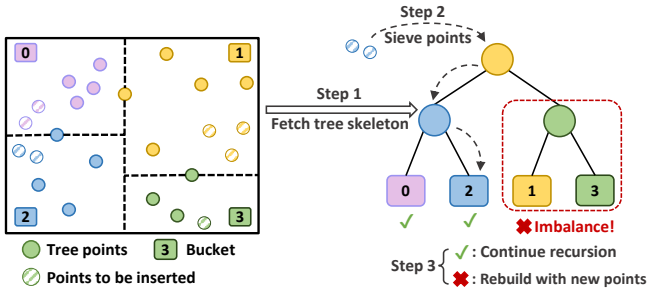


**Figure 2: Illustation of our batch insertion to a $k$d-tree**. We first sieve points into the corresponding bucket and rebuild the (sub)-tree if necessary.

current Pkd-tree $T$. Pkd-trees does not always keep the tree perfectly balanced as in existing parallel implementations [28, 43, 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 upon imbalance. Fig. 2 illustrates the high-level idea. We allow the sizes of the two subtrees to be off by at most a factor of $\alpha$, 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. The rebuild cost can be amortized to the updated points in all batches. This idea of lazy updates with reconstruction has been studied sequentially on various of trees for point updates [31, 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 for our new batch-update algorithm. In practice, we conduct an in-depth performance study with the relaxation of balancing criteria. In Sec. 6.4, we show that the query performance of Pkd-tree is mostly unaffected for $\alpha \leq 0.4$ (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 take as input a $k$d-tree $T$ and a set of points $P$, and insert the points in $P$ to (or delete $P$ from) $T$. The approach for our insertion and deletion functions are almost identical. In general, we partition the points in $P$ that belong 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 the update. If so, we will directly flatten all points in the current subtree with the points to be inserted, and 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 will 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 line 3 (i.e., removing points instead of adding them). For simplicity, we also assume that $P \cap T = \emptyset$ for an insertion and $P \subseteq T$ for a deletion

**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.

```
// Insert points P into kd-tree T
1  Function BatchInsert(T, P)
2      if P = ∅ then return T
3      if T is leaf then return BuildTree(T ∪ P)        // Insert into a leaf
4      𝒯 ← the skeleton at T
5      Sieve points to the corresponding bucket in 𝒯 using Sieve(P, 𝒯) from
         Alg. 1. Let R[i] be the sequence of all points sieved to bucket i.
6      t ← The root of skeleton 𝒯
7      return InsertToSkeleton(t, R[0..2^λ))
// 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 ← 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 ← number of buckets in t.lc
15             In Parallel:
16                 t.lc ← InsertToSkeleton(t.lc, R[l, m))
17                 t.rc ← InsertToSkeleton(t.rc, R[m, r))
18             return t
19         else return BuildTree(t ∪ (⋃_{i=l}^{r−1} R[i]))   // Rebuild the subtree
```

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 $k$d-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 belong 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 (inserting points already in $T$ and removing points not in $T$). In this case, directly using $|T|+|P|$ or $|T|-|P|$ as the tree size after updating cannot accurately determine the tree imbalance. There are several ways to handle this. A simple way is to maintain a set 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 parallel hash tables [64] or BSTs [13, 67] to maintain the set without affecting the asymptotic bounds. We use another 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

In this section, we will show that our conceptually-simple batch update algorithms also have good theoretical guarantees. Since the

update algorithms use the construction algorithm as a subroutine, we need to accordingly set up the parameters for both algorithms. In particular, we select $\sigma = (6c \log n)/\alpha^2$ for some constant $c > 0$ to ensure a low amortized cost in Thm. 4.1. Here we assume $\alpha$ is a constant and $\sigma = \Theta(\log n)$.

Theorem 4.1 (Updates). *Using $\sigma = (6c \log n)/\alpha^2$, a batch update (insertions or deletions) of a batch of size $m = O(n)$ on a Pkd-tree of size $n$ has $O(\log^2 n)$ 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.*

For simplicity, Thm. 4.1 assumes the batch size $m = O(n)$. If $m = \omega(n)$, we just need to replace the term $n$ by $m+n$ in the bounds for batch insertions (no change needed for batch deletion).

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 [13]. Again in practice, we use the sieve approach in Alg. 1, which leads to $O(M^\epsilon \log n)$ span and supports sufficient parallelism.

The update cost bound for Pkd-tree is higher than using the logarithmic method—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. 6, 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 better.

In addition, Pkd-tree can support stronger balancing criterion for by setting $\alpha = o(1)$. 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. For example, we can enable $\log n + O(1)$ tree height by using $\alpha = O(1)/\log n$. However as mentioned, our experimental results show that using tree height as $O(\log n)$ is good enough to give overall good performance for both updates and queries in practice.

## 5 Implementation Details

In this section, we discuss additional implementation details that contribute 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 24 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 $\lambda = \epsilon \log M$ for some constant $\epsilon < 1/2$. 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. 6.4.

**Handling of Duplicates.** One issue we observed in some existing $k$d-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. Pkd-tree uses a special *heavy leaf* to handle this. 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, a 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 to 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 can be 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 on the fly: the function will pass the subspace of the current tree node to recursive calls at its children, so the subspaces for the children can be further computed with the splitter hyperplane. This is not as tight as the bounding box, but in our experiments, we observed that avoiding explicitly storing bounding boxes gives better overall performance for Pkd-tree.

**Queries.** Since Pkd-tree is a single $k$d-tree, we can use all standard $k$d-tree query algorithms. The only strong query bounds we know of for the standard $k$d-tree are for orthogonal range queries and range counts. A range count on $D$ dimensions takes $O(2^{h(D-1)/D})$ work on a $k$d-tree of height $h$ [2, 16], which is $O(n^{(D-1)/D})$ if the tree height is $\log n + O(1)$. The bound for a range query has an additive term $k$ where $k$ is the output size. We can set the parameters of Pkd-tree accordingly to achieve this bound in theory, although later in Sec. 6.4 we show that the query performance does not degenerate by a slightly larger tree height. While no strong bounds are known for $k$-NN queries on $k$d-tree on general distributions, previous work has shown that $k$d-tree is highly practical for such queries, and it is the main use case for $k$d-trees in the real world. Our implementation also include $k$-NN queries.

**Parallel Granularity Control.** As standard parallel granularity control, for tree construction and batch update, when the input size is smaller than 1024, we will continue the process using the standard sequential algorithm.

## 6 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 TB RAM. Pkd-tree is implemented in C++ using ParlayLib [11] 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. We ran each experiment for 4 times and took the average of the last 3 runs. We set tree skeleton height $\lambda = 6$ as explained in Sec. 5. We set balancing parameter $\alpha = 0.3$, and

| Op. | Dims | Uniform-100M | | | | | Varden-100M | | | | | Uniform-1000M | | | | Varden-1000M | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 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. 6. The fastest runtime for each benchmark is underlined. "n.a.": not available. "s.f.": segmentation fault. "t.o.": time out (more than 3 hours).

discuss the choice of parameter in Sec. 6.4. Our code is available at [52].

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

- **Zd-tree** [12]. A parallel quad/octree based on Morton order (space-filling curve). Zd-tree maps each point to an integer by interleaving the coordinates' bits and uses integer sort as preprocessing. Due to integer-precision issues, Zd-tree's implementation only supports inputs in 2 or 3 dimensions.
- **BHL-tree** [70]. The single-tree version of the parallel $k$d-tree from ParGeo, using the binary heap layout. It supports parallel construction, but a batch update simply rebuilds the whole tree.
- **Log-tree** [70]. The $k$d-tree implementation based on the 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** [28]. The $k$d-tree in the open-source computational geometry library CGAL. CGAL supports 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 [12]. For fairness, we still use 192 hyperthreads when testing CGAL.

**Datasets.** We tested both synthetic and real-world datasets. We will introduce the real-world datasets in Sec. 6.2. For synthetic datasets, we use two distributions: `Varden` [32] and `Uniform`. We use `Varden` from [32], which is a skewed distribution that contains some very dense subareas in the dataset. `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.

## 6.1 Basic Operation Evaluation

**Overall Performance.** We summarize the performance for Pkd-tree and other baselines in Tab. 3. We tested tree 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 $10^8$. Since none of the other baselines supports range count query, so we test range count queries for Pkd-tree separately, and show the result later in this section and in Tab. 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, 8.2–15.9× than Log-tree, 8.1–13.2× than BHL-tree, and 52.1–401× than CGAL. All Pkd-tree, Zd-tree, Log-tree, and BHL-tree scale well with the increment of processors (see the scalability curve in Fig. 8). Pkd-tree achieves the best performance among them due to the better I/O efficiency of Alg. 1. As mentioned, CGAL has a known scalability issue [12] and cannot utilize 36 or more threads (also see the scalability curve in Fig. 8). For this reason, CGAL is much slower than others and we exclude it in 1000M tests as 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. Zd-tree is a quad/octree based on the space-filling curve, which handles multi-dimensional points as integers. Therefore, the computation is simpler than the $k$d-tree-based structures in construction and update (and is thus reasonable to achieve higher performance), but it cannot efficiently handle dimensions $D > 3$. Interestingly, Pkd-tree is still faster than

Zd-tree in all constructions and most batch updates. We believe this is due to the I/O optimization in Pkd-trees.

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 than BHL-tree, all in high dimensions. As mentioned in Sec. 5, 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 trades 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. For these reason, we do not maintain the bounding boxes in Pkd-tree, such that Pkd-tree achieves the best overall performance for both updates and queries.

Another interesting finding is that almost all $k$d-trees showed better performance on `Varden` than `Uniform` in $k$-NN queries. Since `Varden` contains some dense subareas, the $k$-NN of many query points are likely to fall into the same leaf, which yields more effective pruning than the uniform distributions.

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

**Batch Update.** To study the performance of batch updates, we design experiments to vary the batch sizes, and show the results of all $k$d-tree implementations in Fig. 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 a batch of points from the tree.

Our Pkd-tree shows the best performance for batch update on `Uniform`, while it is slightly slower than Zd-tree for batch insertion and small-size batch deletion on `Varden`. As mentioned above, one advantage of Zd-tree is that it handles the 2- or 3-dimensional points as integers, which is simpler and faster for updates. Such an advantage is more significant on the skewed distribution `Varden`, where Pkd-tree incurs more frequent reconstructions. Even so, Pkd-tree is still competitive. For batch insertion, the running time is still faster than Zd-tree on `Uniform` data, and is within 1.4–1.6× of Zd-tree on `Varden`. For batch deletion, Pkd-tree is still always faster on `Uniform` data, and also performs better than Zd-tree when the deletion batch is large—when the number of remaining points in the tree decreases, the partial reconstruction also becomes cheaper, and Pkd-tree can also outperform Zd-tree in this case.

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 merges a subset of the trees. One may notice that there is a jump for Log-tree in batch insertions. The reason is that for a 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 full 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 Fig. 4. All queries run in parallel. Pkd-tree is always among the fastest. Most other baselines are also competitive within 1.04–17.6×. The only exception is Log-tree: querying on all $O(\log n)$ $k$d-trees significantly slows down the performance. It also incurs high space usage, and was unable to run 100-NN query due to the out-of-memory issue.

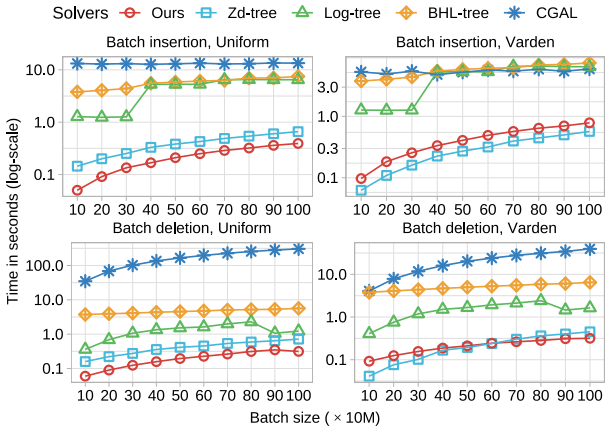


**Figure 3: Time required for batch update on points from `Varden` and `Uniform` with size 100M and dimension 3. Lower is better.** The batch size is the number of points in the batch. The time is measured in seconds and transformed into a logarithmic scale.

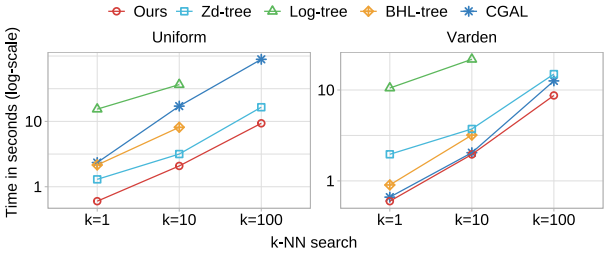


**Figure 4: $k$-NN time for all 100M points from `Varden` and `Uniform` in 3 dimensions. Lower is better.** The $k$ is set to be 1, 10 and 100. The time is measured in seconds and plots in log-scale. There is no data for Log-tree and BHL-tree when $k = 100$ due to out-of-memory.

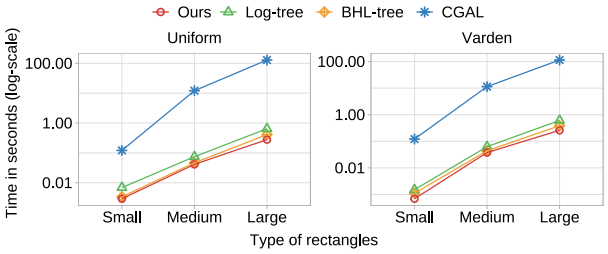


**Figure 5: Orthogonal range query for three rectangle types in parallel. Lower is better.** The benchmark contains 100M points in 3 dimensions. Each type of rectangle contains 10K candidates. The time is measured in seconds and shown in log-scale. Zd-tree did not implement range query.

**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 Fig. 5. Overall, Pkd-tree has the best performance. Most other baselines (except for CGAL) are competitive.

**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, 71, 74].

| Dims | Uniform | | | Varden | | |
|---|---|---|---|---|---|---|
| | 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.

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. Tab. 4 presents the results. Compared with the running time on range queries, a range count query is faster than reporting all elements in the range. For example, for 3D-U-100M and 3D-V-100M, a range count query is up to 3.0× faster and up to 3.8× faster than a range query, respectively.

## 6.2 Real-World Datasets

We evaluate the time for tree construction and 1-NN, and 10-NN query for all-point, on real-world datasets described below. Our benchmarks include very large datasets COSMOS (CM) [61] and OpenStreetMap (OSM) [38] that contains up to 2.7 billion points, and high-dimensional datasets HT [40] and HouseHold (HH) [39] with up to 10 dimensions. We also test the benchmark GeoLifeNoScale (GLNS) [75] that contains highly duplicated points. All coordinates are 64-bit real numbers. Tab. 5 demonstrates the results.

Pkd-tree shows the best performance in all but two instances. CGAL is 2.15× faster than Pkd-tree 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. For the same reason, Zd-tree performed well in construction on GLNS (1.9× faster than Pkd-tree) but extremely poorly in queries: with heavy duplicates, their approach based on the space-filling curve may map many points to the same leaf, resulting in a large 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 $k$d-tree, BHL-tree, is the most competitive to Pkd-tree on real-world datasets, but it is still 1.9–8.6× slower in construction and 1.4–28× slower in query than Pkd-tree.

## 6.3 Technique Analysis for Tree Construction

In the tree construction Alg. 1, we mainly employed two techniques to improve I/O efficiency: 1) building $\lambda$ levels at a time to save total data movement, and 2) using sampling to determine the splitters to save memory accesses. We measure the performance of the above techniques in tree construction by testing four versions of Pkd-tree with different levels of optimizations, and present the results in Fig. 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

| | 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. Lower is better.** 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. 6. 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.

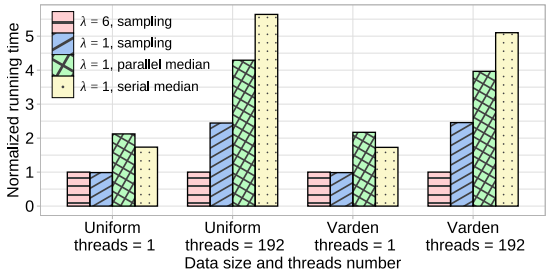


**Figure 6: Measuring the effectiveness of techniques in tree construction. Lower is better**. The test is categorized by datasets ($n$ = 100M, $d$ = 3) and number of threads used. The vertical axis is the running time for different technique combinations normalized to the default one (use tree skeleton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

to determine the splitter (yellow bar). We tested on both Uniform and Varden, using 1 thread and 192 threads, respectively. Comparing the blue bar with the green bar (or the blue bar with the yellow bar for the sequential setting), we can observe that sampling always improves the performance by about 1.6×. However, comparing the red and the blue bar indicates that building multiple levels together significantly improves 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 the performance.

## 6.4 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 $\alpha$. In this section, we systematically study the choice of $\alpha$ and the corresponding impact to the performance.

To do this, we create adversarial input such that belated rebalancing may result in a large tree height. We generate skewed batch

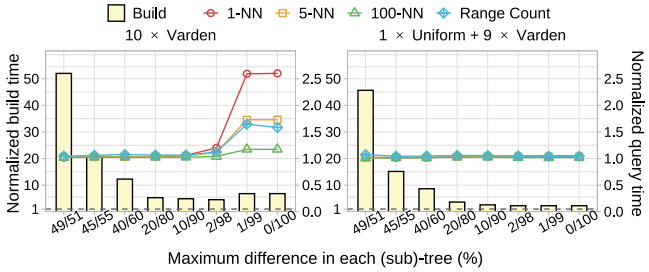


**Figure 7: Construction and query time with varying balancing parameter $\alpha$. Lower is better.** The input is the concatenation of 10 datasets each with 10M 3D points. The bars and the left axis show the time to build the tree by dividing inputs into 100 batches and inserting them into the tree one by one. The time is normalized to that of building the tree directly from the 100M inputs. The lines and the right axis show the running time on various queries on the final tree, normalized to the query time on the tree constructed directly from the 100M inputs.

sequences and insert them into an empty tree by 100 batches, after which we perform queries on the tree to see how the imbalance affects the query performance. We test $\alpha$ 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.

In particular, we generated multiple adversarial distributions for Pkd-tree, and TYPE I is the one where the query is affected the most, so we pick it here and compare it to TYPE II, which is a normal case.

Fig. 7 presents the construction and $k$-NN time wrt. balancing parameter $\alpha$. The "build time" is to construct a tree by inserting 100 batches incrementally. We normalize the construction time to that if we directly build the tree $T^*$ once using the points in all batches. The "query time" is to perform queries on the final constructed tree. We normalize it to if we 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 balance controlled by $\alpha$, 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 unbalanced. This is because the cost of traversing the tree in batch insertion also increases when the tree grows skewed. For queries, an unbalanced tree poses a more negative influence on $k$-NN search with a smaller value of $k$ than the larger ones. This is because when $k$ becomes large, the main cost will be dominated by writing $k$ elements to output, and the tree imbalance will not be as significant.

Overall, the query performance seems 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. However, allowing relaxed balancing criteria greatly improved the update performance: if we allow a 20/80 imbalance on subtrees, then 100 batch updates can be 9.9× more efficient than if the ratio of subtrees sizes has to be within 49/51. Based on these observations, we choose $\alpha = 0.3$ in our implementation (20/80) to achieve a tradeoff between the balance of the tree and update performance.

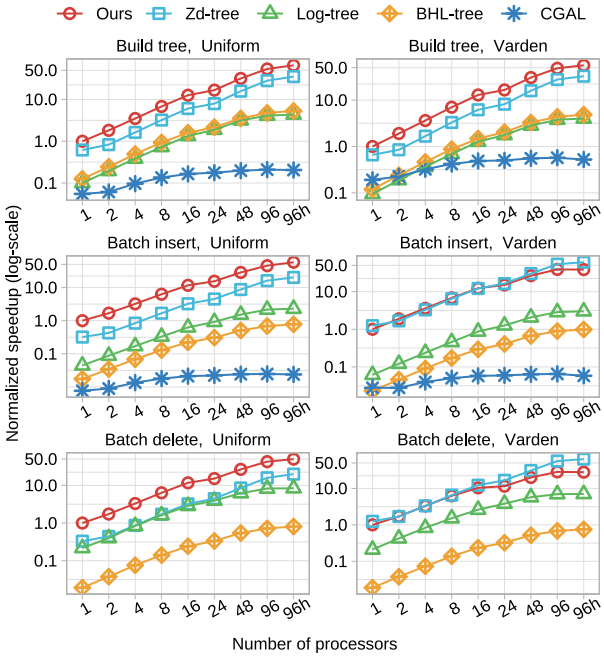


**Figure 8: Normalized speedup of basic operations over `Uniform` and `Varden` for Pkd-tree and other baselines on varying numbers of processors. Higher is better.** The curves show relative running time on different number of threads normalized to Pkd-tree on one thread. The benchmark contains 100M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

### 6.5 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 Fig. 8. Generally, our Pkd-tree overall has very good scalability. For `Uniform`, our Pkd-tree reaches 66× self-relative speedup on construction, 58× on batch insertion and 50× on batch deletion on our 96-core machine. The numbers for `Varden` is 56× on tree construction, 39× on batch insertion, and 28× on batch deletion. The speedup on `Varden` is lower since the skewed distribution for `Varden` triggers more reconstruction than in the `Uniform` case. Indeed, Zd-tree, Log-tree, and BHL-tree have also shown good parallel speedup, and the performance difference is mainly due to the sequential (1 core) performance. The only implementation that does not scale well is CGAL, which was observed in previous work as well [12].

## 7 Related Work

In this section, we review the literature of the $k$d-tree, with a summary of the most relevant ones in Tab. 6. The original algorithm proposed by Bentley et al. [7, 29] 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 $k$d-trees, mainly in the two categories: *logarithmic method* ("log-tree" in Tab. 6), and *partial rebuild*. The logarithmic method was first proposed by Bentley in [8], and has been followed up by later work, including optimizing the I/O costs [2, 56] and parallelism [70]. However, as shown in Tab. 3, maintaining $O(\log n)$

| | | | Layout& Update | Rebalancing | I/O Optimizations | Notes&More Optimizations |
|---|---|---|---|---|---|---|
| **1975** | **Bentley [7]** | Seq. | Single tree; | No rebalance | - | • Original kd-tree paper |
| **1978** | **Bentley [8]** | 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 [28]** | Par. | Single tree; Full rebuild (sequential deletion) | Perfect | • Leaf wrap | • "CGAL" tested in Sec. 6<br>• Dimension choosing<br>• Open-source library |
| **2003** | **Bkd-tree [56]** | Seq. | Log-tree; Tree merging | Perfect | • I/O-efficient construction<br>• I/O-efficient point update | - |
| **2003** | **Arge et al. [2]** | Seq. | Log-tree; Tree merging | Perfect | • I/O-efficient construction<br>• I/O-efficient point update | - |
| **2016** | **Agarwal et al. [1]** | Dist. | Single tree; Static | Relaxed (randomized) | - | • Sampling<br>• Multi-level construction |
| **2021** | **ikd-tree [21]** | Seq. | Single tree; Partial rebuild (lazy deletion) | Relaxed (weight balanced) | - | - |
| **2021** | **ParGeo [70]** | Par. | Log-tree; Tree merging (lazy deletion) | Perfect | • Leaf wrap<br>• vEB layout | • "Log-tree" tested in Sec. 6<br>• Open-source library |
| **2021** | **ParGeo [70]** | Par. | Single tree; Full rebuild | Perfect | • Leaf wrap<br>• Binary heap layout | • "BHL-tree" tested in Sec. 6<br>• Open-source library |
| **2021** | **Zd-tree [12]** | Par. | Quad/octree; Rotations | Balanced BST | • Leaf wrap | • "Zd-tree" tested in Sec. 6<br>• Open-source library |
| | **This paper** | Par. | Partial rebuild | Relaxed (weight balanced, randomized) | • Leaf wrap<br>• I/O-efficient construction<br>• I/O-efficient batch update | • Sampling<br>• Multi-level construction<br>• Dimension choosing<br>• Open-source library |

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

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 [31], ikd-tree [21], and the divided $k$-d tree [69]. 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 $k$d-trees. Earlier ones simply consider flattening 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. [70] 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 exist parallel $k$d-tree algorithms, but most of them are not I/O-optimized or do not support a full interface (other than ParGeo). Static $k$d-tree construction is widely studied, using techniques such as randomization and space-filling curves [1, 5, 20, 22, 25, 41, 58, 63, 72]. 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 [28] has the interface for updates, it simply rebuilds the tree after updates, and the performance overhead can be significant. There also exist concurrent $k$d-trees [23, 42] that achieve linearizability and lock-freedom. Our work focuses on batch-parallel setting, which aims to support a batch of insertions or deletions with good work, span, and I/O bounds.

In this paper, we also compare Pkd-trees to Zd-trees [12], which is a recent parallel quad/oct-tree implementation based on the Morton order. Compared to $k$d-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, and can be slower in queries on skewed data.

## 8 Conclusion

We present Pkd-tree, a parallel $k$d-tree that has strong theoretical guarantee in work, span, and I/O cost for tree construction and batch update, as well as high performance in practice. Our main techniques include using sampling, multi-level construction, the sieving algorithm, and the weight-balance scheme to holistically optimize the work, span, I/O in both constructions and updates. In this way, our approach relaxes the balancing criteria by a controllable manner, which allows for overall good performance considering construction, update, and various queries. In our experiments, Pkd-tree achieves the best performance in construction and update among all tested $k$d-tree implementations, with competitive or better query performance. It also outperforms Zd-tree, a quad/octree for low-dimensional data points, in most of the construction and batch update tests, and is always faster in queries.

## 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] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.
[8] Jon Louis Bentley. 1979. Decomposable searching problems. *Inform. Process. Lett.* 8, 5 (1979), 244–251.
[9] Jon Louis Bentley. 1990. Experiments on traveling salesman heuristics. In *Proceedings of the first annual ACM-SIAM symposium on discrete algorithms*. 91–99.
[10] Jon Louis Bentley. 1990. K-d trees for semidynamic point sets. In *Proceedings of the sixth annual symposium on Computational geometry*. 187–197.
[11] 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.
[12] 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.
[13] 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)*.
[14] 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.
[15] 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)*.
[16] 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)*.
[17] 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.
[18] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).
[19] 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.
[20] Russell A Brown. 2014. Building a balanced kd tree in o (kn log n) time. *arXiv preprint arXiv:1410.5420* (2014).
[21] Yixi Cai, Wei Xu, and Fu Zhang. 2021. ikd-tree: An incremental kd tree for robotic applications. *arXiv preprint arXiv:2102.10808* (2021).
[22] 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.
[23] Bapi Chatterjee, Ivan Walulya, and Philippas Tsigas. 2018. Concurrent linearizable nearest neighbour search in lock free-kd-tree. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*. 1–10.
[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] 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)*.
[28] 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.
[29] 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.
[30] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.
[31] Igal Galperin and Ronald Rivest. 1993. Scapegoat Trees.. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 93. 165–174.
[32] 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.
[33] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.
[34] 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.
[35] 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.
[36] 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.
[37] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3 (1994), 357–399.
[38] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive computing* 7, 4 (2008), 12–18.
[39] Georges Hebrail and Alice Berard. 2012. Individual household electric power consumption. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58K54.
[40] 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.
[41] 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.
[42] Jeffrey Ichnowski and Ron Alterovitz. 2020. Concurrent nearest-neighbor searching for parallel sampling-based motion planning in SO (3), SE (3), and euclidean spaces. In *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer, 69–85.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. 2003. The global k-means clustering algorithm. *Pattern recognition* 36, 2 (2003), 451–461.
[48] 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.
[49] 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.
[50] Leland McInnes and John Healy. 2017. Accelerated hierarchical density based clustering. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 33–42.
[51] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2024. Parallel kd-tree with Batch Updates. https://github.com/ucrparlay/KDtree/blob/main/full-version-paper.pdf. (2024).
[52] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2024. Pkd-tree: a parallel kd-tree with batch update. https://github.com/ucrparlay/KDtree.
[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 Joã o 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] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[68] 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.

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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 $k \to 1^+$, 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 *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 $\alpha$ 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 [13], so the amortized I/O per point is $O(\log(n/m))$. Putting both cost together gives the stated I/O bound. □