Parallel Dynamic Spatial Indexes

Ziyang Men
ziyang.men@email.ucr.edu
UC Riverside

Bo Huang
bo.huang@email.ucr.edu
UC Riverside

Yan Gu
ygu@cs.ucr.edu
UC Riverside

Yihan Sun
yihans@cs.ucr.edu
UC Riverside

Abstract

Maintaining spatial data (points in two or three dimensions) is crucial and has a wide range of applications, such as graphics, GIS, and robotics. To handle spatial data, many data structures, called *spatial indexes*, have been proposed, e.g., *k*d-trees, oct/quadtrees (also called Orth-trees), R-trees, and bounding volume hierarchies (BVHs). In real-world applications, spatial datasets tend to be highly dynamic, requiring batch updates of points with low latency. This calls for efficient parallel batch updates on spatial indexes. Unfortunately, there is very little work that achieves this.

In this paper, we systematically study parallel spatial indexes, with a special focus on achieving high-performance update performance for highly dynamic workloads. We select two types of spatial indexes that are considered optimized for low-latency updates: Orth-tree and R-tree/BVH. We propose two data structures: the P-Orth tree, a parallel Orth-tree, and the SPaC-tree family, a parallel R-tree/BVH. Both the P-Orth tree and the SPaC-tree deliver superior performance in batch updates compared to existing parallel *k*d-trees and Orth-trees, while preserving better or competitive query performance relative to their corresponding Orth-tree and R-tree counterparts. We also present comprehensive experiments comparing the performance of various parallel spatial indexes and share our findings at the end of the paper.

1 Introduction

Spatial data widely appear in geographic information systems (GIS), spatial databases, computer graphics, robotics and its planning, and many other domains. Efficiently processing such geometric objects (usually points) in two or three dimensions is of great importance, for both maintenance (construction, insertion, deletion) and queries (range queries, nearest-neighbor queries, etc.).

Given the wide applicability, many well-known data structures (usually called "spatial indexes") have been proposed to handle spatial data, such as *k*d-trees[12], oct/quadtrees[26] (collectively referred to as *orth-trees*), range trees [13], R-trees [33], and bounding volume hierarchies (BVHs) [4]. Spatial indexes typically organize points as a tree, with each subtree corresponding to a subspace (not necessarily non-overlapping). The bounding boxes of the subspaces can be used to prune subtrees during queries. For instance, consider a nearest-neighbor search: when the search reaches a subtree, if its sub-region is farther from the query point than the current nearest neighbor, the subtree can be pruned. Despite maintaining different invariants, all these trees share the same high-level intuition: skip most of the objects in queries by pruning, leading to efficient query performance.

Real-world applications can involve highly dynamic data, and updates may be latency-sensitive or throughput-sensitive. For example, in 3D games, moving objects must be reflected quickly to affect lighting and collision detection, whereas GIS applications often ingest high-volume sensor streams where total update throughput is critical. In both scenarios, updates frequently arrive in batches and must be incorporated into the index promptly. To handle both updates and queries efficiently, different spatial indexes offer different trade-offs. Traditionally, *k*d-trees are considered highly efficient for queries due to their strongest invariant (splitting at object medians), but updates are costly. Orth-trees offer competitive query performance and faster updates due to their simpler invariant (splitting at spatial medians). R-trees/BVHs encompass a large family of solutions; they usually provide the simplest and fastest updates but slower queries.

With the ever-growing data volume, *parallelism* becomes essential in designing efficient data structures. Unfortunately, little work is known on parallel spatial indexes with batch updates. In the two famous libraries, CGAL [25] and Boost [51], most spatial indexes are sequential. The only exception is CGAL's *k*d-tree, but it has known scalability issues [17, 43]. Parallel construction for range trees was described in [57], but it does not support batch updates. In 2022, Blelloch and Dobson [17] proposed Zd-tree, the first parallel quadtree. The idea is to leverage the Morton curve, a space-filling curve (SFC) that maps 2D or 3D points to 1D integers, and use this information to facilitate construction and batch updates. However, Zd-trees are slower than the parallel *k*d-tree (the Pkd-tree), proposed later [43], despite better theoretical bounds for updates ($O(\log n)$ vs. $O(\log^2 n)$ per updated point). We believe the main reason is the I/O (cache) optimizations in Pkd-tree. More interestingly, despite R-trees/BVHs having the simplest structure, we are aware of little work on parallel batch updates for them. Indeed, most existing approaches are either based on single insertions/deletions [7, 15, 33, 51, 54], or fully rebuilding the tree upon updates [31, 46, 59]. The only relevant work [49] uses the logarithmic method, which can substantially slow down the query time. Hence, it is natural to ask whether Orth-trees and R-trees/BVHs can still leverage their strengths for highly dynamic workloads in the parallel setting. In particular, we investigate whether they can achieve much faster construction and batch updates (with better theoretical guarantees) than *k*d-trees in parallel, while preserving query performance as in their sequential counterparts.

In this paper, we systematically study parallel spatial indexes, with a special focus on achieving high-performance updates in highly dynamic workloads. We propose two new (families of) data structures: **P-Orth trees** and the **SPaC-tree family**. We integrate these data structures into a library called the *Parallel Spatial Index Library* [9], abbreviated as PSI-Lib or Ψ-Lib.

We first show our design for a parallel Orth-tree called the P-Orth tree. Almost all existing Orth-trees [17, 39, 40, 46, 61] use space-filling curves (SFCs) to accelerate construction and updates. However, simply computing and sorting the SFC codes of the points already requires several passes of reading and moving all data, which is time-consuming. In this paper, we present the design of P-Orth trees that does not use SFCs. We borrow the idea of the *sieving algorithm* from the Pkd-tree [43], which directly reorders the points while constructing or inserting them into the tree, so that achieving I/O-efficient construction and batch updates for Orth-tree. Conceptually, our algorithms are equivalent to integer-sorting SFC codes, but without generating, storing, or using them. We believe the algorithmic idea is interesting, and refer readers to Sec. 3 for algorithmic details and analysis.

Our next question, then, is whether SFCs are still useful spatial indexes. As mentioned, SFCs have been used in both Orth-tree and R-trees/BVHs [17, 31, 38–40, 46, 50, 58, 61]. However, we are unaware of any implementations with update performance competitive with Pkd-tree and P-Orth trees, mostly due to limited or no parallel support.

In this paper, we propose the SPaC-tree family, which supports extremely fast updates (as R-trees are supposed to) while maintaining query performance competitive with existing R-trees/BVHs. To achieve this, our backbone is the PaC-tree[24], a parallel balanced binary search tree. The key insight of PaC-trees is to use join-based algorithms[2, 3, 18, 57] to efficiently rebalance during parallel updates, and to use leaf blocking (maintaining 16–32 objects in each leaf in a flat array) to improve cache locality. To support spatial queries, a simple approach is to store points using their SFC codes as keys in a PaC-tree and augment each tree node with bounding boxes. However, this plain adaptation yields poor update speed (up to 3.5× slower than Pkd-tree; see the columns "CPAM-H" and "CPAM-Z" in Fig. 3). We observe that the main bottleneck is maintaining the SFC-induced *total* order over all points in PaC-trees. To address this challenge, we carefully redesign the join-based algorithms in PaC-trees to maintain spatial data under only a *partial order*. We provide details in Sec. 4. We refer to our design as the Spatial PaC-tree, or SPaC-tree for short. In Ψ-Lib, we adopt both Morton curves (SPaC-Z-tree) and Hilbert curves (SPaC-H-tree).

Our P-Orth trees and SPaC-trees are backed by strong theoretical support. We show that the update cost per object is $O(\log n)$ for a SPaC-tree and $O(\log \Delta)$ for a P-Orth tree (Δ is the aspect ratio, see Sec. 3.3), which is much stronger than $O(\log^2 n)$ for a Pkd-tree. Our batch updates achieve polylogarithmic span, indicating strong and scalable parallelism.

We tested Ψ-Lib on workloads with various input distributions, query distributions, query types, and update patterns. We compare Ψ-Lib with existing parallel and sequential baselines including Pkd-trees, Zd-trees, etc. Our experiments simulates both a static setting and a highly dynamic setting where updates are consecutively applied to an initial tree. This setting better reflects the capability of each data structure to handle highly dynamic workloads, especially showcases whether and how the index quality are affected under a progressively evolving dataset. With our new algorithms, both P-Orth tree and SPaC-tree achieved superior construction and update performance, while preserving comparable query performance to regular Orth-tree and R-trees. P-Orth tree is almost always the fastest on uniformly distributed data in construction and queries, and is close to the best on updates. SPaC-tree supports extremely fast parallel batch updates—it can be 2-6 times faster than Pkd-trees, and is especially good for skewed distribution of input points, queries, or insertion/deletion orders. With comprehensive experiments, we share our findings in Sec. 5.4, and visualize the query-update tradeoff of each parallel spatial index in Fig. 6. Our anonymous code is available at [9].

2 Preliminaries and Related Work

Throughout the paper, we use n to denote the input size or the tree size. We use the $\log n$ notation to denote the $\log_2(n+1)$ logarithm. When representing arrays, we shorten $A[l], A[l+1], ..., A[r-1]$ as $A[l, r)$.

2.1 Computational Models

We consider the shared-memory multiprocessor setting with the classical fork-join paradigm with binary forking [8, 19, 22]. Each computational thread is a sequential Random Access Machine (RAM) augmented with a fork instruction that spawns two child threads executing in parallel, with the parent thread resuming upon completion of both children. Parallel for-loops are efficiently simulated through logarithmic levels of forking. When analyzing algorithms, we use the work-span model, where the work is the total number of operations in the algorithm and the span is the longest dependence chain in the parallel computation. Using randomized work-stealing schedulers, a computation with work W and span S executes in $W/\rho + O(S)$ time with high probability (in W) on ρ processors [8, 22, 32].

We use the ideal-cache model [27] to analyze the I/O cost of our algorithms. In this model, memory is divided into two levels: a fast memory (cache) of size M and an arbitrarily large slow memory. The CPU can only access data in the fast memory (at no cost), and data is transferred between the two levels in blocks of size B. Each block transfer incurs unit cost. The cache is fully associative, and the optimal offline cache replacement policy is used. The cache complexity of an algorithm is measured by the number of block transfers between the two levels of memory during its execution.

2.2 Spatial Data

In this paper, we study points in Euclidean space $\mathbb{R}^D$ for $D = 2$ or 3, although the proposed techniques can generalize to shapes and any constant integer $D > 1$.

Queries on Spatial Data. To benchmark the quality of spatial indexes, we use standard k-NN queries and range queries. A k-nearest neighbor (k-NN) query takes a set of points P and a query point q as input, and returns the k-closest points to q in P. A range query takes a set of points P and an axis-aligned rectangle subregion r. The range-count query returns the number of points in P within r, and the range-list query returns all points within r.

Spatial Filling Curves. A spatial filling curve (SFC) embeds multidimensional points into a one-dimensional sequence. In Ψ-Lib, we use *Z-curve (Morton-curve)* and *Hilbert-curve*, illustrated in Fig. 1. Both of them encode each point as an integer, which determines the point's order along the curve. For integer coordinates, both Hilbert- and Z-curve can be computed in a constant time. SFCs are widely used to facilitate spatial indexes [17, 31, 38, 40, 46, 50].

2.3 Existing Commonly-Used Spatial Indexes

Space-Partitioning Trees: Orth-trees and kd-trees. In space-partitioning trees, each node represents a subspace. All of its children form a non-overlapping partition of that subspace, usually by axis-aligned partition hyperplanes, i.e., a splitting dimension d and a coordinate x. Space-partitioning trees thus differ in how they select partition hyperplanes.

As typical examples, a kd-tree [12] chooses the median coordinate in the splitting dimension across all points, and thus always yields a balanced partition into two subtrees. An orth-tree in D dimensions partitions the space into 2^D subspaces evenly using the midpoint in each dimension (and is therefore a 2^D-ary tree). Specifically, an orth-tree is called a quadtree [26] in 2D and an octree in 3D [36].

There are parallel versions of both kd-trees and Orth-trees. Blelloch et al. [17] proposed a parallel Orth-tree called Zd-tree, which uses Morton curve to facilitate construction and updates. Yesantharao et al. [62] proposed two parallel kd-trees, BHL-tree and Log-tree. Only Log-trees support efficient parallel batch updates, using the *logarithmic method*, i.e., it maintains $O(\log n)$ trees with sizes 1, 2, ... $n/2$, such that a batch update can be broken down into at most $O(\log n)$ tree reconstructions. However, this method can greatly slow down queries [43]. A recent work proposed the Pkd-tree [43] that avoids logarithmic method, and achieves optimal work and cache complexity for parallel construction and batch updates. The underlying idea is to use sampling to approximate the object median, together with the *sieving* algorithm to partition points in an I/O-efficient manner. Our P-Orth trees also borrow this idea; see Sec. 3 for details. However, the Pkd-tree requires $O(m \log^2 n)$ work to update a batch of size m. We will show how Ψ-Lib achieves better bounds.

Object-Partitioning Trees: R-Trees/BVHs. In the object-partitioning trees, the objects (points) in each (sub)tree are partitioned into disjoint subsets, and each subset corresponds to a child node and is built recursively. Each tree node typically stores a bounding box (or a bounding volume in 3D) that is the smallest enclosing axis-aligned region of all objects in its subtree. Though named differently—R-trees in databases and usually in 2D ("R" for rectangle), and bounding volume hierarchies (BVHs) in graphics and usually in 3D ("V" for volume)—they share the same underlying concept. For simplicity, we use the term "R-tree" to refer to the general idea of object-partitioning trees. They can be either binary [15, 31, 60] or have a larger branching factor [30, 33, 38, 51], and can be built either offline [31, 46, 51, 59] or incrementally (thus supporting updates) [7, 15, 33, 38, 51, 54].

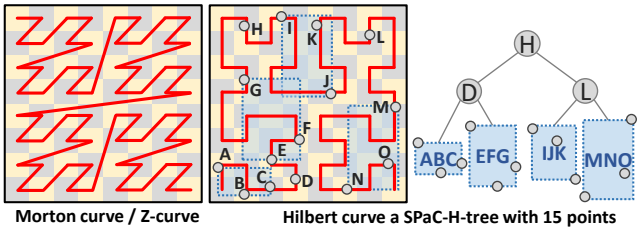

Figure 1. Space-filling curves and an example of a SPaC-tree with 15 points and size-3 leaf wrapping. Each leaf in this case has 3 points and its bounding box marked in blue.

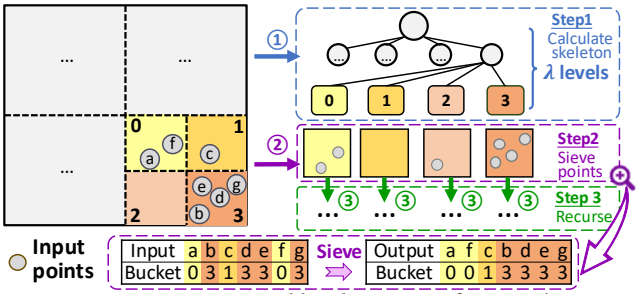

Figure 2. Construction and batch insertion for P-Orth trees.

To our knowledge, the only parallel R-tree with batch updates is by Qi et al. [50], which uses the logarithmic method. However, as noted, the logarithmic method significantly slows down queries and is therefore non-ideal. There also exist lock-based concurrent R-trees [23, 44].

3 The Parallel Orth-tree (P-Orth Tree)

In this section, we introduce our design of the Parallel Orth-tree (P-Orth tree), which partitions points into nested regions recursively based on the spatial median.

Previous algorithms. The naïve approach to construct or update an Orth-tree is to distribute the points to subtrees level by level from the root until reaching the leaves [26, 36]. However, this approach is slow because the number of rounds of global data movement is proportional to the tree height, which can be large. Hence, almost all subsequent Orth-trees [17, 39, 40, 46, 61] use SFCs, specifically the Morton curve (see Fig. 1), to speed up the algorithm. The high-level idea is to sort all input points in Morton order, which only requires $O(\log_M n)$ rounds of global data movement, where M is the cache size. Then, since Orth-trees always

Algorithm 1: Parallel Orth-tree (P-Orth tree) construction

Input: A sequence of points P, region box H.
Output: A P-Orth tree T on points in P.
Parameter: λ: the height of a tree skeleton.
ϕ: the leaf wrap of the kd-tree.

```
1  Function BuildOrth(P, H)
2      if |P| < φ then
3          return A leaf node with points P and its bounding box
4      Build the tree skeleton T by constructing the first
         λ = (log_2 M)/2D levels based on H
5      B[] ← Split H based on T    // B[i]: the sub-region for bucket i
       // Reorder points to make those in the same bucket consecutive
6      R[] ← Sieve(P, T)    // R[i]: the slice for all points in bucket i
7      parallel-foreach external node i of T do
8          t ← BuildOrth(R[i], B[i])                    // Recursive build
9          Replace the external node i with t
10     Compute the bounding boxes for all internal nodes in T, and
         merge non-leaf subtrees with sizes no more than φ
11     return The root of T
```

partition at the spatial median, a binary search on the sorted values can identify the partition hyperplane, and all points in one subtree also form a consecutive range in Morton order. Blelloch and Dobson, in their Zd-tree paper [17], also use this idea to achieve a parallel Orth-tree.

Issues on Existing Works. Although the long-standing Morton-based approach achieves good work, span, and cache bounds, a closer look reveals two major drawbacks.

- **Performance.** This approach must additionally compute the Morton code for each point as preprocessing and sort the ⟨code, point⟩ pairs. This increases memory footprint and induces more rounds of reads and writes to all data, which leads to significant overhead (see "Zd-tree" in Fig. 3).
- **Applicability.** While SFCs map higher-dimensional data into one dimension, they suffer from precision limitations. Most modern machines use 64-bit words, which suffices for 2D data (32-bit precision per dimension). However, 3D support is constrained to 21 bits per dimension, and handling higher dimensions ($D > 3$) is mostly infeasible. Even in lower dimensions, a fallback to the naïve partition-based solution is needed when precision is exhausted in certain subregions, which is not elegant.

Our Solution. To overcome these issues, our P-Orth tree design entirely avoids SFCs. We show that the sorting-based idea can be implemented conceptually equivalently without using SFC. Consequently, our P-Orth tree is fast and flexible to any coordinate types and ranges (not necessary integers). Theoretically, the P-Orth tree achieves strong bounds for both construction and batch updates. Practically, P-Orth trees outperform Pkd-trees and Zd-trees in almost all cases, except for very skewed distributions; see Sec. 5 for details. Below, we present our construction algorithm in Sec. 3.1, update algorithm in Sec. 3.2, and cost analysis in Sec. 3.3.

3.1 P-Orth Tree Construction

Our idea for P-Orth tree construction is to coordinate the "conceptual" sorting process together with the tree construction. The goal is to build λ levels of the tree at once with one round of data movement, and at the same time achieve high parallelism. Here we adopt the Sieve$(P, \mathcal{T})$ function from [43], which distributes the point set P based on a λ-level *tree skeleton* $\mathcal{T}$ in parallel. At a high level, our algorithm is equivalent to integer sort on Morton codes, on the λD most significant bits in each round. However, no codes needed to be computed, stored, or compared. Note that Sieve() is also used in the update algorithms. We show our P-Orth tree construction in Alg. 1 and illustrate it in Fig. 2.

Alg. 1 has three steps as shown in Fig. 2. The first step (lines 4–5) builds a tree skeleton $\mathcal{T}$ with $\lambda = (\log_2 M)/2D$ levels, where M is the cache size. This ensures that the number of leaves (external nodes) of $\mathcal{T}$ is $2^{\lambda \cdot D}$ fits into the cache. In theory, this step can be done in parallel, although given the small amount of work, in practice this step is run sequentially. Note that computing the $\mathcal{T}$ requires the bounding region for the current subtree, so we also need to compute the corresponding sub-regions for all $\mathcal{T}$'s leaves (line 5). Once $\mathcal{T}$ is built, the second step is to sieve the points in P to $\mathcal{T}$'s leaves. This step is implemented by the Sieve() function shown on line 6. The illustration of this step is shown in Fig. 2, and after that, all points in the same leaf of $\mathcal{T}$ are gathered together, conceptually stored in an array $R[]$.

Sieve() is implemented by dividing P into chunks of size l, then counting the number of points in each leaf in $\mathcal{T}$ in parallel. Then a matrix transpose is performed to compute the offsets for each leaf in each chunk, and finally, all points are distributed to the final destination in parallel. The final step is to recursively build the Orth-tree for each leaf (subtree) in parallel (line 8). Once finished, we update the bounding boxes of all internal nodes in $\mathcal{T}$ (line 10) and return the root of the skeleton (line 11).

Algorithm 2: Batch insertion for P-Orth tree

Input: A sequence of points P, a P-Orth tree T with region H.
Output: A P-Orth tree with P inserted.
Parameter: λ: the maximum height of a fetched tree skeleton.

```
// The deletion is symmetric.
1  Function BatchInsertOrth(T, P, H)
2      if P = ∅ then return T
3      if T is a leaf then                          // Insert into a leaf
4          return BuildOrth(T ∪ P, H)
5      T ← Retrive the skeleton at T
6      B[] ← Split H based on T    // B[i]: the sub-region for bucket i
7      R[] ← Sieve(P, T)          // R[i]: the slice for points in bucket i
8      parallel-foreach external node i for T do
9          t ← BatchInsertOrth(i, R[i], B[i])// Recursive insertion
10         Replace the external node of T with t
11     Update the bounding boxes of all affected nodes in T
12     return The root of T
```

3.2 Batch Updates for P-Orth Trees

Both batch insertion and deletion for P-Orth trees closely resemble the construction algorithm. Here we first introduce the batch insertion algorithm, given in Alg. 2, and discuss the deletion algorithm later.

The batch insertion algorithm takes a batch of points P, and adds them to an existing P-Orth tree T. To do so, we sieve the points also for λ levels, and then recursively insert points to each bucket in parallel. One can almost see a one-to-one mapping for these three steps in Fig. 2 and Alg. 2, except for some minor differences in handling base cases. For deletions, an additional step is needed: for all affected leaves, we flatten their ancestors if the total subtree sizes are smaller than the leaf wrap threshold. Our update algorithms remain simple since no rebalancing is needed for Orth-trees.

3.3 Theoretical Analysis

Due to page limit, we defer the analysis to the Appendix A, and only list the theorems here.

Theorem 3.1. *Alg. 1 constructs a P-Orth tree of size n using $O(n \log \Delta)$ work, $O(\log n \log \Delta)$ span, and $O(n/B \log_M \Delta)$ cache complexity. A batch update of size $m = O(n)$ on a P-Orth tree of size n uses $O(m \log \Delta)$ work, $O(\log m \log \Delta)$ span, and $O(m/B \log_M \Delta)$ cache complexity.*

Here, Δ denotes the aspect ratio, defined as $\frac{\max d(x,y)}{\min d(x,y)}$ for all points x and y. Note that $\log \Delta \geq \log \Theta(n^{1/D}) = \Omega(\log n)$ when the point set contains no duplicates in $\mathbb{R}^D$.

With stronger assumptions—for instance, a *bounded aspect ratio* ($\Delta \leq n^c$ for some constant $c > 0$) and a *constant expansion rate* (full definition in the Appendix A)—we may obtain tighter bounds. With bounded aspect ratio, we can show that the construction with $O(n \log n)$ work, $O(\log^2 n)$ span, and $O(n/B \log_M n) = O(\mathrm{Sort}(n))$ cache complexity. Updates have $O(m \log n)$ work, $O(\log m \log n)$ span, and $O(m/B \log_M n)$ cache complexity. With both assumptions, a k-NN query can be answered in $O(k \log n)$ work [17].

4 The Spatial PaC-Tree (SPaC-Tree)

This section presents the design of the Spatial PaC-tree (SPaC-tree), a highly parallel R-tree with extremely fast construction and updates while maintaining good query speed.

Existing R-trees. As introduced in Sec. 2, R-trees are object-partitioning trees, leaving flexibility in the heuristics used to build them. The original and early designs [11, 14, 33, 45, 55] are incremental: points are inserted one by one; a greedy strategy iteratively selects a subtree for this point. When a subtree is much heavier than its siblings, a split is applied by a heuristic (e.g., "linear" [5, 33], "quadratic" [33], or "R*" [11, 33]). While simple and highly dynamic, this approach is hard to generalize to parallel batch updates. Consequently, prior work on parallel R-trees has primarily focused on parallel queries [37, 42, 48, 63] or static construction (bulk loading) [1, 6, 29, 41, 47, 52, 56]. However, for purely static scenarios, kd-trees and Orth-trees are often preferable choices.

A promising approach to parallelize R-trees is via space-filling curves (SFCs). SFCs map points in higher dimensions to 1D (see Fig. 1), enabling all points to be organized in this 1D order using a binary search tree (BST) or a B-tree—equivalently yielding an R-tree if each node maintains its bounding box. This idea was first noted by Tropf and Herzog [58], and later realized in the Hilbert R-tree [35, 38], which is built atop a B-tree. Unfortunately, parallel batch update on B-trees can be challenging. Qi et al. [50] showed that the logarithmic method can sidestep parallel updates for B-trees, but it introduces substantial query overhead [43].

The PaC-tree. The PaC-tree [24] is a parallel binary search tree (BST) with the leaf-wrapping technique to enable better space- and I/O-efficiency, where a subtree of size under a threshold ϕ (typically 32) is flattened into a compressed leaf stored as an array. It uses a "Join-based framework" in a divide-and-conquer manner for high parallelism, and supports the full BST interface, including construction, single and batch updates, and various 1D queries.

Our SPaC-Tree. At first glance, PaC-trees appear to provide a straightforward solution for parallelizing R-trees: they can be directly adopted to support an SFC-based approach, achieving both efficiency and high parallelism. We implemented this straightforward design and, somewhat unexpectedly, found it much slower than P-Orth trees and Pkd-trees (see CPAM-H and CPAM-Z in Fig. 3). The bottleneck is that a PaC-tree enforces a total order on points according to an SFC, which is overly costly. In contrast, P-Orth trees and Pkd-trees leave points in the leaves unsorted.

To reduce update costs, we introduce the ***Spatial-PaC-tree (SPaC-tree)***. The primary goal is to keep leaf points unsorted, which requires redesigning and disentangling parts of the underlying PaC-tree algorithms. The remainder of this section presents the new design and its analysis.

4.1 SPaC-Tree Construction

We first show the construction algorithm for SPaC-trees in Alg. 3. To use PaC-tree for construction, a simple idea is to first compute the SFC code for each point, sort the points accordingly, and then build a balanced BST tree on the sorted points. Despite theoretical efficiency, directly calling the PaC-tree in CPAM in this way is up to 3× slower than Pkd-tree construction. To improve performance, our main effort is to avoid unnecessary memory reads/writes by redesigning the sorting algorithm, shown in function "HybridSort" Alg. 3, with two major improvements. First, instead of pre-calculating SFC values before sorting, we compute them when the points are first touched in sorting, which saves one round of reads and writes to associated arrays. Second, we only sort the $\langle code, id \rangle$ pairs (line 13), without the coordinates. This reduces the memory footprint of the recursive sorting process (thus faster speed), at the cost of more cache misses when fetching points to the leaves. Overall this reduces the running time. Combining the two techniques together, Alg. 1 can achieve a consistent speedup over the plain implementation (3.1–3.5× on 2D data; see Fig. 3).

4.2 Batch Updates on SPaC-Trees

Our SPaC-tree builds upon PaC-tree [24], a parallel BST using the join-based algorithmic framework [18]. The high-level idea is to use and only use the Join operation for tree

Algorithm 3: Parallel SPaC-tree construction

Input: A sequence of points P.
Output: A SPaC-tree T on points in P.

```
1  Function BuildSPaCTree(P)
2      A ← Auxiliary sequence of empty pairs ⟨code, id⟩ with size |P|
3      A' ← HybridSort(P, A)
4      return BuildSorted(P, A')
   // Modify the sample-sort to compute the SFC code with sorting
5  Function HybridSort(P, A)
6      Sample points from P and compute their SFC codes
7      Sort samples and sub-sample them to get the pivots
8      Partition P into blocks, and compute offsets of blocks as F[]
9      parallel-for i-th block B do
10         parallel-for j-th point p in B do
11             k ← The SFC code of p
12             id ← The id of p
13             A[F[i] + j] ← ⟨k, id⟩        // Store the code and id in A
14         Sort the slice A[F[i], A[F[i + 1])
15         Merge with samples to get counts for each block
16     Redistribute A to buckets A' using the matrix
         transpose [10, 20], where the i-th bucket has offset F'[i]
17     parallel-for the i-th bucket do                // Recursive sorting
18         Sort the slice A'[F'[i], A'[F'[i + 1])
19     return The sorted sequence A'
   // Recursively construct the tree.
20 Function BuildSorted(P, A)
21     n ← |P|
22     if n ≤ φ then                  // Input size is below the leaf wrapping
23         Retrieve points S ⊆ P using the ids in A
24         return A leaf node with points S and its bounding box
25     else
26         m ← n/2
27         In Parallel:
28             L ← BuildSorted(P[0, m), A[0, m))
29             R ← BuildSorted(P[m + 1, n), A[m + 1, n))
30     k ← the point in P with id in A[m]                // The pivot point
31     return An interior node with left child L, right child R, pivot
         k, and computing the bounding box from children
```

rebalancing, which takes two subtrees L, R, and a key k in the middle, and returns a new, balanced tree with $L \cup \{k\} \cup R$. Our key observation here is that, as a spatial index, the order of the points in a leaf, which in this case is based on Hilbert- or Z-Code, does not facilitate spatial queries—queries on a leaf must scan all points anyway. Therefore, our goal is to carefully redesign the Join-based algorithms, such that we can maintain theoretical efficiency, and adapt them best to the spatial index setting by *relaxing the key order in the leaves*. In our experiments, such an improvement significantly speeds up the updates without sacrificing query performance.

We show the detailed batch insertion algorithm in the Alg. 4. The algorithm begins with computing the SFC code and sorting the inputs. After sieving points to the leaves, the algorithm either appends points to the leaf and marks it as unsorted or rebuilds the leaf if its size exceeds the threshold (line 11 and line 12). Next, the standard Join operation combines two subtrees L and R, and performs the rebalancing (line 19). Without loss of generality, we assume L is heavier than R, and the RightJoin operation is called (line 21). The

Algorithm 4: Parallel Batch Insertion on SPaC-trees

Input: A sequence of points P and a SPaC-tree T.
Output: A SPaC-tree with P inserted.

```
1  Function PTreeBatchInsert(T, P)
2      Compute SFC codes for points in P, and sort P accordingly.
         // In practice we use the HybridSort() from Alg. 3
3      return InsertSorted(T, P)
4  Function InsertSorted(T, P)
5      n ← |P|
6      if n = 0 then return T
7      if T is a leaf then
8          if |T| + n ≤ φ then
9              Append P to T, and mark T as unsorted
10             Update the bounding box of T
11             return T
12         else return BuildSPaCTree(P ∪ T)
13     k ← the SFC code associated with the pivot in (root of) T
14     t ← binary search k in P (based on the code)
15     In Parallel:
16         L ← InsertSorted(T_ℓ, P[0, t))
17         R ← InsertSorted(T_r, P[t, n))
18     Update the bounding box of T based on those of L and R
19     return Join(L, T_p, R)
   // Return a balanced tree joining L and R with pivot k.
20 Function Join(L, k, R)  // this function remains the same as in [18, 24]
21     if L is heavier then return RightJoin(L, k, R)
22     if R is heavier then return LeftJoin(L, k, R)
23     return Node(L, k, R)
   // Recursively check L's right spine until the sub-tree size balances with
      R. Create a new tree node R' with children the two balanced sub-trees,
      attach R' to L, and re-balance L.
24 Function RightJoin(L, k, R)                    // LeftJoin is symmetric
25     ⟨L_ℓ, k', L_r⟩ ← Expose(L)        // Expand L into a tree if it is a leaf
26     if L and R is balanced then                // The split terminates here
27         return Node(L, k, R)                   // Return a balanced tree
28     R' ← RightJoin(L_r, k, R)// Recursively split the right sub-tree of L
29     L' ← Node(L_ℓ, k', R')    // Attach the newly balanced tree R' to L
30     Re-balance L' by rotation
31     return L'
   // Expand T into a tree if it is a leaf, and reorder the points if necessary.
32 Function Expose(T)
33     if T is a leaf then
34         Re-order the points if T is marked as unsorted
35         Build a perfect balanced tree T' from the sorted points in T
36         return {T'_ℓ, T'_p, T'_r}
37     else return {T_ℓ, T_p, T_r}                // Return the tree as is
38 Function Node(T_ℓ, k, T_r)      // Maintain the leaf wrapping invariant.
39     Create a node T with pivot k, left sub-tree T_ℓ and right sub-tree T_r.
40     n ← |T|
41     if n > 2φ then return T              // Leaf wrapping does not apply
42     else if n > φ then       // Redistribute points in leaves T_ℓ and T_r
43         Sort points in T_ℓ and T_r if they are marked as un-sorted.
44         Redistribute sub-trees of T into two leaf nodes with size n/2.
45         return T
46     else   // Tree size is below the leaf wrapping, embed it into one leaf
47         Flatten T and create a leaf node wrapping it.
48         return this new leaf node
```

RightJoin recursively splits the right subtree of L until it is possible to return a balanced tree using R (line 26). When the split reaches a leaf, we expand the leaf into a tree as in PaC-trees using the Expose operation (line 32). The difference is if the leaf is marked as unsorted, we will sort the points first (line 43). When the split subtree is balanced with R (line 26), we create a new tree node R' with children the two balanced subtrees (line 28), and attach R' to L (line 29).

Note that the previous leaf expansion may break the leaf wrapping for affected leaves. In this case, we restore the leaf wrapping by checking the tree size: either directly flatten it into one leaf if the size fits within (line 46), or redistribute the points into two leaves if necessary (line 42). We will sort the points first if leaves are marked as unsorted (line: 43).

Despite Alg. 4 appearing complicated, we can prove its correctness by showing its equivalence to a PaC-tree. For page limit, we defer the analysis to Appendix B.

The batch deletion algorithm is similar to the insertion. The only difference is that when it reaches a leaf, it removes the points there, marks the leaf as unsorted if necessary, and updates the bounding box. The invariant of leaf wrapping is maintained the same way as in insertion, i.e., line 23 and 29.

4.3 Theoretical Analysis

Due to the page limit, we defer the full analysis to Appendix B, and present only the results here.

Theorem 4.1. *For n points with integer coordinates, a SPaC-tree with Hilbert- or Z-curve can be constructed in $O(n \log n)$ work, $O(\log n)$ span, and $O(Sort(n))$ cache complexity. A batch update (insertion or deletion) of size m on a SPaC-tree of size n uses $O(m \log n)$ work and $O(\log^2 n)$ span.*

5 Experiments

We conduct in-depth experiments to understand the performance of Ψ-Lib and other spatial indexes on both synthetic and real-world datasets. We show that both P-Orth trees and SPaC-trees achieve superior construction and update performance, outperforming Pkd-tree in most cases, and are much faster than existing Orth-tree and R-tree baselines. Both P-Orth trees and SPaC-trees also exhibit comparable or better query performance to their corresponding counterparts in prior work. In addition to showing the effectiveness of our new algorithms, we believe our experiments also provide the first systematic study of various parallel spatial indexes, including kd-trees, Orth-trees, and R-trees.

Setup. We use a machine with 112 cores (224 hyperthreads) with four Intel Xeon Platinum 8176 CPUs and 1.47 TB RAM. Ψ-Lib is in C++ and compiled using GCC 14.2.1 with `-O3`. We use the ParLaylib [16] for fork-join parallelism. Our anonymous code is available at [9]. We report numbers as the average of 3 runs after a warm-up run. More details about parameter choosing are shown in Appendix C.

Baselines. We compare to the following baselines.

- **Pkd-trees** [43]: The state-of-the-art parallel kd-tree.
- **Zd-trees** [17]: The state-of-the-art parallel Orth-tree. Zd-tree uses Morton code to presort the data to aid the construction and update algorithm in a standard Orth-tree. The original code from [17] has known bugs in the update algorithms (confirmed by the authors). We use our own implementation based on their paper. We have carefully verified that our construction time is similar to their code.
- **CPAM** [24]: As a baseline, we use PaC-trees from the CPAM library (as a black box) to store each point's SFC code as the key. It preserves a total order of all points based on the Morton curve (CPAM-Z) or the Hilbert curve (CPAM-H). This baseline highlights how our new design by maintaining only a partial order improves performance.
- **Boost R-trees** [51]: The R-tree from the Boost library. Boost R-tree is sequential, and only supports point updates (no batch updates). We mainly use it as a baseline to verify the query performance for our SPaC-trees. Hence, among all the variants, we use the `quadratic` version, which gives the best tree quality in the dynamic setting.

Within Ψ-Lib, we tested the parallel Orth-tree—the P-Orth tree—as introduced in Sec. 3, and two R-trees—SPaC-H-tree and SPaC-Z-tree—which use Hilbert and Morton curve, respectively, on the SPaC-tree detailed in Sec. 4. We maintain bounding boxes for all tested indexes. We refer readers to the Appendix C for more implementation details.

5.1 Overall Evaluation under Synthetic Datasets

Setup. We test different distributions for points, queries, and update patterns of synthetic data. All coordinates are 64-bit integers in $[0, 10^9]$. We use three workloads: `Uniform`, `Sweepline` and `Varden`. `Uniform` draws each point uniformly random from the space. `Sweepline` also uses uniform data, but sorts all points along the first dimension. This is used to simulate a skewed update pattern, where the updated points exhibit spatial locality. `Varden` [28] is generated by randomly walking in the space with a low probability to restart at a random position. Points are clustered and different clusters are far from each other, simulating a skewed point distribution.

We test both static and dynamic cases. Besides directly measuring the tree construction time, we also use the *incremental insertion/deletion* workload with various batch sizes to simulate a highly dynamic scenario. For batch size b, an incremental insertion workload means to construct the index by n/b batch insertions progressively, and vice versa for deletions (deleting the index in n/b batches). We report the total running time of all operations. This reflects how the *update efficiency* of each index is affected under a constantly evolving dataset. Under this workload, we further time the queries after half of the batches. The query performance reflects how the *quality* of each index is affected after massive updates. For the static setting, we also provide query times after building a tree with half of the data for easy comparison with the dynamic setting. We also test the update time for a single batch, and show the results in the Appendix D.

We tested k-NN and range queries (introduced in Sec. 2). We run 10^7 10-NN queries for both in-distribution (InD) and out-of-distribution (OOD) queries. For range queries, we test 5×10^4 range-count and range-list queries, with range sizes 10^4–10^6. Different queries run in parallel. Besides Fig. 3, we further study how k-NN and range-list performance changes with their output sizes and show results in Fig. 4 and 5.

We summarize in Fig. 3 the performance of all tested in-

		Build time	Query after Build (50%)				Incremental Insert				Query after Inc. Ins. (50%)				Incremental Delete				Query after Inc. Del. (50%)			
			10NN		Range		Batch Size				10NN		Range		Batch Size				10NN		Range	
			InD	OOD	Count	List	10%	1%	0.1%	0.01%	InD	OOD	Count	List	10%	1%	0.1%	0.01%	InD	OOD	Count	List
Uniform	★P-Orth	3.23	**.362**	**.363**	**.078**	1.15	4.27	10.3	19.7	29.7	**.381**	**.382**	**.080**	**1.04**	**4.32**	**10.2**	**19.6**	**29.9**	**.394**	**.397**	**.089**	1.18
	Zd-Tree	4.83	.490	.485	.142	1.59	7.80	13.2	24.9	48	.482	.486	.145	1.53	9.30	18.3	32.5	52	.510	.508	.137	1.50
	★SPaC-H	3.34	1.93	1.93	.157	1.17	3.90	6.00	13.0	26.8	1.96	1.96	.163	1.25	4.69	14.4	27.7	42.2	1.95	1.96	.161	1.24
	★SPaC-Z	**3.10**	9.13	9.20	.229	1.22	**3.68**	**5.75**	**12.6**	**26.2**	9.21	9.29	.237	1.31	4.78	14.2	27.4	42.0	9.12	9.20	.237	1.30
	CPAM-H	11.3	2.52	2.50	.216	1.39	15.3	38.2	108	159	2.44	2.44	.225	1.46	17.2	39.0	107	157	2.43	2.43	.225	1.46
	CPAM-Z	10.8	11.9	11.9	.311	1.46	14.7	37.6	108	158	11.6	11.7	.326	1.54	16.7	38.4	106	157	11.5	11.5	.325	1.52
	Boost-R†	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	11.1	11.1	1.49	7.81	N/A	N/A	N/A	N/A	.775	.783	.653	4.60
	Pkd-Tree	3.66	.398	.397	.097	**1.08**	4.52	10.64	20.9	48.9	.433	.416	.103	1.05	4.37	11.2	23.0	59.8	.411	.412	.104	**1.07**
Sweepline	★P-Orth	4.63	**.220**	**.373**	.098	1.22	5.29	5.67	5.72	9.40	**.227**	**.333**	**.074**	**1.01**	**1.92**	3.08	4.25	8.74	**.222**	**.322**	**.073**	**1.02**
	Zd-Tree	5.37	.284	.388	.140	1.64	6.00	4.08	6.1	12.5	.285	.408	.127	1.47	6.00	4.73	11.9	29.0	.293	.415	.121	1.45
	★SPaC-H	3.32	.855	.661	.156	1.19	2.99	3.09	4.12	8.85	1.05	.797	.162	1.29	2.32	2.50	3.23	**8.37**	.941	.766	.159	1.26
	★SPaC-Z	**3.04**	1.94	.874	.214	1.23	**2.77**	**2.83**	**3.83**	**8.74**	1.74	1.16	.217	1.32	2.16	**2.27**	**3.01**	9.09	2.00	1.05	.215	1.30
	CPAM-H	10.3	1.16	.765	.216	1.43	13.5	10.2	8.29	19.4	1.19	.812	.227	1.48	15.0	11.1	8.67	21.2	1.19	.853	.222	1.48
	CPAM-Z	9.81	2.62	1.04	.292	1.47	13.1	9.96	8.04	20.4	2.50	1.21	.309	1.53	14.4	10.7	8.52	22.7	2.71	1.18	.300	1.54
	Boost-R†	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	.921	.931	3.18	6.10	N/A	N/A	N/A	N/A	.653	.651	.455	4.14
	Pkd-Tree	5.16	.243	.435	**.093**	**1.11**	13.7	24.0	35.8	39.9	.331	1.61	.108	1.09	10.0	24.3	36.8	62.8	.263	.337	.098	1.07
Varden	★P-Orth	12.2	.155	.279	**.054**	1.12	13.7	11.5	12.6	26.2	.160	.247	**.050**	**1.01**	6.79	8.48	12.5	28.3	.160	.239	**.050**	**1.01**
	Zd-Tree	5.6	.192	**.156**	.086	1.57	5.95	4.24	6.1	14.6	.196	**.158**	.073	1.40	6.08	5.25	15.9	36.1	.198	**.157**	.072	1.39
	★SPaC-H	3.09	2.24	.565	.103	1.15	2.71	2.95	4.18	9.23	2.26	.494	.107	1.23	2.22	2.54	3.54	8.21	2.25	.578	.106	1.22
	★SPaC-Z	**2.94**	3.92	1.02	.165	1.19	**2.52**	**2.68**	**3.77**	**8.74**	3.61	.855	.171	1.27	**2.03**	**2.27**	**3.26**	**7.80**	4.30	2.29	.168	1.26
	CPAM-H	10.0	2.52	.663	.146	1.38	13.3	10.2	8.59	19.0	2.48	.592	.152	1.45	14.9	11.2	9.12	19.7	2.55	.680	.152	1.43
	CPAM-Z	9.66	4.62	1.22	.229	1.42	13.0	9.94	8.24	18.5	3.32	1.08	.239	1.49	14.0	10.8	8.75	20.2	5.08	2.86	.237	1.49
	Boost-R†	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	.922	.924	.521	4.45	N/A	N/A	N/A	N/A	.617	.624	.429	4.20
	Pkd-Tree	6.10	**.110**	.632	.060	**1.07**	12.8	25.2	32.9	53.6	**.109**	.725	.064	1.03	9.36	18.9	28.8	51.6	**.112**	.804	.063	1.04

The **fastest time** is in bold and underlined : within 1.1x the fastest : within 2x the fastest : within 5x the fastest : > 5x the fastest

Figure 3. Running time (in seconds) on synthetic data. Lower is better. The fastest time in each test is in bold and underlined. We use colors to mark results within 1.1×, 2×, 5×, and > 5× the fastest time. Detailed settings for build, queries, and incremental insertion/deletion are introduced at the beginning of Sec. 5.1. InD/OOD: in-/out-of-distribution. †: Boost R-tree is sequential and only support point updates. Therefore, we omit the construction/update times, and report query times after incremental inserting/deleting points one by one.

dexes on synthetic datasets with 10^9 2D points. We provide the results on 3D points in Appendix F. Next we analyze the performance in detail.

5.1.1 Construction. For tree construction, our SPaC-tree is the fastest among all indexes across all workloads. The advantage comes from embedding 2D data into 1D that simplifies the computation, and various optimizations in Ψ-Lib introduced in Sec. 4. SPaC-Z-tree is slightly faster than SPaC-H-tree, since Morton code has simpler computation than Hilbert code. The baselines CPAM-H and CPAM-Z are about 3× slower than our SPaC-trees, due to the overhead in maintaining the ordering in leaves. This effect is even more significant in batch updates and queries. This justifies the necessity of our technique of relaxing the ordering in leaves.

For Orth-trees, on `Uniform` and `Sweepline`, the P-Orth tree also achieves good performance (within 52% slower than the fastest SPaC-Z-tree), and is faster than all other baselines. The advantage of P-Orth trees over Pkd-trees come from two aspects: 1) as a Quad-tree, P-Orth tree allows for shallower tree height and better locality than the binary kd-tree, and 2) determining the splitter at each node in a P-Orth tree (computing the middle of the coordinate range) is simpler than kd-tree (estimating the median among all points).

On `Varden`, P-Orth tree becomes slower than others. Since Orth-trees split the space using the coordinate median, it is naturally not resistant to skewed data, and is most affected by the skewed distribution. Although Zd-tree is also a Orth-tree, it achieves reasonable performance— the main cost for Zd-tree construction is to sort all points in Morton order, and this is done by a comparison sort in our implementation. All other indexes are comparison-based, and the effect of skewed data on them is minimal in construction time.

In summary, SPaC-trees have consistently better performance than all other baselines in construction. P-Orth tree is also competitive on non-skewed data, but exhibits a disadvantage on skewed data.

5.1.2 Incremental Batch Updates. The conclusions for batch updates are very similar to those of construction. SPaC-trees has the best overall performance, and SPaC-Z-tree has a slight advantage over SPaC-H-tree. SPaC-Z-tree is the fastest in all incremental insertions, and most cases in incremental deletions. For the same reason analyzed in Sec. 5.1.1, P-Orth trees are less ideal for `Varden` data. In all other cases, P-Orth trees are either the best or close to the best.

For all indexes, the incremental update time increases when the batch size decreases. On the one hand, smaller batches result in less potential for parallelism.On the other hand, having more batches also means more modifications to the tree, requiring more effort to rebalance the tree and leaving the tree further from being perfectly balanced. The only index that avoids rebalancing is the Orth-tree, and its performance with continuous updates is the least affected by the batch size.

On highly dynamic data, Pkd-trees are less competitive in update time compared to P-Orth trees and SPaC-trees. One essential reason is that Pkd-tree has $O(\log^2 n)$ amortized cost per updated point, while P-Orth tree and SPaC-tree have cost of $O(\log \Delta)$ and $O(\log n)$, respectively, where Δ is

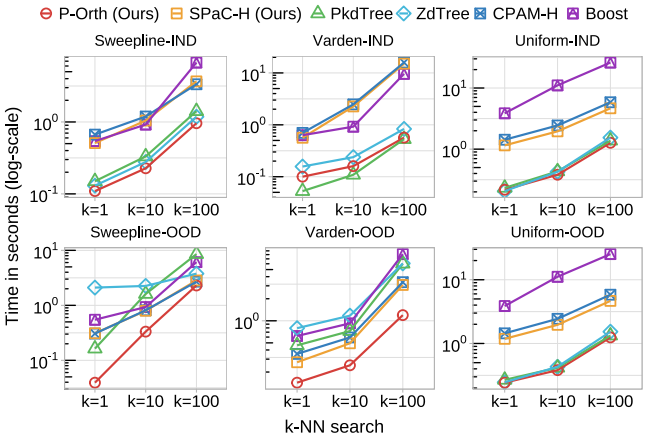

Figure 4. Running time (in seconds) of k-NN queries for $k \in \{1, 10, 100\}$. Lower is better. The dataset contains 500M points in 2 dimensions. The tree is constructed by incremental insertion with batch ratio 0.01%. The test contains k-NN queries from 10^7 points from both InD and OOD distribution. Plots are in log-log scale.

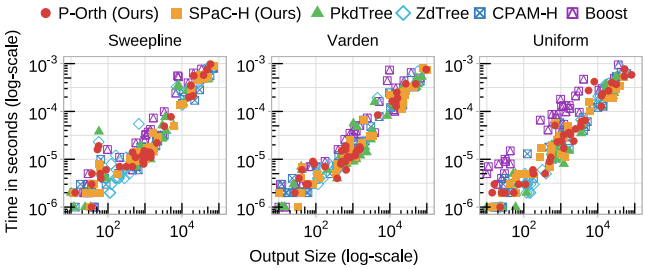

Figure 5. Running time (in seconds) of range report queries for w.r.t output sizes. Lower is better. The dataset contains 500M points in 2 dimensions. The tree is constructed by incremental insertion with batch ratio 0.01%. Plots are in log-log scale.

the aspect ratio. Hence, both P-Orth trees and SPaC-trees are faster than Pkd-trees in updates. In particular, P-Orth trees are up to 7.18× faster than Pkd-tree in incremental updates, and SPaC-tree can be up to 7.5× faster. Even for Varden where Δ is relatively large, P-Orth trees are almost always faster than Pkd-trees in incremental updates.

5.1.3 Queries. We run queries in three settings: 1) after constructing a tree of size 5×10^8, 2) after applying 50% of the insertion batches, and 3) after applying 50% of the deletion batches. Most indexes are nearly perfectly balanced after construction, and thus the first setting reflects their best-case (static) query performance. The other two settings reflect how the index quality is affected by updates. In Fig. 3, we only select results for 10-NN query and a relatively large range query. To give more details, in Fig. 4 and Fig. 5, we further show how query performance changes with the output size, i.e., k in k-NN, and the range size in range-list queries.

***k*-NN Queries.** As shown in Fig. 4, space-partitioning trees are evidently faster than R-trees in k-NN queries. This is natural due to overlapping bounding boxes in R-trees. For SPaC-trees, while the SPaC-H-tree is slightly slower than SPaC-Z-tree in construction and updates, it is much more efficient in queries. This is because the Hilbert curve has better locality than the Morton curve (adjacent codes are always geometrically close to each other). Among the R-trees, SPaC-trees achieve similar or better performance than Boost R-tree—in all queries, SPaC-H-tree is between 3.7× slower to 5.66× faster, with a geometric mean of 2.5× faster.

Among the space-partitioning trees, Orth-trees has the best overall performance. This is because when visiting a subtree, the P-Orth tree can select 1 out of 4 quadrants, which is more effective than Pkd-trees and Zd-trees that select 1 out of 2 half spaces. Hence, Pkd-trees and Zd-trees are competitive but usually slower than P-Orth trees. The only exception is on Varden data. For InD queries, due to the skewed distribution of Varden, Orth-trees may be unbalanced, and thus the comparison-based Pkd-trees perform better. Interestingly, on the contrary, both Orth-trees exhibit an advantage on OOD queries on Varden. The reason is still in imbalance—for Varden, points are highly clustered, making these regions in the tree deep and other regions shallow. Since the OOD queries distribute differently from the input, they likely hit the shallow regions and thus are much faster.

Range Queries. As shown in Fig. 3 and 5, Pkd-trees show a small but consistent advantage on range queries. This is because a range query visits all subtrees overlapping the query box. In this case, P-Orth trees have to explicitly check the bounding boxes for four subtrees, while every non-overlapping check on a Pkd-tree node can prune half of the points in this subtree. For other indexes, the relative performance on range queries is similar to k-NN queries. Interestingly, while SPaC-trees are still slower than kd-trees and P-Orth trees in range-list queries, the difference is much smaller, especially on large ranges— in this case, the query time is mostly spent emitting the result list, hiding the difference in pruning effectiveness across indexes. Therefore, range queries are less sensitive to the index type than k-NN queries.

Impact of Updates to Queries. In the dynamic setting, the Orth-trees (P-Orth tree and Zd-tree) are history-independent (modulo leaf-wrapping), namely, the final state of the tree is not affected by the operation order. Therefore, their query performance is least affected by batch updates, and is the best in the dynamic setting.

For all other indexes, the tree may get less balanced after updates. Indeed, they all get slower to some extend compared to the static setting. This impact is moderate for most indexes (mostly within 20%). The exceptions all appear in OOD k-NN queries, where Pkd-tree gets 3.7× slower after incremental insertion on Sweepline, and CPAM-Z and SPaC-Z-tree get about 2.5× slower after incremental deletion on Varden.

In summary, for queries, Orth-trees and kd-trees are naturally better than R-trees. kd-trees are better in dealing with InD queries on non-uniform data, but may be worse in OOD queries. P-Orth tree has the best or close to the best query performance in almost all queries and workloads.

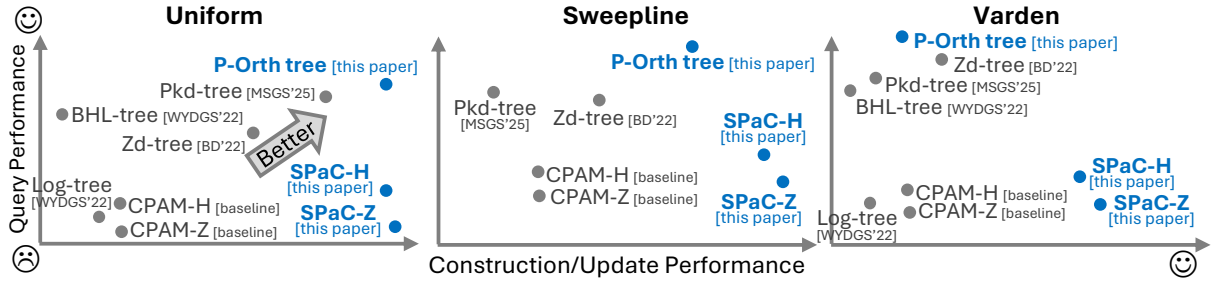

Figure 6. Summary of tested index in update and query performance. Results are summarized from numbers in Fig. 3. In particular, the data points are based on the geometric mean of all relevant operations (updates or queries) in Fig. 3. Data points for Log-tree and BHL-tree [62] are estimated from the Pkd-tree paper [43]. Our new algorithms are marked in blue. We note that this figure only gives the average of the tested benchmarks in this paper. More comprehensive conclusions can be found in Table 1.

	Build	Update Ins.	Update Del.	Query 10NN	Query RG	Build	Update Ins.	Update Del.	Query 10NN	Query RG
	Cosmo (3D), n=317M					OSM (2D), n=776M				
★P-Orth	1.90	16.2	17.9	.120	**.566**	4.96	14.5	14.9	.083	.050
Zd-Tree	1.65	13.8	20.7	.146	.862	5.88	16.5	23.9	.182	.055
★SPaC-H	1.02	6.59	9.40	.393	.764	2.26	8.19	7.98	.981	.085
★SPaC-Z	**.837**	**5.63**	**8.46**	2.58	.980	**2.12**	**7.91**	**7.37**	2.91	.132
CPAM-H	5.48	19.6	19.8	.509	1.04	7.26	15.6	16.7	1.10	.118
CPAM-Z	5.38	19.0	19.2	4.39	1.30	7.01	15.4	16.6	3.47	.182
Boost-R	N/A	N/A	N/A	.274	.977	N/A	N/A	N/A	.484	.435
Pkd-Tree	1.89	101	800	**.107**	.602	4.32	29.3	26.3	**.071**	**.049**

fastest time : <1.1x fastest : <2x fastest <5x fastest : > 5x fastest

Figure 7. Running time (in seconds) on real-world datasets. Lower is better. Insert/Delete: incremental insertion/deletion/ with batch size 0.01%. "RG": Range-list queries.

5.2 Operations on Real-World Datasets

For real-world datasets, we test a highly clustered dataset COSMO [53] and the OpenStreetMap (OSM [34]) for Northern America. We test 10^7 InD 10-NN quiries, and 5×10^4 range-list queries with range size 10^4–10^6. Coordinates are rounded down to 64-bit integers. We remove duplicates and shift all points to positive coordinates. To ensure the SFC works properly in 3D, we scale the coordinates to $[0, 10^6]$. We evaluate construction, incremental updates with batch ratio 0.01% and queries after construction, and show results in Fig. 7.

SPaC-trees are much faster than others in construction and updates. On real-world data, this advantage is more significant than synthetic data. In particular, they are about 2× faster than Pkd-trees in construction, and 3.5–94× faster in updates. P-Orth trees have similar construction times to Pkd-trees, but are much faster in updates (1.8–44.7× faster).

On queries, R-trees still perform worse than space-partitioning trees. In most of the cases, Pkd-tree achieves the best query performance, but P-Orth trees are always competitive—in all cases, the difference in within 20%. Considering that P-Orth trees are 1.8–44.7× faster in updates, P-Orth trees offer a much better query/update tradeoff than Pkd-trees.

5.3 Scalability

We evaluate the scalability of tested indexes in construction, insertion, and deletion. Since most of them achieve high parallelism, we only list the conclusions here, and refer readers to Appendix E for details).

In general, all indexes scale well to 224 hyperthreads, which means the performance difference mainly comes from the work (i.e., one-core performance). Among them, SPaC-H-tree has the best self-relative speedup, which is up to 82.9× in build and 80× in insertion. This is likely due to its simple structure as a 1D search tree. Combining both low work and good scalability, SPaC-H-tree has the best overall construction and update performance. The P-Orth tree have good scalability on Uniform, but is slightly worse on Sweepline and Varden due to the imbalanced tree.

5.4 Summary

Combining all the experimental results, we visualize the tradeoff between update and query performance for all tested indexes in Fig. 6 and provide a brief summary here. We also summarize the conclusions in a table, which we put in the appendix due to page limit. We recommend readers to read the table as well to see more detailed analysis.

Pkd-trees. Pkd-trees offer solid performance in queries, but can degrade on OOD queries. Its update performance is reasonable but less competitive than P-Orth trees and SPaC-trees. Our results suggest they are best suited to scenarios with light to moderate update rates, high query throughput requirements, and predominantly in-distribution queries.

P-Orth trees (this paper). P-Orth trees generally give the best overall performance and trade-off between query and updates, especially non-skewed data. It is best suited to scenarios with less skewed data, with any update-query ratio. It is also friendly to queries after high-volumes of updates, since the tree quality does not degrade with frequent updates.

SPaC-trees (this paper). SPaC-H-tree performs slightly worse in updates than SPaC-Z-tree, but significantly better in queries. We would recommend SPaC-H-tree as the default setting for SPaC-trees. Compared to Pkd-trees and P-Orth trees, SPaC-H-trees are less effective in queries, but significantly faster in construction and updates. In general, SPaC-H-trees are best suited to highly dynamic scenarios where either updates requires very high throughput/low latency, or updates are much more frequent than queries.

6 Conclusion

In this paper, we systematically study parallel spatial indexes, with a special focus on achieving high-performance updates in highly dynamic workloads. We proposed two new data structures: a parallel Orth-tree, the P-Orth tree,

and a parallel R-tree, the SPaC-tree family. Both achieve superior update performance compared to existing parallel spatial indexes, while remaining competitive with or better than their counterparts in the literature for queries. We also highlight our comprehensive experiments to understand the performance of existing and our new parallel spatial indexes, and share our findings in Sec. 5.4 and Fig. 6.

References

[1] Daniar Achakeev, Marc Seidemann, Markus Schmidt, and Bernhard Seeger. 2012. Sort-based parallel loading of R-trees. In *BigSpatial@ACM Special Interest Group on Spatial Information (SIGSPATIAL)*. 62–70.

[2] Stephen Adams. 1992. *Implementing Sets Effciently in a Functional Language*. Technical Report CSTR 92-10. University of Southampton.

[3] Stephen Adams. 1993. Efficient sets—a balancing act. *J. Functional Programming* 3, 04 (1993), 553–561.

[4] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2019. *Real-time rendering*. Crc Press.

[5] Chuan-Heng Ang and Tuck-Choy Tan. 1997. New linear node splitting algorithm for R-trees. In *International Symposium on Spatial Databases (SSD)*. Springer, 337–349.

[6] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)* 4, 1 (2008), 1–30.

[7] Lars Arge, Klaus H Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. 2002. Efficient bulk operations on dynamic R-trees. *Algorithmica* 33, 1 (2002), 104–128.

[8] 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.

[9] Anonymous authors. 2025. Code for PSI-Lib. https://anonymous.4open.science/r/SpaceTreeLib-422B/.

[10] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-place parallel super scalar samplesort (ipsssso). In *European Symposium on Algorithms (ESA)*.

[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 322–331.

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

[13] Jon Louis Bentley. 1978. *Decomposable searching problems*. Technical Report. Carnegie Mellon University.

[14] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. In *Proceedings of the VLDB Endowment (PVLDB)*. Morgan Kaufmann, 28–39.

[15] Jiři Bittner, Michal Hapala, and Vlastimil Havran. 2015. Incremental BVH construction for ray tracing. *Computers & Graphics* 47 (2015), 135–144.

[16] 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.

[17] 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.

[18] 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)*.

[19] 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.

[20] 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)*.

[21] Guy E. Blelloch and Yan Gu. 2020. Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*.

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

[23] J. K. Chen, Yin-Fu Huang, and Yeh-Hao Chin. 1997. A Study of Concurrent Operations on R-Trees. *Inf. Sci.* 98, 1-4 (1997), 263–300.

[24] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[25] Andreas Fabri and Sylvain Pion. 2009. CGAL: The computational geometry algorithms library. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 538–539.

[26] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4 (1974), 1–9.

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

[28] 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.

[29] Yván J García R, Mario A López, and Scott T Leutenegger. 1998. A greedy algorithm for bulk loading R-trees. In *ACM International Symposium on Advances in Geographic Information System (SIGSPATIAL GIS)*. 163–164.

[30] Yan Gu, Yong He, and Guy E Blelloch. 2015. Ray specialized contraction on bounding volume hierarchies. In *Computer Graphics Forum*, Vol. 34. 309–318.

[31] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH construction via approximate agglomerative clustering. In *High-Performance Graphics (HPG)*.

[32] 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.

[33] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 47–57.

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

[35] Herman Haverkort and Freek V Walderveen. 2008. Four-dimensional Hilbert curves for R-trees. *Journal of Experimental Algorithmics (JEA)* 16 (2008), 3–1.

[36] Chris L Jackins and Steven L Tanimoto. 1980. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing (CGIP)* 14, 3 (1980), 249–270.

[37] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 21, 2 (1992), 195–204.

[38] Ibrahim Kamel and Christos Faloutsos. 1993. On packing R-trees. In *International conference on Information and Knowledge Management (CIKM)*. 490–499.

[39] Jinha Kim, Seung-Keol Kim, and Hwanjo Yu. 2013. Scalable and parallelizable processing of influence maximization for large-scale social networks?. In *2013 IEEE 29th international conference on data engineering (ICDE)*. IEEE, 266–277.

[40] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 375–384.

[41] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 497–506.

[42] Lijuan Luo, Martin DF Wong, and Lance Leong. 2012. Parallel implementation of R-trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*. IEEE, 353–358.

[43] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Parallel kd-tree with Batch Updates. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 3, 1 (2025), 1–26.

[44] Vincent Ng and Tiko Kameda. 1994. The R-Link Tree: A Recoverable Index Structure for Spatial Data. In *Database and Expert Systems Applications (DEXA)*, Vol. 856. Springer, 163–172.

[45] Yutaka Ohsawa and Masao Sakauchi. 1990. A new tree type data structure with homogeneous nodes suitable for a very large spatial database. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 296–297.

[46] Jacopo Pantaleoni and David Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *High Performance Graphics (HPG)*. 87–95.

[47] Apostolos Papadopoulos and Yannis Manolopoulos. 2003. Parallel bulk-loading of spatial data. *Parallel Comput.* 29, 10 (2003), 1419–1444.

[48] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshop*. IEEE, 618–627.

[49] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability. *Proceedings of the VLDB Endowment (PVLDB)* 11, 5 (2018), 621–634.

[50] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2020. Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability. *ACM Transactions on Database Systems (TODS)* 45, 3 (2020), 1–47.

[51] Boris Schäling. 2011. *The boost C++ libraries*. Boris Schäling.

[52] Bernd Schnitzer and Scott T. Leutenegger. 1999. Master-Client R-Trees: A New Parallel R-Tree Architecture. In *SSDBM*. IEEE Computer Society, 68–77.

[53] 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.

[54] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the VLDB Endowment (PVLDB)*. Morgan Kaufmann, 507–518.

[55] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the VLDB Endowment (PVLDB)*. Morgan Kaufmann, 507–518.

[56] Lai Shuhua, Zhu Fenghua, and Sun Yongqiang. 2000. A Design of Parallel R-tree on Cluster of Workstations. In *International Workshop on Databases in Networked Information Systems*. Springer, 119–133.

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

[58] Herbert Tropf and Helmut Herzog. 1981. Multidimensional Range Search in Dynamically Balanced Trees. *Angewandte Info.* 2 (1981), 71–77.

[59] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Aleksi Tervo, and Jarmo Takala. 2018. PLOCTree: A fast, high-quality hardware BVH builder. *ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018), 1–19.

[60] Ingo Wald, Thiago Ize, and Steven G Parker. 2008. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics* 32, 1 (2008), 3–13.

[61] Michael S Warren and John K Salmon. 1993. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. 12–21.

[62] Rahul Yesantharao, Yiqiu Wang, Laxman Dhulipala, and Julian Shun. 2021. Parallel Batch-Dynamic k d-Trees. *arXiv preprint arXiv:2112.06188* (2021).

[63] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on gpus using r-trees. In *ACM SIGSPATIAL international workshop on analytics for big geospatial data*. 23–31.

	General Features	Existing Solutions and Their Features
***k*d-tree**	+ Linear space + Flexible for most queries (e.g., k-NN, range) + Non-overlapping bounding boxes (thus effective pruning in queries) + Generally fast queries across distributions + Comparison-based, resistant to skewed data + Easily generalizable beyond three dimensions – Slow/complicated updates	**Pkd-tree [43]** + I/O optimizations for construction and updates + Fast construction: $O(n \log n)$ work and polylogarithmic span + Among the fastest for queries in most tests, except for OOD queries on skewed distributions – $O(m \log^2 n)$ work for batch update of batch size m, unfriendly to workloads with frequent updates **BHL-tree and Log-tree [62]** + Can leverage vEB layouts for query optimization (due to their static nature) + Construction with $O(n \log n)$ work and polylogarithmic span – Large batch-update cost: $O(m \log^2 n)$ (Log-tree) or $O((n+m)\log(n+m))$ (BHL-tree, due to fully rebuild) – Log-tree uses logarithmic method, leading to inefficient queries
Orth-tree	+ Linear space + Flexible for most queries (e.g., k-NN, range) + Non-overlapping bounding boxes (thus effective pruning in queries) + Fast queries, especially on non-skewed data + History-independent (modulo leaf wraps) + Simple/fast construction and updates, especially on non-skewed data – Sensitive to skewed data – Usually not generalizable beyond three dimensions	**P-Orth tree (this paper)** + I/O optimizations for construction and updates ★ Fastest query performance on non-skewed data ★ Usually faster updates than Pkd-trees, even on reasonably skewed data; slower than SPaC-trees + Fast construction: $O(n \log \Delta)$ work and polylogarithmic span ★ Fast batch updates: $O(m \log \Delta)$ work and polylogarithmic span – Most affected by skewed data in construction, updates and queries; less efficient for InD queries on skewed data **Zd-tree [17]** + Relatively skew-resistant due to comparison sorting + Fast construction: $O(n \log n)$ work and polylogarithmic span + $O(m \log \Delta)$ work for batch update, where Δ is the aspect ratio – Generally slower updates/construction than the P-Orth tree – Integer coordinates and Morton curve only
R-tree/BVH	+ Linear Space + Flexible rules due to object-partitioning + Simple/fast construction and updates + Applicable to common queries (e.g., k-NN, range) + Easily generalizable beyond three dimensions – Overlapping bounding boxes (thus ineffective pruning in queries); usually slower queries than space-partitioning trees	**SPaC-tree (this paper)** ★ Compatible with Hilbert, Morton or other space-filling curves ★ Embeds multi-dimensional data to 1D, enabling simple algorithm design and high parallelism (best self-speedup among tested indexes) + Fast construction: $O(n \log n)$ work and polylogarithmic span ★ Super fast batch updates: $O(m \log n)$ work and polylogarithmic span + Comparison-based; robust to skewed data ★ Fastest construction and update time among all baselines; significant advantage on updates – Integer coordinates only – Slow queries than space-partitioning trees due to overlapping bounding boxes **Boost R-tree [51]** + Supports multiple heuristics – No parallel construction or batch updates – Slow queries than space-partitioning trees due to overlapping bounding boxes
Range tree	+ Worst-case work bound for range queries – $O(n \log n)$ space – Only supports range queries – Inefficient in more than two dimensions	**CPAM/PAM range tree [24, 57]** + Parallel construction with $O(n \log n)$ work and polylogarithmic span – No simple support for parallel batch updates

Table 1. Summary of the main features of different spatial trees and existing solutions for parallel construction, updates, and queries. The symbol "★"marks our key technical contributions. In the bounds, m is the batch size, n is the index size, and Δ is the aspect ratio.

A Analysis on P-Orth Trees

We now analyze the theoretical guarantees for our P-Orth tree construction and batch update algorithms. Let $S \subseteq M$ a finite point set in the bounded Euclidean space M and denote $B_p(r) \subseteq S$ the set of points enclosed by a ball with radius r centered at p. Then S has $(\rho, c)-$***expansion*** if and only if $\forall p \in M$ and $r > 0$:

$$|B_p(r)| \geq \rho \implies |B_p(2r)| \leq c \cdot |B_p(r)| \tag{1}$$

The constant c is referred to ***expansion rate*** and ρ is usually set to be $O(\log |S|)$. We say the expansion rate is *low* if $c = O(1)$. Intuitively, the low expansion property ensures the points distributed uniformly in the space.

Similarly, the ***aspect ratio*** Δ is defined as:

$$\Delta = \frac{\max d(x, y)}{\min d(x, y)} \quad \forall x, y \in S \tag{2}$$

and is said to be *bounded* if $\Delta < n^c$ holds for some constant $c > 0$.

We will now show that P-Orth tree with the assumption of bounded aspect ratio. Without the assumptions, the tree height becomes $O(\log \Delta)$. We can replace the tree heights in the following analysis to get Thm. 3.1.

Lemma A.1. *The height for P-Orth tree on points P with size n is $O(\log n)$, assuming the low expansion rate and the bounded aspect ratio for P.*

Proof. By the low expansion rate, the H has side length at most a constant fraction of d_{max}, and the recursion stops when two points with distance d_{min} are separated. Since $d_{max}/d_{min} = n^c$ by the bounded aspect ratio, and the splitters cut the H in the spatial median, it takes $O(\log n)$ levels of splitters to reduce the side length of H to d_{min}. The proof follows then. □

With the above lemma, we now show our Orth-tree construction algorithm has $O(n \log n)$work, polylogarithmic span and $O(\text{Sort}(n))$ cache complexity on n points. Here wssume the cache size $M = \Omega(\text{polylog}(n))$ as in [20, 21], by setting the skeleton height $\lambda = \epsilon \log(M)$ for $\epsilon < 1/(2D)$, and chunk size $l = 2^{D \cdot \lambda}$ in the sieving algorithm. Denote $O(\text{Sort}(n)) = O(n/B \cdot \log_M n)$ the optimal cache complexity for sorting [20].

Theorem A.2. *With parameters specified above, a P-Orth tree can be constructed on points P with size n in $O(n \log n)$ work, $O(\log^2 n)$ span and $O(Sort(n))$ cache complexity, assuming the low expansion rate and the bounded aspect ratio for P.*

Proof. Every point is processed at most once in each round, except for the points sieving, where finding the bucket for one point takes $O(\lambda \cdot D)$ work. The algorithm terminates after $O(\log n)/\lambda$ rounds of recursion, which implies $O(\lambda \cdot D) \cdot O(\log n)/\lambda = O(\log n)$ total work per point. Therefore, the total work is $O(n \log n)$.

For the span, practically the tree skeleton construction and processing each block is done sequentially. However, theoretically, they can be parallelized in $O(\lambda \log n)$ and $O(\log n)$ span, respectively [20]. All other operations takes $O(\log n)$ span. In total, the span in each round is $O(\lambda \log n)$. The algorithm has $O(\log n)/\lambda$ rounds of recursion, so the overall span is $O(\log^2 n)$.

Now consider the cache complexity. Both building the tree skeleton and sub-regions computation fully fit in cache. The chunk size $l = 2^{\lambda \cdot D} = M^{\epsilon \cdot D} \leq \sqrt{M}$, which implies that each chunk fully fits in cache. Therefore, the sieving algorithm takes $O(n/B)$ block transfers. All other operations take $O(n/B)$ block transfers, in total $O(n/B \cdot \log n/\lambda) = O(n/B \cdot \log_M n)$ I/Os. □

For batch updates, we assumes the batch size $m = O(n)$, and if $m = \omega(n)$, we simply replace n with $m + n$ in below bounds for insertions, and there is no change for deletions.

Theorem A.3. *The Update (insertion or deletion) of a batch of size $m = O(n)$ on a P-Orth tree of size n can be performed in optimal $O(m \log n)$ work, $O(\log^2 n)$ span, and $O(m(\log(n/m) + (1 + \log_M m)/B))$ cache complexity, assuming the low expansion rate and the bounded aspect ratio for the updated points.*

Proof. We take the insertion as an example, the deletion is similar. For the work, note the tree after updates is same as the one built from scratch on all points, which has height $O(\log(m+n))$ by Lem. A.1. The height difference is $O(\log(m+n)) - O(\log n) = O(1)$. Since each leaf wraps $O(1)$ points, and every point needs $O(\log n)$ work to reach the leaf, the total work is $O(m \log n)$.

The analysis for span is the same as for construction in Thm. A.2.

The cache bound for updates has two parts. The first is sorting within the batch. This part has $O(m(1 + \log_M m)/B)$ cache complexity. The second part is accessing the tree nodes in the original P-Orth tree. Finding m leaves in a tree of size n will touch $O(m \log(n/m))$ tree nodes [18]. Putting both cost together gives the stated cache complexity. □

Replacing all the tree height $O(\log n)$ with $O(\log \Delta)$ gives Thm. 3.1, without the assumption of bounded aspect ratio.

B Analysis on SPaC-Trees

B.1 Correctness

We prove the correctness of the update algorithms for SPaC-trees by showing its equivalence to that of the PaC-tree. Here we discuss the insertion algorithm, and deletion can be shown similarly. First, the Alg. 3 constructs the same tree as the PaC-tree, so the tree returned in line 6 and line 12 remains the same. The split key in line 13 is same for both trees, therefore the SPaC-tree will insert same points in leaves as the PaC-tree in line 9, but keep the points unsorted. The Join and RightJoin operations (line 19 and line 21) are identical for both tree. In this case, the tree split will reach the same leaves in both trees, and line 34 and line 43 ensure the points order in SPaC-tree to be identical to those in PaC-tree before further proceeding. The other operations in Expose and Node remains the same, and the correctness follows.

B.2 Cost Analysis

Theorem B.1. *A SPaC-tree with n points can be constructed in $O(n \log n)$ work, $O(\log n)$ span, and $O(Sort(n))$ cache complexity.*

Proof. The HybridSort inAlg. 3 is a simple modification of the sample-sort algorithm [10, 20]—all extra operations (i.e., computing the SFC code and storing the point id) take no additional asymptotic cost. The BuildSorted in Alg. 3 is a parallel divide-and-conquer algorithm that takes $O(n)$ work, $O(\log n)$ span and $O(n/B)$ cache complexity. The proof then follows. □

In the following proof we assume the batch size $m = O(n)$. Note that the following proof depends on the analysis of PaC-tree, which can be found at [24].

Theorem B.2. *A batch update (insertion or deletion) of size m on a SPaC-tree of size n uses $O(m \log n)$ work and $O(\log^2 n)$ span.*

Proof. We first show the span bound. The sorting takes $O(\log m)$ span, and the following points insertion/deletion takes $O(\log n)$ rounds to reach leaves. Expanding the leaf and restore the points order take constant time. Both the Join and RightJoin take $O(\log m)$ span [24] in each round. In total, the entire process has $O(\log m \log n)$ span.

We now show the work bound. Sorting m points takes $O(m \log m)$ work, and each point takes $O(1)$ operation in each round. The leaf expansion takes constant time. The work by Join is asymptotically bounded by the work of RightJoin [24], and the total work of RightJoin is $O(m \log \frac{n}{m})$. The total work therefore is $O(m \log n)$. □

Combining both lemmas gives Thm. 4.1.

C Implementation Details

P-Orth tree. We choose to build $\lambda = 3$ levels for 2D points and $\lambda = 2$ levels for 3D points in the P-Orth tree skeleton, which provides generally good performance on our machine. For each bounding box, we store only the point coordinates (with no extra metadata) of the lower-left and upper-right corners to save memory. A single k-NN query traverse subtrees in increasing order of their minimum distance to the query point, computed by comparing the query point with the bounding box associated with each subtree.

SPaC-tree. The implementation of SPaC-tree builds on the code of PaC-tree [24], but with a careful redesign to optimize performance. Simply treating PaC-tree as a black box introduces overhead from transforming input points into key-value pairs—using the SFC code as the key and the entire point as the value—as suggested in its user manual. We avoid this by redesigning the interface so that SPaC-tree automatically parses the SFC code in each point as the key and treats the remaining attributes as the value. This allows it to operate directly on the input sequence, reducing preprocessing time and memory usage.

We also introduce a heuristic to optimize batch updates when a leaf node overflows. The original approach in Alg. 4 unconditionally rebuilds the parent subtree by invoking Alg. 3. This can be inefficient when many points are affected, because the insertion batch must be merged with the points in the leaves prior to recursive node allocation, even when the batch is already sorted. Hence, it incurs significant overhead. An alternative is to explicitly expose the leaf as a balanced tree with empty leaves and then perform the batch insertion on that subtree. Our method chooses between these strategies via a threshold. If the combined size of the overflowing leaf and the new batch is below a threshold (in our case, 4ϕ), we perform a standard, localized rebuild. Otherwise, we expose the leaf and perform batch insertion on the exposed subtree.

Parameter Choosing. We empirically set the parameters to achieve the best performance on our machine for all implementations. We set the leaf wrap to 40 for both SPaC-tree and CPAM, and 32 for all other baselines. Both SPaC-tree and CPAM use the weight-balanced scheme with balancing parameter $\alpha = 0.2$, i.e., the weights of left and right sub-tree can be differ by at most 20%. For Pkd-tree, we adopt $\alpha = 0.3$ as suggested in their paper.

D Batch Updates

We now provide more experimental results for single batch updates. We evaluate the performance of batch updates by varying the batch size from 10^5 to 10^9 points, with results presented in Fig. 8. The experimental setup consists of an initial tree constructed with 10^9 points. We then perform two separate operations: a batch insertion, which adds new points drawn from the same distribution, and a batch deletion, which removes an equivalent number of existing points from the tree. Smaller batch sizes were omitted from this analysis, as their low computational cost diminishes the practical benefits of parallelism.

All baselines scale well on both single batch insertions and deletions. The SPaC-H-tree is faster than others on all benchmarks, except for the batch deletion on `Uniform`, where it is slightly slower than the P-Orth tree due to the handling of imbalance. The P-Orth tree is slower than the SPaC-H-tree on batch insertions on `Varden`, since it is skewed on highly clustering data, and the tree traversal time incurs more overhead. The Pkd-tree is generally slower than SPaC-H-tree on skewed datasets such as `Sweepline` and `Varden`, since its reconstruction-based balancing scheme is more expensive when the rebuilt sub-tree is large, which is typical on skewed datasets.

E Scalability Test

We evaluate the scalability of tested indexes in construction, insertion, and deletions, which is illustrated in Fig. 9. We use 10^9 2D points. A batch insertion uses a single batch of size 10^7. The data points in Fig. 9 show the speedup over the 1-core performance of SPaC-H-tree, and therefore Fig. 9 also reflects the true efficiency comparison of each index (higher is better). In general, all indexes scale well to 224 hyperthreads, and the difference mainly comes from the work

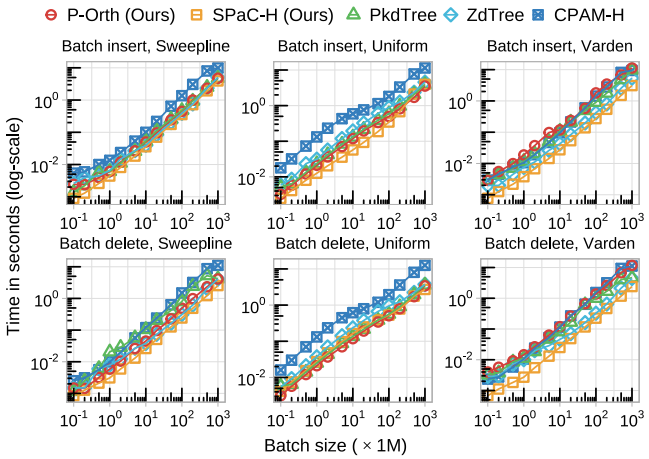

Figure 8. Running time for batch updates on points from synthetic datasets on a tree with 1000M points in 2 dimensions. Lower is better. The batch insertion is to insert another batch from same distribution, and the batch deletion is to delete a batch from the existing points. The batch size is the number of points in the batch (× 1M). Plots are in log-log scale.

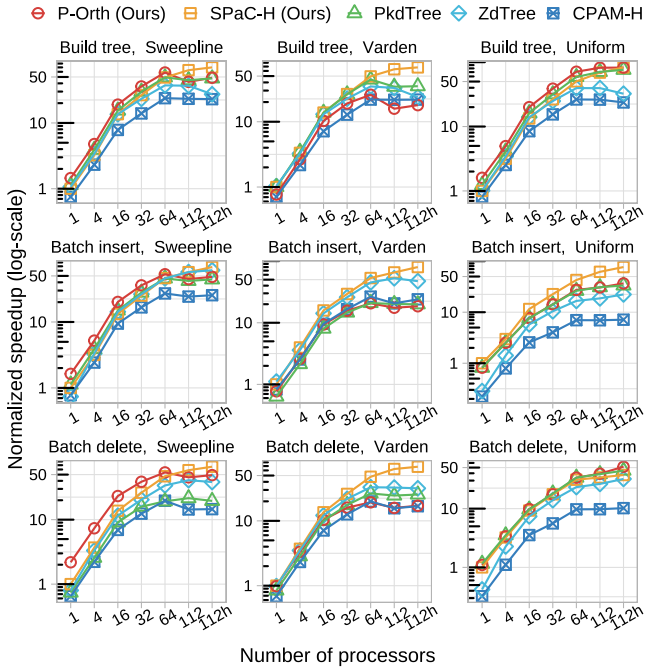

Figure 9. Normalized parallel speedup of tree construction, batch insertion and batch deletion on different number of processors. Higher is better. The speedup is normalized to the running time on the SPaC-tree on 1 thread respectively. The dataset contains 1000M points in 2 dimensions. "Batch insert" is to insert another 1% points into a tree containing 1000M points, and "Batch delete" is to delete 1% points from the tree. Both in a single batch. "112h" means using all 112 cores (224 hyper-threads). There is no result for Boost since it is sequential.

(i.e., one-core performance). Among them, SPaC-H-tree has the best self-relative speedup, which is up to 82.9× in build and 80× in insertion. This is likely due to its simple structure as a 1D search tree. Combining both low work and good scalability, SPaC-H-tree has the best overall construction and update performance. The scalability on batch deletion is similar to that of batch insertion, where the SPaC-H-tree has the best scalability on 112 cores, which is 67× on `Sweepline`, 37.4× on `Varden` and 68.4× on `Uniform`. The P-Orth tree have good scalability on `Uniform`, but is slightly worse on `Sweepline` and `Varden` due to the imbalanced tree.

F Performance on 3D Synthetic Datasets

We now provide more experimental results on 3D synthetic datasets. We generate 3D synthetic datasets with the same method as in 2D, but limit the coordinates range within $[0, 10^6]$ to make it compatible to the SPaC-H-tree. The experiments set up is the same as in 2D Sec. 5. The results are shown in Fig. 10. We omit other baselines since they have been shown to be slower as in 2D case Fig. 3 .

For tree construction, SPaC-H-trees remain the fastest ones, and the time is more close to the 2D case compared with P-Orth trees and Pkd-trees, since the SFC-based indexes are less sensitive to the dimensionality. As a result, SPaC-H-trees are 1.3–2.2× faster than P-Orth trees and 1.2–2.1× faster than Pkd-trees.

Regarding the batch updates, the SPaC-H-tree remains the fastest one on most of the benchmarks, except on `Uniform` where it is slightly slower than the P-Orth tree due to handling of imbalance. P-Orth trees are faster than Pkd-trees on all benchmarks. The reasons are 1) the P-Orth tree does not need to handle the imbalance, and 2) the range of coordinates is limited to $[0, 10^6]$, which enables the tree height of P-Orth trees become much smaller than it in 2D cases (the data range is $[0, 10^9]$), so that the tree traversal time is much reduced in the skewed data such as `Sweepline` and `Varden`. However, Pkd-trees still keep the advantage on queries—the fastest one on most of the benchmarks, and competitive on the rest.

		Build time	Query after Build (100%)				Incremental Insert				Query after Inc. Ins. (50%)				Incremental Delete				Query after Inc. Del. (50%)			
			10NN		Range		Batch Size				10NN		Range		Batch Size				10NN		Range	
			InD	OOD	Cnt	List	10%	1%	0.1%	.01%	InD	OOD	Cnt	List	10%	1%	0.1%	.01%	InD	OOD	Cnt	List
Uniform	★P-OrthTree	5.18	.924	.904	**.701**	1.93	5.69	12.6	24.1	35.2	.988	.967	**.715**	1.80	5.93	**12.5**	**23.8**	**35.4**	1.07	1.06	.964	2.25
	★SPaC-H	**4.08**	7.34	7.29	.968	1.93	**4.70**	**7.09**	**14.7**	**28.8**	7.02	7.04	1.00	2.08	5.78	17.6	33.4	49.9	7.06	7.09	.993	2.05
	PkdTree	4.70	**.916**	**.894**	.758	**1.80**	5.74	13.41	24.5	36.9	**.983**	**.935**	.772	**1.77**	**5.41**	13.0	25.9	61.3	**.916**	**.893**	**.772**	**1.79**
Sweepline	★P-OrthTree	6.41	.637	1.373	.877	2.03	6.44	6.33	6.58	19.62	**.666**	1.97	**.830**	**1.80**	4.30	5.15	**5.93**	21.1	.666	1.98	**.831**	**1.80**
	★SPaC-H	**3.90**	3.84	**1.04**	1.13	2.02	**3.76**	**4.03**	**5.81**	**13.0**	4.66	**1.57**	1.15	2.18	**3.01**	**3.30**	6.13	**20.5**	3.86	1.26	1.14	2.16
	PkdTree	5.57	**.603**	1.83	**.854**	**1.87**	16.4	21.1	28.5	69.0	1.010	6.71	1.039	1.98	14.6	28.5	38.0	65.8	**.647**	**.346**	.867	1.83
Varden	★P-OrthTree	7.5	.257	.214	**.552**	1.83	6.7	8.9	11.5	19.0	.263	.224	**.593**	1.71	6.04	7.78	9.7	17.3	.264	.238	.640	1.82
	★SPaC-H	**3.37**	2.33	1.59	.833	1.84	**3.81**	**4.40**	**5.66**	**10.8**	2.52	1.59	.847	1.98	**4.05**	**5.32**	**6.66**	**11.3**	1.87	1.30	.836	1.98
	PkdTree	7.04	**.152**	**.159**	.599	**1.72**	11.8	16.3	19.6	33.8	**.157**	**.167**	.628	**1.70**	6.20	8.9	12.0	23.7	**.160**	**.164**	**.613**	**1.69**

The **fastest time** is in bold and underlined : within 1.1x the fastest : within 2x the fastest : within 5x the fastest : > 5x the fastest

Figure 10. Running time (in seconds) on 3-dimensional synthetic data. Lower is better. The fastest time in each test is in bold and underlined. We use colors to mark results within 1.1×, 2×, 5×, and > 5× the fastest time. Detailed settings for build, queries, and incremental insertion/deletion are introduced at the beginning of Sec. 5.1. InD/OOD: in-/out-of-distribution.