A Shard-Local Hybrid k-d Tree with Deferred Deletion

***Abstract*—We propose a scalable hybrid k-dimensional tree structure optimized for multicore systems, designed to support exploring spatial indexing. The key idea is to partition the data space into multiple disjoint shards, each maintained independently as a mini k-d tree within a `ConcurrentHashMap`. Insertions and search queries leverage atomic operations to ensure efficient parallelism, while deletions are performed by lazy synchronization. An organizing thread periodically reclaims logically deleted nodes, and allows active operations to continue without disruption. A key objective of the design is to eliminate global locks and reduce contention by employing shard-local updates. Experimental evaluations show that our design achieves strong scalability as compared to state-of-the-art implementations. The combination of shard-based partitioning, lock-free operations, and lazy deletion enables the design to balance concurrency, performance, and maintainability effectively.**

***Index Terms*—Multi-Threading, Synchronization, Concurrent k-d trees, Contention-Efficient, Spatial Index**

I. Introduction

As multicore processors have become the norm across modern computing platforms, the demand for scalable concurrent data structures has grown substantially. Such data structures are crucial for enabling performance-critical applications that must operate efficiently under increasing levels of parallelism. Domains such as real-time analytics, geospatial databases, and machine learning pipelines routinely deal with large volumes of high-dimensional data and require low-latency, high-throughput processing capabilities. In many of these settings, *k*-d trees are a go-to choice for supporting spatial queries, including nearest-neighbor and range searches. Their utility spans a broad range of applications, including computer vision [1], [2], robotics [3], [4], recommendation systems, and geographic information systems (GIS) [5]–[8].

However, despite their conceptual simplicity and effectiveness in single-threaded environments, traditional *k*-d trees face significant challenges when deployed under concurrent workloads. Chief among these are synchronization bottlenecks and inefficient support for dynamic updates, particularly deletions. Most concurrent implementations [9] adopt coarse-grained locking mechanisms that introduce high contention among threads and severely limit scalability on multicore systems. Although some approaches attempt to reduce contention using finer-grained locks or balancing heuristics, these often come at the cost of increased algorithmic complexity and global coordination overhead. The situation is further complicated when deletions trigger structural rebalancing, which requires additional synchronization and may block ongoing queries or insertions.

Moreover, general-purpose concurrent trees such as AVL trees, Red-Black trees, and B-trees [15]–[18], [21], [22] are tailored primarily for one-dimensional ordered data and are not naturally suited for spatial indexing. These structures typically do not support multi-dimensional spatial queries, making them ill-equipped for the kinds of workloads encountered in vision, robotics, and spatial databases. As a result, our evaluation centers on the LFkD-tree [9], a concurrent and linearizable *k*-d tree that represents the most relevant baseline for comparison in this context.

To address the aforementioned limitations, we propose a novel concurrent spatial indexing structure called the **Shard-Local Hybrid *k*-d Tree with Lazy Deletion (SLHkD-tree)**. The core idea is to partition the multidimensional space into independent shards using a lightweight hashing function. Each shard is maintained as a local mini *k*-d tree, stored in a `ConcurrentHashMap`, and managed independently without requiring global coordination. Within each shard, insertions and deletions are implemented using non-blocking atomic primitives such as `compareAndSet`. Deletions are handled via logical marking, and physical node removals are deferred to a background cleanup thread, allowing the structure to remain responsive and non-blocking.

This hybrid strategy of shard-level partitioning combined with lazy deletion leads to several benefits. It reduces synchronization contention, improves cache locality, and allows operations like insert, delete, and contains to scale gracefully with the number of threads. By decoupling logical and physical deletion, SLHkD-tree also amortizes the cost of memory management, ensuring high throughput even in workloads with frequent deletions.

Our empirical evaluations demonstrate that SLHkD-tree achieves up to 2.1× higher throughput than LFkD-tree under balanced workloads, while maintaining strong scalability up to 64 threads. These improvements are consistent across different dataset sizes, dimensionalities, and workload distributions, making SLHkD-tree a practical and efficient choice for real-time concurrent spatial indexing on multicore architectures.

II. Related Work

A. Concurrent k-d Trees

The Lock-Free k-d tree (LFkD-tree) by Chatterjee et al. [9] is the most relevant baseline, offering linearizable nearest-neighbor and range queries. However, its centralized structure often becomes a hotspot under contention. Lock-based variants [15], [17] mitigate some issues but suffer from blocking

and deadlocks. Our SLHkD-tree differs by introducing shard-local partitioning: each shard is maintained as an independent mini k-d tree with lock-free updates and lazy deletion, avoiding global coordination.

B. Concurrent Ordered Data Structures

Several concurrent balanced trees such as AVL, red–black trees, treaps, and skip lists [15]–[18], [21], [22] provide scalable set operations for one-dimensional keys. While valuable for ordered sets, they do not directly support multi-dimensional nearest neighbor or range search. Our approach addresses this gap by combining lock-free primitives with shard-local spatial partitioning.

C. Spatial Indexes

R-trees are widely used for spatial data, and concurrent R-tree variants exist. However, these rely on locks and do not provide the same non-blocking guarantees. LSM-tree–based spatial indexes target disk-resident workloads and batch updates, whereas SLHkD-tree is designed for in-memory, fine-grained concurrency.

D. Recent Advances

Two very recent studies further emphasize the relevance of parallel k-d trees. Men et al. [19] introduce the Pkd-tree, which supports *batch updates* with theoretical guarantees on work and cache complexity. Their design reconstructs a single tree in batches, unlike our shard-local approach that targets continuous online updates. Manohar et al. [20] propose CLEANN, a *lock-free augmented kd-tree* designed for low-dimensional k-NN queries. CLEANN augments a centralized tree with additional metadata, while SLHkD-tree emphasizes scalability via independent shards and lazy deletion. These approaches are complementary, focusing on different concurrency models.

III. Background

This section outlines the core design principles behind our scalable concurrent *k*-d tree architecture based on shard-based partitioning. Each input point is deterministically mapped to a shard using a lightweight hash function called `shardKey()`. This divides the multidimensional space into disjoint regions, each managed independently by a mini *k*-d tree stored in separate buckets of a `ConcurrentHashMap` as shown in Figure 1. This localized structure reduces contention and enables high parallelism. We now describe the shard key computation and walk through an example that illustrates local shard insertions.

To address synchronization bottlenecks in existing designs, our SLHkD-tree partitions data into independent shards, applies lock-free concurrency within each shard, and performs deletions lazily. This combination of fine-grained parallelism and deferred cleanup enables scalable performance for high-dimensional workloads.

Algorithm 1 Shard Key Computation

```
1: procedure SHARDKEY(Point p)
2:     a ← (k > 0)?(p.get(0) ⋙ 8) : 0
3:     b ← (k > 1)?(p.get(1) ⋙ 8) : 0
4:     c ← (k > 2)?(p.get(2) ⋙ 8) : 0
5:     return (a ≪ 16) ⊕ (b ≪ 8) ⊕ c
6: end procedure
```

A. Shard Key Computation

The `shardKey()` function maps each point to a unique shard identifier by extracting and combining the higher-order bits from its first three coordinates. The resulting 24-bit integer serves as a key in the `ConcurrentHashMap`, where each entry corresponds to a shard-local mini *k*-d tree.

This computation groups nearby points (those with similar high bits) into the same shard, preserving spatial locality. The use of bitwise shifts and XORs distributes data uniformly across shards, ensuring load balance and reducing contention. Thus, each shard acts as an independent tree, allowing insertions, deletions, and queries to proceed concurrently without global coordination.

B. Design of SLHkD-tree

Figure 1 illustrates the insertion of several 2D points, showing how the shard key computation directs them into separate mini *k*-d trees. Below is a condensed step-by-step description:

- **(25, 30), (12, 18), (50, 13), (100, 10), (0, 5)** map to shard 0 and form a deeper subtree through alternating x/y splits.
- **(1020, 1234), (1111, 1222), (990, 190), (400, 200)** each map to distinct shard keys, creating separate independent trees.

This example highlights how spatially close points are clustered into the same shard, while distant points are routed to different shards, enabling independent growth and parallel access. Shards evolve independently, avoiding global coordination and allowing high-throughput concurrent operations.

While this example focuses on insertions, SLHkD-tree handles deletions in two phases: nodes are first logically marked as deleted, and later physically removed by a background cleanup thread. This deferred strategy reduces the synchronization overhead and improves overall update performance.

IV. Concurrent Operations

This section describes the internal structure and concurrent behavior of the SLHkD-tree, with a focus on how it supports search, insertion, deletion, and background cleanup. The implementation avoids global synchronization by relying on atomic primitives such as `compareAndSet()` and `AtomicBoolean.set()`. Shard-level partitioning is managed through a `ConcurrentHashMap`, which allows scalable access by leveraging internal striping. For each key operation, we provide algorithmic pseudocode and detailed explanations to highlight how atomic operations and logical deletion collectively enable high concurrency and throughput.

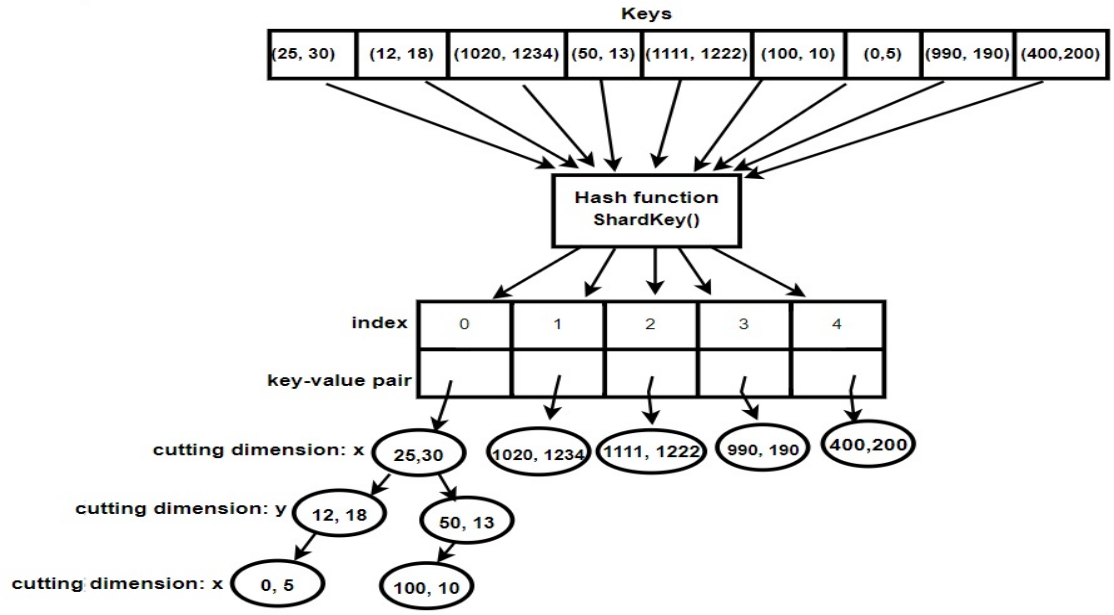

Fig. 1. Insertion of 2D points into shard-local mini k-d trees in SLHkD-tree.

A. Node Structure

TABLE I
FIELDS OF A CONCURRENT SLHKD-TREE NODE

final Point *point*
final int *depth*
volatile AtomicReference<SLHkD-tree Node> *left*, *right*
volatile AtomicBoolean *isDeleted*

Each node in the SLHkD-tree maintains a minimal set of fields, as shown in Table I, to facilitate efficient and thread-safe modifications. The `point` and `depth` fields define the spatial semantics of the node within the k-d tree. The fields `left` and `right` are atomic references to the child nodes, allowing for structural updates through compare-and-set (CAS) operations. The `isDeleted` field is a volatile flag that supports logical deletions.

The use of atomic references ensures that multiple threads can traverse and modify the tree concurrently. Logical deletions allow update and search operations to proceed without blocking, while a background cleanup mechanism eventually reclaims obsolete nodes to maintain structural compactness and reduce synchronization overhead.

B. Contains Operation

The `Contains` operation, shown in Algorithm 2, checks whether a given point is present in the SLHkD-tree. This operation is entirely read-only, enabling multiple threads to execute queries concurrently without interfering with one another or with ongoing updates.

The process begins by verifying that the input point's dimensionality matches the tree's expected dimension k. If not, an exception is thrown to ensure type safety. Once validated, the algorithm computes a shard key using the `shardKey()` function, which maps the point to a specific shard based on the high-order bits of its coordinates. The corresponding shard root is retrieved from the global `shardRoots` map. If the root is not `null`, the search continues within the local mini k-d tree

Algorithm 2 Contains Operation

```
1: procedure CONTAINS(Point p)
2:     if dimensionality of p does not match k then
3:         throw IllegalArgumentException
4:     end if
5:     key ← SHARDKEY(p)
6:     node ← SHARDROOTS.GET(key)
7:     if node ≠ null then
8:         return CONTAINSRECURSIVE(node, p)
9:     else
10:        return false
11:    end if
12: end procedure
```

Algorithm 3 Contains Recursive

```
1: procedure CONTAINSRECURSIVE(Node curr, Point p)
2:     while curr ≠ null do
3:         if not curr.isDeleted.get() and curr.point = p
   then
4:             return true
5:         end if
6:         cd ← curr.depth mod k
7:         if p.get(cd) < curr.point.get(cd) then
8:             curr ← curr.left.get()
9:         else
10:            curr ← curr.right.get()
11:        end if
12:    end while
13:    return false
14: end procedure
```

using the recursive helper function `containsRecursive`; otherwise, the method immediately returns `false`, indicating the point is absent.

Algorithm 3 performs the actual tree traversal. Starting from the root of a shard-local mini-tree, the function follows traditional k-d tree search logic adapted for concurrency. At each node, it checks whether the node is not logically deleted and if its point matches the query. If both conditions are satisfied, the search returns `true`.

If the current node does not match, the algorithm determines the comparison dimension based on the node's depth modulo k and navigates either left or right based on the target point's value along that dimension. This process continues until a match is found or the traversal reaches a null reference, in which case the method returns `false`.

Since the entire procedure involves only atomic reads and no structural modifications, `containsRecursive` is fully non-blocking. This design allows for highly scalable and efficient membership queries in read-dominant workloads, achieving low-latency performance even under high concurrency.

C. Insert Operation

Algorithm 4 Insert Operation

```
1: procedure INSERT(point)
2:     if point's coordinate length ≠ k then
3:         throw IllegalArgumentException
4:     end if
5:     key ← SHARDKEY(point)
6:     current ← SHARDROOTS.GET(key)
7:     if current = null then
8:         newNode ← NODE(point, 0)
9:         if SHARDROOTS.PUTIFABSENT(key, newNode) = null
   then
10:            return
11:        end if
12:        current ← SHARDROOTS.GET(key)
13:    end if
14:    INSERTITERATIVE(current, point)
15: end procedure
```

The `Insert` procedure first validates dimensionality and then determines the shard in which the point belongs. If the shard has not yet been initialized, a root node is created and atomically installed using `putIfAbsent`, ensuring that only one thread succeeds in creating the shard under concurrent execution.

Once the shard root is available, the `insertIterative` routine performs a lock-free traversal of the local k-d tree. At each step, the splitting dimension is selected as the node's depth modulo k. If the current node already stores the point but has been marked as deleted, it is reactivated by clearing the deletion flag. Otherwise, the traversal proceeds toward the appropriate child based on coordinate comparison. When the child reference is null, a new node is created and inserted using a single CAS, guaranteeing atomicity without locks.

This design combines shard-level isolation with CAS-based updates, allowing threads to insert concurrently with minimal contention. Because no global synchronization is required, insertions scale naturally with the number of shards while preserving non-blocking progress guarantees.

Algorithm 5 Insert Iterative Operation

```
1: procedure INSERTITERATIVE(current, point)
2:     while true do
3:         cd ← current.depth mod k
4:         if current.point = point then
5:             CURRENT.ISDELETED.SET(false)
6:             return
7:         end if
8:         nextRef ← (point.get(cd) <
   current.point.get(cd))
9:                 ? current.left : current.right
10:        next ← NEXTREF.GET
11:        if next = null then
12:            newNode ← NODE(point, current.depth + 1)
13:            if NEXTREF.COMPAREANDSET(null, newNode) then
14:                return
15:            end if
16:        else
17:            current ← next
18:        end if
19:    end while
20: end procedure
```

D. Delete Operation

Algorithm 6 Delete Operation

```
1: procedure DELETE(Point p)
2:     if p's dimensionality does not match k then
3:         throw IllegalArgumentException
4:     end if
5:     key ← SHARDKEY(p)
6:     root ← SHARDROOTS.GET(key)
7:     if root ≠ null then
8:         DELETERECURSIVE(root, p)
9:     end if
10: end procedure
```

Algorithm 7 Recursive Deletion

```
1: procedure DELETERECURSIVE(Node node, Point p)
2:     while node ≠ null do
3:         if node.point = p then
4:             NODE.ISDELETED.SET(true)
5:             return
6:         end if
7:         cd ← node.depth mod k
8:         if p.get(cd) < node.point.get(cd) then
9:             node ← node.left.get()
10:        else
11:            node ← node.right.get()
12:        end if
13:    end while
14: end procedure
```

The `Delete` procedure also begins by validating dimensionality and identifying the relevant shard. If the shard exists, the recursive search follows the standard k-d tree partitioning rule until either the target node is found or the traversal

terminates. Upon locating the node, deletion is performed logically by atomically setting the `isDeleted` flag.

This lazy deletion approach avoids expensive structural modifications and makes the deletion immediately visible to other threads. Insertions and searches are unaffected, since they can skip or reactivate logically deleted nodes as appropriate. Physical removal of such nodes is postponed to a background cleanup phase, which safely reclaims nodes when they become redundant.

By decoupling logical and physical deletion, the structure reduces contention, supports non-blocking progress, and achieves higher throughput under mixed workloads.

E. Cleanup Operation

Algorithm 8 Recursive Cleanup

```
1: procedure CLEANRECURSIVE(parent, parentRef, node)
2:     if node = null then
3:         return null
4:     end if
5:     leftClean ← CLEANRECURSIVE(node, node.left, node.left.get())
6:     rightClean ← CLEANRECURSIVE(node, node.right, node.right.get())
7:     NODE.LEFT.SET(leftClean)
8:     NODE.RIGHT.SET(rightClean)
9:     if NODE.ISDELETED.GET then
10:        if leftClean ≠ null and rightClean = null then
11:            replacement ← leftClean
12:        else if rightClean ≠ null and leftClean = null then
13:            replacement ← rightClean
14:        else
15:            replacement ← null
16:        end if
17:        if replacement ≠ null then
18:            if PARENTREF.COMPAREANDSET(node, replacement) then
19:                PHYSICALDELETIONS.INCREMENTANDGET
20:                return replacement
21:            end if
22:        end if
23:    end if
24:    return node
25: end procedure
```

The `cleanup` operation is responsible for reclaiming nodes that have been logically deleted. While logical deletion marks a node as inactive, cleanup ensures long-term efficiency by physically pruning such nodes once they are no longer needed. This process runs periodically or in the background and is designed to coexist with concurrent updates without blocking them. The procedure iterates over all shard roots and applies `cleanRecursive`, which traverses each subtree in post-order. During traversal, both children are first processed recursively, and the node's references are updated with cleaned versions. If the current node is logically deleted, it may be replaced by one of its children, provided that at most one child remains. This replacement is performed atomically using a CAS to guarantee consistency in the presence of concurrent modifications. Each successful removal is recorded in a global counter of physical deletions.

By deferring physical removal to this cooperative background phase, the design avoids introducing contention into the fast path of insertions and lookups. At the same time, cleanup gradually reduces memory overhead, improves cache locality, and maintains structural compactness. This separation between logical and physical deletion is central to the scalability of SLHkD-tree, as it enables high-throughput updates while ensuring the tree does not accumulate redundant nodes over time.

F. Nearest Neighbor Search

Algorithm 9 Nearest Neighbor Search

```
1: procedure NEARESTNEIGHBOR(Point target)
2:     if target's dimensionality ≠ k then
3:         throw IllegalArgumentException
4:     end if
5:     best ← ATOMICREFERENCE(null)
6:     bestDist ← ATOMICREFERENCE(Double.MAX_VALUE)
7:     for all root ∈ SHARDROOTS.VALUES do
8:         NEARESTRECURSIVE(root, target, best, bestDist)
9:     end for
10:    return best.get()
11: end procedure
```

Algorithm 10 Recursive Nearest Neighbor Traversal

```
1: procedure NEARESTRECURSIVE(node, target, best, bestDist)
2:     if node = null then
3:         return
4:     end if
5:     if ¬node.isDeleted.get() then
6:         d ← NODE.POINT.DISTANCETO(target)
7:         if d < bestDist.get() then
8:             BEST.SET(node.point)
9:             BESTDIST.SET(d)
10:        end if
11:    end if
12:    cd ← node.depth mod k
13:    near ← (target.get(cd) < node.point.get(cd)) ? node.left.get() : node.right.get()
14:    far ← (target.get(cd) < node.point.get(cd)) ? node.right.get() : node.left.get()
15:    NEARESTRECURSIVE(near, target, best, bestDist)
16:    if |target.get(cd) − node.point.get(cd)| < bestDist.get() then
17:        NEARESTRECURSIVE(far, target, best, bestDist)
18:    end if
19: end procedure
```

The nearest neighbor search identifies the point in the SLHkD-tree that is closest to a given target under the Euclidean metric. To remain correct under concurrent execution, the procedure maintains two atomic references: one to the best candidate point discovered so far and another to the corresponding distance. These shared references allow threads to update results without explicit locks.

The search iterates through all shard-local mini k-d trees maintained by the global map of shard roots. Within each shard, a recursive traversal explores the tree in a best-first manner. Logically deleted nodes are skipped, while valid nodes are compared against the target. If a closer point is found, the candidate and its distance are atomically updated.

To reduce unnecessary exploration, the traversal first descends into the "near" subtree, i.e., the side of the splitting plane where the target lies. The "far" subtree is only visited if its bounding region intersects the current best search radius, ensuring that potentially closer candidates are not overlooked.

This pruning strategy, combined with shard-level parallelism, provides efficient and scalable nearest neighbor queries. By avoiding locks and relying solely on atomic updates, the method integrates seamlessly with the SLHkD-tree's non-blocking design while preserving query correctness in the presence of concurrent modifications.

G. Range Query

Algorithm 11 Range Query

```
1: procedure RANGEQUERY(Point min, Point max)
2:     result ← new empty list
3:     for all root ∈ SHARDROOTS.VALUES do
4:         RANGERECURSIVE(root, min, max, result)
5:     end for
6:     return result
7: end procedure
```

Algorithm 12 Recursive Range Traversal

```
1: procedure RANGERECURSIVE(Node node, Point min, Point max, List out)
2:     if node = null then
3:         return
4:     end if
5:     in ← true
6:     for i = 0 to k − 1 do
7:         if node.point[i] < min[i] or node.point[i] > max[i] then
8:             in ← false
9:             break
10:        end if
11:    end for
12:    if in and ¬node.isDeleted.get() then
13:        OUT.ADD(node.point)
14:    end if
15:    RANGERECURSIVE(node.left.get(), min, max, out)
16:    RANGERECURSIVE(node.right.get(), min, max, out)
17: end procedure
```

The range query retrieves all points that fall within a given axis-aligned hyper-rectangle of k dimensions. Leveraging the shard-based structure of the SLHkD-tree, the query executes independently on each shard-local mini k-d tree, which naturally reduces contention and distributes the workload across shards.

The query begins by initializing an empty result list and traversing all shard roots maintained in the global map. Each root is explored recursively using the `rangeRecursive` procedure. Within a shard, the traversal evaluates whether a node's point lies inside the specified region across all dimensions. Points that fall within bounds and are not logically deleted are appended to the result list.

Traversal proceeds in depth-first order, visiting both left and right subtrees to guarantee that all qualifying points are discovered. Unlike nearest neighbor search, which prunes branches based on geometric distance, the range query is exhaustive by design. Although this introduces additional checks, the combination of shard-level independence and lightweight read-only operations ensures that the query scales effectively under high concurrency.

This design allows the SLHkD-tree to support efficient multidimensional range queries without synchronization bottlenecks, making it suitable for workloads that combine updates with analytical queries over large datasets.

V. Concurrency Correctness and Safety

The SLHkD-tree is designed to provide both *linearizability* and *non-blocking* semantics. This ensures that all operations behave as if they occur atomically at some point between their invocation and response, while allowing threads to complete independently without relying on global locks. This section formalizes the correctness guarantees by identifying linearization points and outlining key safety properties of the structure.

A. Linearization Points

Linearization points (LPs) represent the precise moment during an operation's execution at which its effect becomes visible to other threads. They are critical for reasoning about concurrent correctness [10], [11]. In SLHkD-tree, each operation has a clearly defined LP that coincides with an atomic update or decision point.

For the `contains(p)` operation, the LP occurs when the method finds a node matching the target point that is not logically deleted. This is captured in line 3 of the `containsRecursive` procedure, where the method checks whether `node.point == p` and `isDeleted.get()` return `false`. If this condition holds, the method immediately returns `true`, and that return constitutes the LP. If no valid node is found, the LP occurs at line 11, where the search completes and returns `false`.

For `insert(p)`, the LP depends on whether the insertion initializes a new shard or adds to an existing one. If the shard is not present, the LP is at line 7 of the `Insert` procedure, where the `putIfAbsent()` call successfully installs a new root. Otherwise, insertion proceeds into the shard-local tree, and the LP is at line 13 of the `insertIterative` method, where a successful `compareAndSet()` appends the new node. Both cases reflect the atomic moment the tree structure is extended.

In the `delete(p)` operation, the LP occurs when the node matching the point is found and logically marked as deleted. This happens at line 3 of the `deleteRecursive` procedure, where `isDeleted.set(true)` is invoked. From that moment, the node is logically absent from the

tree. Although the node may remain physically present until a subsequent cleanup, its logical removal is effective immediately, supporting non-blocking progress while maintaining consistency. For read-only operations such as `nearestNeighbor(target)` and `rangeQuery(min, max)`, linearization points are also well defined. These queries do not modify the underlying structure; instead, they traverse shard-local trees using atomic references and ignore nodes marked as logically deleted. In the nearest neighbor search, the LP occurs at the point where the global best candidate is atomically updated with a closer point. For range queries, the LP is reached each time a valid point is collected into the result set. Since updates (`insert` via CAS and `delete` via logical marking) already guarantee their own LPs, queries observe a consistent snapshot relative to these update events. Thus, both NN and range queries maintain linearizability despite concurrent updates.

B. Safety

The SLHkD-tree satisfies the fundamental safety guarantees expected of a concurrent spatial indexing structure. It preserves the spatial partitioning rules of k-d trees by placing each point according to its coordinate along an alternating split dimension, thereby ensuring correct and efficient traversal. To avoid duplicates, the structure revives logically deleted nodes by resetting the `isDeleted` flag when a matching point is reinserted, rather than creating redundant entries. Logical consistency is maintained by decoupling logical deletion from physical removal—nodes marked for deletion remain structurally intact until they are safely reclaimed during background cleanup. This design prevents interference with concurrent operations and avoids race conditions. All structural modifications, including insertions, deletions, revivals, and cleanups, are performed using atomic operations such as `compareAndSet()` and `AtomicBoolean.set()`, ensuring visibility and correctness under multithreaded execution without relying on coarse-grained locks.

VI. Results and Discussion

A. Experimental Setup

We evaluated SLHkD-tree against the state-of-the-art LFkD-tree [9] across a range of workloads, dimensionalities, and data sizes. Experiments were conducted using datasets containing 2 million and 20 million points, spanning dimensionalities from 2D to 16D. Each test was run for 100 seconds while varying thread counts from 1 to 64. Input points were generated uniformly at random. We acknowledge that highly skewed or correlated datasets could increase contention in overloaded shards and lead to deeper shard-local k-d trees. While correctness and non-blocking progress remain unaffected due to CAS-based updates and lazy deletion, performance may degrade under such skew. Adaptive shard management strategies, such as dynamic splitting or workload redistribution, could mitigate these effects; we highlight this as a limitation and an avenue for future work.

We selected LFkD-tree as the primary baseline because it represents the closest concurrent and linearizable k-d tree implementation currently available. While other concurrent data structures like AVL trees, Red-Black trees, B-trees, or R-Trees have been extensively studied for ordered key access, they are not inherently designed for multi-dimensional spatial indexing and thus are not directly comparable for the nearest neighbor and range queries that SLHkD-tree supports. Consequently, LFkD-tree provides the most relevant point of comparison for evaluating the concurrency, scalability, and spatial query performance of our design. All implementations were developed in Java (OpenJDK 19.0.2) and executed on a dual-socket Intel(R) Xeon(R) Platinum 8260 server equipped with 48 physical cores (96 hardware threads), 512 GB of RAM, and running Ubuntu 22.04.2 LTS at 2.40 GHz. Although the system supports up to 96 concurrent threads, we limited our experiments to a maximum of 64 threads. This decision was made to avoid potential interference from operating system services, background processes, and hyperthreading-related variability. The remaining threads were reserved to ensure stable system behaviour and to isolate the benchmarking workloads from non-experimental noise.

We evaluated three workload distributions, represented in the form XC-YI-ZD, where C, I, and D denote the percentages of `Contains`, `Insert`, and `Delete` operations, respectively. The workloads include: 30C-35I-35D (balanced), 50C-25I-25D (query-heavy), and 90C-9I-1D (read-mostly). Metrics reported include throughput measured in millions of operations per second (MOPS), cache efficiency (via `perf` tool [12]), and energy efficiency (via `jRAPL` tool [13], [14]).

B. Throughput Evaluation

Figure 2 presents the throughput comparison for 2D and 4D datasets under varying thread counts (1 to 64) and workloads. In these lower-dimensional settings, SLHkD-tree demonstrates significant performance improvements over LFkD-tree across all workloads. The gains are most evident under balanced workloads (30C–35I–35D), where SLHkD-tree achieves up to $2.1\times$ higher throughput at 64 threads. This is primarily due to reduced contention, as shard-local structures allow threads to operate independently without central coordination. Figure 3 extends this analysis to higher-dimensional datasets (8D and 16D). Even as dimensionality increases, SLHkD-tree continues to maintain its throughput advantage. In read-heavy workloads (90C–9I–1D), it exhibits near-linear scalability, while LFkD-tree performance saturates quickly due to contention at the root and higher internal rebalancing overheads. The advantage also holds under query-heavy configurations (50C–25I–25D), where SLHkD-tree benefits from disjoint access paths to shard-local trees and minimal synchronization overhead. This improvement in throughput is attributed to three main design features: first, the shard-local architecture that minimizes contention; second, the use of atomic operations for updates; and third, the lazy deletion mechanism that decouples structural changes from the critical path. These features ensure high concurrency and minimal coordination overhead.

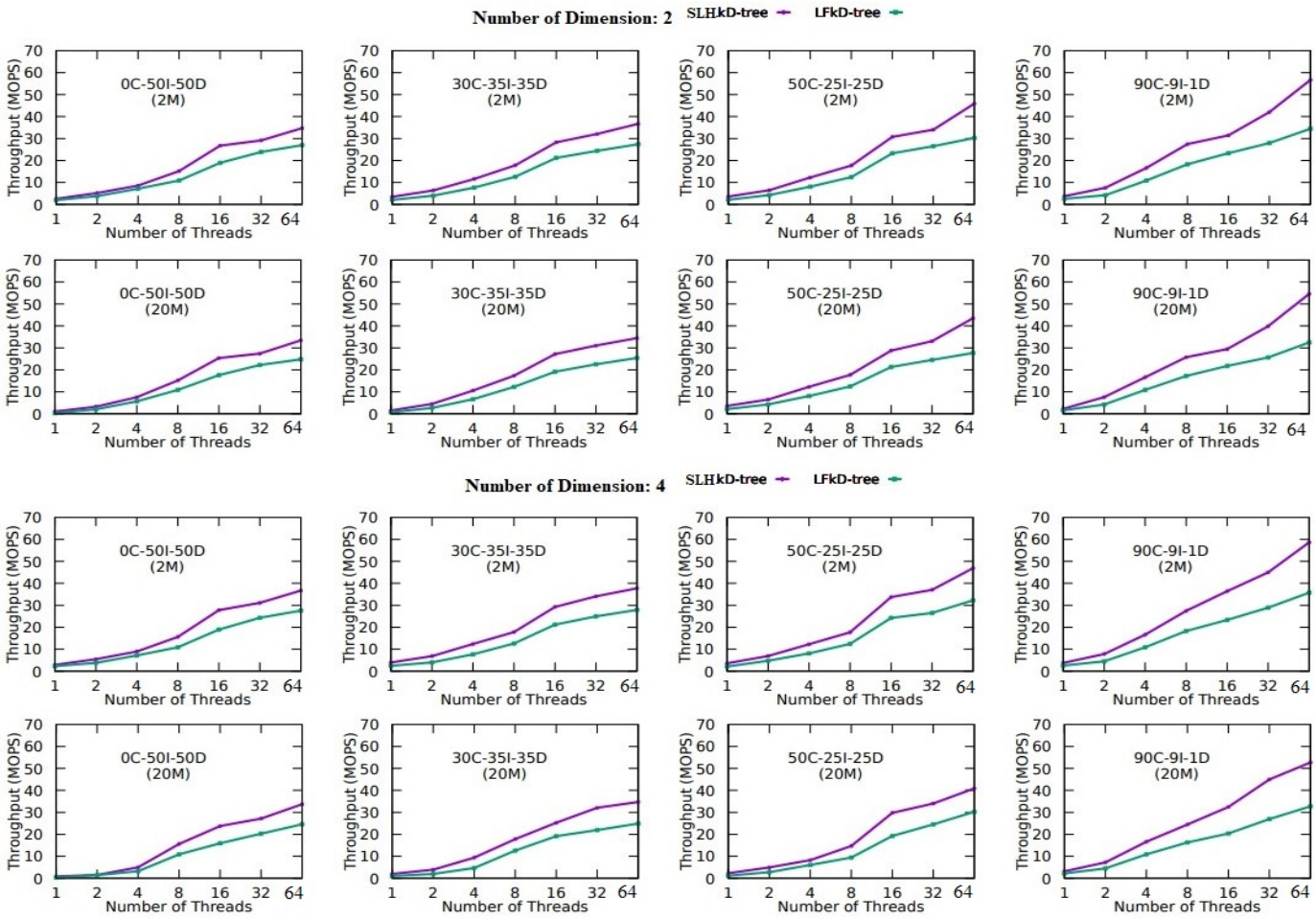

Fig. 2. Throughput comparison (in MOPS) of SLHkD-tree and LFkD-tree under different workloads and thread counts for 2D and 4D datasets. Higher values indicate better performance.

Overall, SLHkD-tree delivers up to a 16.75× improvement over its single-threaded baseline, confirming its ability to scale effectively across various workloads and dimensionalities.

To evaluate cache behavior, we used the Linux `perf` tool to measure the Average Number of Cache Misses per Operation (ANCMP), as shown in Figure 4. Lower ANCMP values indicate better cache locality. SLHkD-tree consistently incurs fewer cache misses than LFkD-tree across all workloads. This benefit stems from its shard-based memory layout, where the `shardKey()` function spatially partitions data, allowing threads to operate on independent memory regions and thereby reducing cache line contention.

Additionally, the lazy deletion mechanism helps preserve subtree stability by avoiding unnecessary pointer rewiring and rebalancing. In contrast, LFkD-tree's centralized structure leads to frequent cache invalidations and increased memory contention under high concurrency.

C. Cache Memory Performance Using `perf`

To assess cache behavior, we used the Linux `perf` tool [12] to measure the Average Number of Cache Misses per Operation (ANCMP), shown in Figure 4. Lower values indicate better cache locality and less contention.

Across all workloads, SLHkD-tree records consistently lower cache misses compared to LFkD-tree. The primary factor behind this is its sharded design. By mapping points to independent mini-trees via the `shardKey()` function, memory accesses become localized within each shard. As a result, operations on different threads tend to touch disjoint memory regions, reducing inter-core interference and cache invalidation.

Additionally, SLHkD-tree's lazy deletion minimizes structural mutations, preserving subtree shapes for longer durations and improving temporal locality. In contrast, LFkD-tree's centralized structure experiences more frequent cache line contention as thread count increases, particularly during updates.

D. Energy Consumption Measurement Using `jRAPL`

Figure 5 presents the Performance per Joule (PPJ) results, quantifying energy efficiency. SLHkD-tree consumes less energy [13], [14] per operation across all thread counts, especially in high-concurrency scenarios.

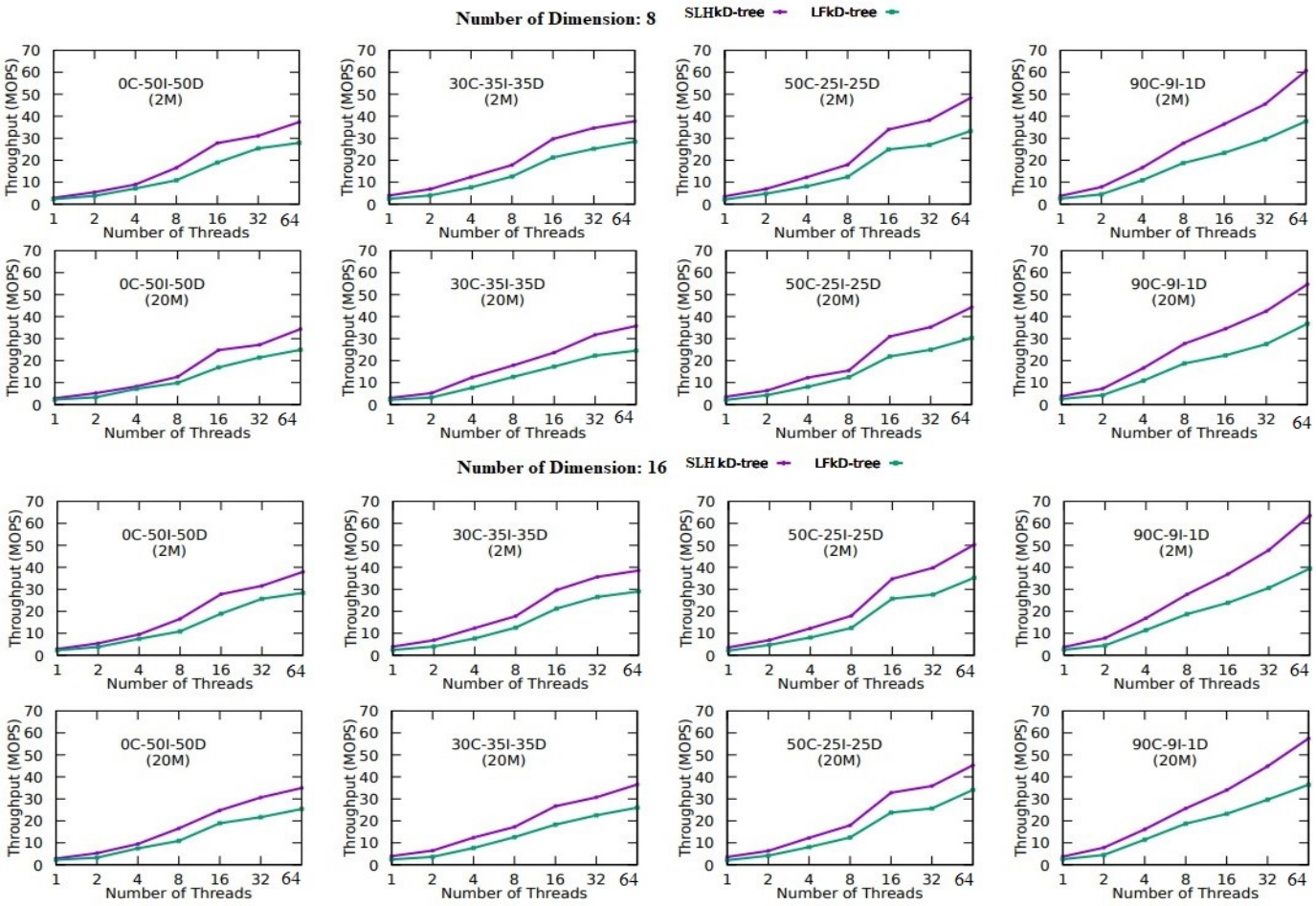

Fig. 3. Throughput comparison (in MOPS) of SLHkD-tree and LFkD-tree under different workloads and thread counts for 8D and 16D datasets. Higher values indicate better performance.

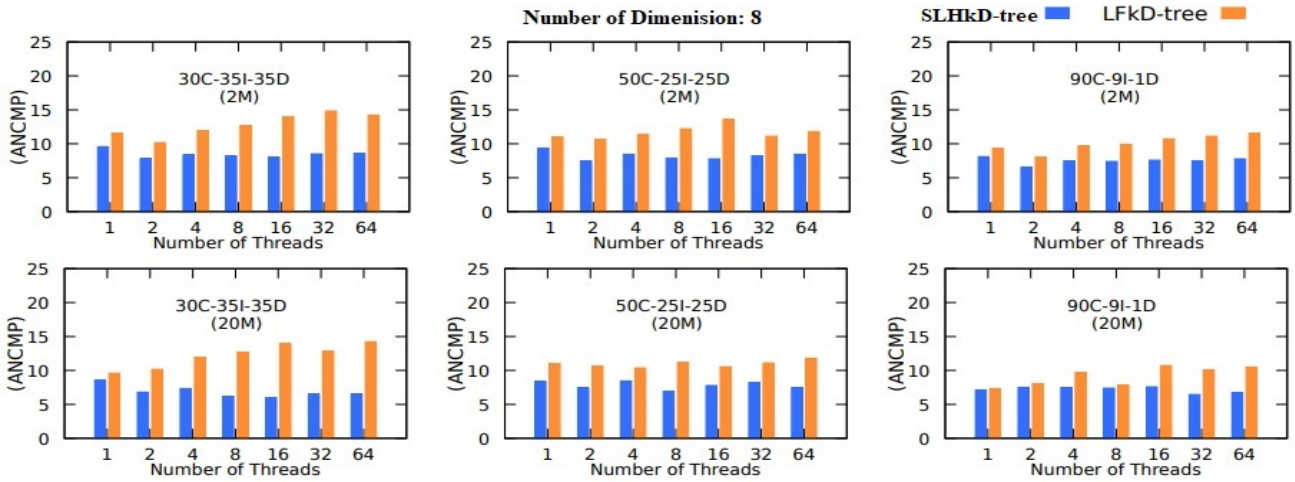

Fig. 4. Cache performance comparison measured as Average Number of Cache Misses per Operation (ANCMP) across all workloads. Lower values indicate better cache locality.

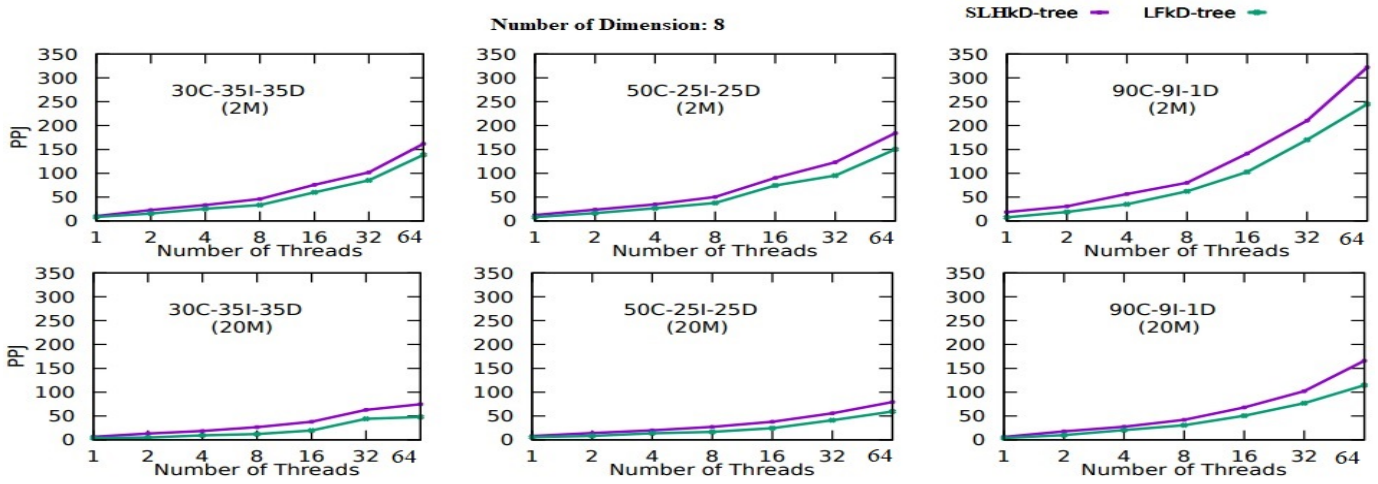

Fig. 5. Energy efficiency comparison measured as Performance per Joule (PPJ) across workloads, dimensions, and thread counts from 1 to 64. Higher values indicate better efficiency.

This improvement stems from multiple sources: (i) reduced contention minimizes retry loops and wasted CPU cycles; (ii) fewer cache misses reduce costly memory stalls; (iii) atomic compare-and-set operations replace locking-based synchronization. Together, these reduce both dynamic and idle power consumption.

Furthermore, SLHkD-tree's design avoids spinning or blocking, which are common in lock-based structures. Since each shard progresses independently, CPU resources are utilized more productively. Overall, SLHkD-tree delivers more useful work per watt compared to LFkD-tree, making it a better fit for both performance-critical and energy-sensitive applications.

VII. Conclusion

We presented SLHkD-tree, a sharded concurrent k-d tree for high-throughput spatial indexing on multicore architectures. By partitioning the key space into independent mini k-d trees and combining lock-free updates with logical deletion and background cleanup, the design minimizes contention and scales with increasing thread counts. Experimental results show that SLHkD-tree consistently outperforms the state-of-the-art LFkD-tree in throughput, cache locality, and energy efficiency across diverse workloads, dimensions, and dataset sizes, achieving near-linear scalability up to 16D under read-heavy settings. Its shard-local parallelism and lightweight synchronization make it well-suited for real-time analytics and concurrent spatial queries. Future work includes adaptive shard management and dynamic splitting for skewed workloads, NUMA-aware placement to improve hardware locality, evaluation on real-world traces, and broader comparisons with emerging concurrent spatial indexes.

VIII. Acknowledgment

We thank all stakeholders for their valuable support. The complete implementation of the k-d tree, including source code, graphs, and supplementary materials, is available on GitHub:https://github.com/concurrentKdTrees/concurrent-K-d-Trees-Public.git.

References

[1] V. R. Tiwari, "Developments in KD tree and KNN searches," *Int. J. Comput. Appl.*, vol. 975, p. 8887, 2023.

[2] S. S. Aung, I. Nagayama, and S. Tamaki, "A high-performance classifier from K-dimensional tree-based dual-kNN," *IEIE Trans. Smart Process. Comput.*, vol. 7, no. 3, pp. 184–194, 2018.

[3] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Trans. Graph.*, vol. 27, no. 5, pp. 1–11, 2008.

[4] L. Minder and A. Sinclair, "The extended k-tree algorithm," *J. Cryptol.*, vol. 25, pp. 349–382, 2012.

[5] A. C.-C. Yao, "On constructing minimum spanning trees in k-dimensional spaces and related problems," *SIAM J. Comput.*, vol. 11, no. 4, pp. 721–736, 1982.

[6] S. Basnet, L.-K. Soh, A. Samal, and D. Joshi, "Analysis of multifactorial social unrest events with spatio-temporal k-dimensional tree-based DBSCAN," in *Proc. 2nd ACM SIGSPATIAL Workshop on Analytics for Local Events and News*, 2018, pp. 1–10.

[7] L. W. Beineke and R. E. Pippert, "The number of labeled k-dimensional trees," *J. Combin. Theory*, vol. 6, no. 2, pp. 200–205, 1969.

[8] V. Carela-Espanol, P. Barlet-Ros, M. Solé-Simó, A. Dainotti, W. de Donato, and A. Pescapé, "K-dimensional trees for continuous traffic classification," in *Int. Workshop on Traffic Monitoring and Analysis*, Springer, 2010, pp. 141–154.

[9] B. Chatterjee, I. Walulya, and P. Tsigas, "Concurrent linearizable nearest neighbour search in LockFree-kD-tree," in *Proc. 19th Int. Conf. Distributed Comput. Networking*, 2018, pp. 1–10.

[10] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[11] V. Singh, I. Neamtiu, and R. Gupta, "Proving concurrent data structures linearizable," in *Proc. 27th IEEE Int. Symp. Softw. Reliab. Eng. (ISSRE)*, 2016, pp. 230–240.

[12] R. Kufrin, "Perfsuite: An accessible, open source performance analysis environment for Linux," in *Proc. 6th Int. Conf. Linux Clusters: The HPC Revolution*, vol. 151, p. 05, 2005.

[13] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., Le, C.: RAPL: Memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design, pp. 189–194 (2010)
[14] Liu, K., Pinto, G., Liu, Y.D.: Data-oriented characterization of application-level energy optimization. In: Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 18, pp. 316–331. Springer (2015)
[15] Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 343–356 (2014)
[16] Besa, J., Eterovic, Y.: A concurrent red–black tree. Journal of Parallel and Distributed Computing **73**(4), 434–449 (2013)
[17] Alapati, P., Saranam, S., Mutyam, M.: Concurrent treaps. In: Algorithms and Architectures for Parallel Processing: 17th International Conference, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017, Proceedings 17, pp. 776–790. Springer (2017)
[18] K. Sagonas and K. Winblad, "A contention adapting approach to concurrent ordered sets," *Journal of Parallel and Distributed Computing*, vol. 115, pp. 1–19, 2018.
[19] Z. Men, J. Shun, G. E. Blelloch, and Y. Sun, "Parallel k-d Tree with Batch Updates," in *Proceedings of the 2025 ACM SIGMOD International Conference on Management of Data*, 2025.
[20] V. Manohar, R. Sharma, and G. Blelloch, "CLEANN: Lock-Free Augmented Trees for Low-Dimensional k-Nearest Neighbor Search," in *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2025.
[21] Y. Sun and G. Blelloch, "Implementing parallel and concurrent tree structures," PPoPP 2019.
[22] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," ICDCS 2013.