Theoretical Computer Science

Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

--Manuscript Draft--

Manuscript Number:	TCS-D-20-00732
Article Type:	VSI:ICDCN2018
Keywords:	concurrent data structure; kD-tree; nearest neighbor search; similarity search; lock-free; linearizability
Manuscript Region of Origin:	AUSTRIA
Abstract:	The Nearest neighbour search (NNS) is a fundamental problem in many application domains dealing with multidimensional data. In a concurrent setting, where dynamic modifications are allowed, a linearizable implementation of the NNS is highly desirable. This paper introduces the LockFree-kD-tree (LFkD-tree): a lock-free concurrent kD-tree, which implements an abstract data type (ADT) that provides the operations Add, Remove, Contains, and NNS. Our implementation is linearizable. The operations in the LFkD-tree use single-word read and compare-and-swap (CAS) atomic primitives, which are readily supported on available multi-core processors. We experimentally evaluate the LFkD-tree using several benchmarks comprising real-world and synthetic datasets. The experiments show that the presented design is scalable and achieves significant speed-up compared to the implementations of an existing sequential kD-tree and a recently proposed multidimensional indexing structure, PH-tree.

Cover Letter: Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

Bapi Chatterjee

Institute of Science and Technology Austria

Ivan Walulya and Philippas Tsigas

Chalmers University of Technology, Gothenburg, Sweden

This paper was published at International Conference on Distributed Computing and Networking (ICDCN 2018). The work proposed a new lock-free data structure algorithm. The algorithm was evaluated on publicly available data and synthetic data. There was no ethical issue concerning this work.

Email addresses: bhaskerchatterjee@gmail.com (Bapi Chatterjee), {ivanw, tsigas}@chalmers.se (Ivan Walulya and Philippas Tsigas)

Highlight: Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

Bapi Chatterjee

Institute of Science and Technology Austria

Ivan Walulya and Philippas Tsigas

Chalmers University of Technology, Gothenburg, Sweden

We extended the conference publication as the following:

1. The conference paper included only a brief proof-sketch; in this version we have included an extensive proof of the algorithm ab initio. The proof is generic and presents a useful contour for proving correctness – linearizability and lock-freedom – of similar data structures.
2. We have included a lock-free Approximate Nearest Neighbour Search algorithm in this version which was not a part of the conference publication.
3. The details of the operations `Add`, `Rem` and `Contains` and the helping sub-methods therein did not find space in the conference version, which have now been included in this version.
4. As we included a new algorithm: Lock-free Approximate Nearest Neighbour Search in this version, we reevaluated each of the compared algorithms.
5. A thorough proofread was done.

Email addresses: `bhaskerchatterjee@gmail.com` (Bapi Chatterjee), `{ivanw, tsigas}@chalmers.se` (Ivan Walulya and Philippas Tsigas)

Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

Bapi Chatterjee

Institute of Science and Technology Austria

Ivan Walulya and Philippas Tsigas

Chalmers University of Technology, Gothenburg, Sweden

Abstract

The Nearest neighbour search (NNS) is a fundamental problem in many application domains dealing with multidimensional data. In a concurrent setting, where dynamic modifications are allowed, a linearizable implementation of the NNS is highly desirable.

This paper introduces the LockFree-kD-tree (LFkD-tree): a lock-free concurrent kD-tree, which implements an abstract data type (ADT) that provides the operations `Add`, `Remove`, `Contains`, and `NNS`. Our implementation is linearizable. The operations in the LFkD-tree use single-word read and compare-and-swap (`CAS`) atomic primitives, which are readily supported on available multi-core processors.

We experimentally evaluate the LFkD-tree using several benchmarks comprising real-world and synthetic datasets. The experiments show that the presented design is scalable and achieves significant speed-up compared to the implementations of an existing sequential kD-tree and a recently proposed multidimensional indexing structure, PH-tree.

Keywords: concurrent data structure, kD-tree, nearest neighbor search, similarity search, lock-free, linearizability

1. Introduction

1.1. Background

Given a dataset of multidimensional points, finding the point in the dataset at the smallest distance from a given *target point* is typically known as the nearest neighbour search (NNS) problem. This fundamental problem arises in numerous application domains such as data mining, information retrieval, machine learning, robotics, etc.

A variety of data structures available in the literature, which store multidimensional points, solve the NNS in a sequential setting. Samet's book [1] provides an excellent collection of data structures for storing multidimensional data. Several of these have been adapted to perform the parallel NNS over a static data structure. However, both sequential and parallel designs primarily consider the NNS queries without accommodating dynamic addition or removal (modifications) in the data structure. Allowing concurrent dynamic modifications exacerbates the challenge substantially.

The wide availability of multi-core machines, large system memory, and a surge in the popularity of in-memory databases, have led to a significant interest in the index structures that can support the NNS with dynamic concurrent addition and removal of data. However, to our knowledge no complete work exists in the literature on concurrent data structures that support the NNS.

Typically, a hierarchical tree-based multidimensional data structure stores the points following a space partitioning scheme. Such data structures provide an excellent tool to *prune* the subsets of a dataset that do not contain the target nearest neighbour. Thus, a NNS query *iteratively scans* the dataset using such a data structure. The iterative scan procedure starts with an initial guess, at every iteration visits a subset of the data structure (e.g. a subtree of a tree) that can potentially contain a

Email addresses: bhaskerchatterjee@gmail.com (Bapi Chatterjee), {ivanw, tsigas}@chalmers.se (Ivan Walulya and Philippas Tsigas)

better guess, and remained unvisited up until the last iteration, updates the current guess if required, and thereby finally returns the nearest neighbour.

In a concurrent setting, performing an iterative scan along with concurrent modifications, faces an inescapable challenge. Consider the case of an operation *op* performing a NNS query in a hierarchical multidimensional data structure that stores points from $\mathbb{R}^d$ and where Euclidean distance is used. Let $a=\{a_i\}_{i=1}^d \in \mathbb{R}^d$ be the target point of the NNS. Let us assume that $k^*=\{k_i^*\}_{i=1}^d \in \{k : k \text{ is key of a node}\}$ is the nearest neighbour of a at the invocation of *op*. In a sequential setting, where no addition or removal of data-points occurs during the lifetime of *op*, k^* remains the nearest neighbour of a at the return of *op*. However, if a concurrent addition is allowed, a new node with key k^{**} may be added to the data structure in a subset that may already have been visited or got pruned by the completion of the latest iteration step. Clearly, *op* would not visit that subset. Suppose that k^{**} was closer to a compared to k^*, if *op* returns k^*, it is not consistent to an operation which observes that the addition of k^{**} completes before *op*. Similarly, on allowing a concurrent removal operation, a NNS query might return a point as the nearest neighbour wherein the returned point might have got removed before the return of the NNS query itself.

Considering concurrent operations on data structures, *linearizability* [2] is the most popular framework for *consistency*. A concurrent data structure is linearizable if each operation in every execution has a *linearization point*: an atomic point where it appears to take effect instantaneously, between its invocation and response. Thus, forming a sequence of concurrent operations, described by the *real-time order* of the linearization points, we observe that concurrent operations meet their *sequential specifications*. In a concurrent setting, we desire linearizability of a NNS query.

Non-blocking progress guarantees are preferred for concurrent operations. In an asynchronous shared-memory system, where an infinite delay or crash failure of a thread is possible, a lock-based concurrent data structure is vulnerable to pitfalls such as deadlock, priority inversion and convoying. On the other hand, in a *non-blocking* data structure, threads do not hold locks, and at least one non-faulty thread is guaranteed to finish its operation in a finite number of steps (lock-freedom). Wait-freedom is a stronger progress condition that all threads will complete an operation in a finite number of steps.

In recent years, a number of practical lock-free search data structures have been designed: skip-lists [3], binary search trees (BSTs) [4, 5, 6, 7], etc. Despite the growing literature on lock-free data structures, the research community has largely focused on one-dimensional search problems. To our knowledge, no complete design of any lock-free multidimensional data structure exists in the literature.

The challenge appears in two ways: designing a concurrent lock-free multidimensional data structure that supports a NNS and ensuring its linearizability.

One of the most commonly used multidimensional data structures for a NNS is the kD-tree, introduced by Bentley [8]. In principle, a kD-tree is a generalization of the BST to store multidimensional data. Friedmann et al. [9] proved that a kD-tree can process a NNS in expected logarithmic time assuming uniformly distributed data points. Various efforts, including approximate solutions, have contributed to improving the performance of the NNS in kD-trees [10, 11]. Furthermore, several parallel kD-tree implementations have been presented, specifically in the computer graphics community, where the focus is on accelerating the applications, such as the ray tracing in single-instruction-multiple-data (SIMD) programming model [12]. Unfortunately, these designs do not fit a concurrent setting, where linearizability becomes crucial for the correctness of operations. For robotic motion planning, Ichnowski *et al.* [13] used a kD-tree of 3-dimensional data in which they add nodes concurrently. Their application does not require a removal operation and thus maintaining a kD-tree is only about adding nodes using atomic `CAS` primitives, which makes it lock-free. They directly transferred the canonical recursive iterative scan used in sequential kD-trees to their concurrent implementation without discussing its consistency. As discussed before, such a canonical implementation of the NNS, using recursive tree-traversal, is not linearizable.

Contributions: We introduce LockFree-kD-tree (LFkD-tree) - an efficient concurrent lock-free kD-tree (section 2). LFkD-tree implements an abstract data type (ADT) that provides `Add`, `Remove`, `Contains` and `NNS` operations for a multidimensional dataset. We describe a novel lock-free linearizable implementation of `NNS` in LFkD-tree (section 4). We provide a Java implementation of the LFkD-tree (code available at [14]). We evalu-

ate our implementation against that of an existing sequential kD-tree and the PATRICIA-hypercube-tree [15][1] (section 6).

1.2. A high-level summary of the work

The main challenge in implementing a linearizable NNS is to ensure that it is not oblivious to the concurrent modifications in the data structure. An NNS requires an iterative scan, which along with pruning collects an atomic *snapshot*.

In general, concurrent data structures do not trivially support atomic snapshots. Some exceptions are - the lock-based BST by Bronson et al. [16], the lock-free Trie by Prokopec et al. [17] and lock-free k-ary search tree by Brown and Avni [18].

Petrank et al. presented a method to support atomic snapshots in one dimensional lock-free ordered data structures that implement sets [19]. They illustrated their method in lock-free linked-lists and skip-lists. Chatterjee [20] extended the method of [19] to propose lock-free linearizable range search in one dimensional data structures. Winblad [21] presented a lock-based method for concurrent range search.

The main idea in [19, 20] is augmenting the data structure with a pointer to a special object, which provides a platform for an Add/ Remove/ Contains operation to *report* modifications to a concurrent operation performing a full or partial snapshot. Nevertheless, collecting an atomic snapshot of a multidimensional data structure to perform an NNS would be naive. We need to adapt the procedure of iterative scan, which benefits from an efficient hierarchical space partitioning structure, to a concurrent setting.

Our work proposes a solution based on augmenting a concurrent data structure with a pointer to a special object called *neighbour-collector*. A neighbour-collector provides a platform for *reporting* concurrent modifications that can otherwise *invalidate* the output of a linearizable NNS.

Essentially, an operation NNS(a) first searches for an exact match of the *target point* α in the data structure, and if it succeeds, returns α itself as its nearest neighbour. If an exact match is not found, before starting the iterative scan, NNS(a) *announces* itself. The announcement uses a new *active* neighbour-collector that contains the target point α and the current best guess for the nearest neighbour of α. On completing the iterative scan, it *deactivates* the neighbour-collector. A concurrent operation, after completing its steps, checks for any active neighbour-collector, and if found, reports its output if it was a better guess than the current best guess available. Finally, NNS(a) outputs the best guess among the collected and the reported neighbours as the nearest neighbour of α.

Naturally, there can be multiple concurrent NNS operations with different target points, and we must allow each of them to continue its iterative scan, after announcing it as soon as it begins. To handle multiple concurrent announcements, we use a lock-free linked-list of neighbour-collector objects. The data structure stores a pointer to one end of this list, say the *head*. A new neighbour-collector gets added only at the other end, say the *tail*.

Consequently, before announcing a new iterative scan, an NNS operation goes through the list and checks whether there is an *active* neighbour-collector that has the same target point. If an active neighbour-collector is found, it is used for a concurrent *coordinated* iterative scan (explained in the next paragraph). A neighbour-collector is removed from the lock-free linked-list as soon as the associated iterative scan is completed. Hence, at any point in time, the length of the list is at most the number of active NNS operations.

During an iterative scan, a subset of the dataset is pruned depending on whether the distance of the target point from a *bounding box* covering the subset is greater than that from the current best guess. Now, if the current best guess at a neighbour-collector is the outcome of multiple already pruned subsets, an NNS that starts its iterative scan at a later time-point, or is slow (or even delayed), will be able to complete much faster. Thus, the coordination among the concurrent NNS, via their iterative scans at the same neighbour-collector, speeds them up in aggregation.

The basic design of the LFkD-tree is based on the lock-free BST of Natarajan et al. [6]. To perform an iterative scan, we implement an efficient fully *non-recursive traversal* using *parent* links, which is not available in [6]. Thus, to manage an extra link in each node, our design requires extra effort for the lock-free synchronization. The modification operations – Add and Remove– use single-word-sized atomic CAS primitives. The *helping mechanism* is based on *operation descriptors* at the *child-links*.

[1] In this work, we are not interested in an existing parallel or sequential implementation that does not provide a Remove operation, in which case lock-free design poses little challenge. We could find only these two existing implementations that provide Remove along with NNS.

Consequently, extra object allocations for synchronization is avoided. The linearizable implementation of NNS is not confined to the LFkD-tree, and it can be used in a similar concurrent implementation of any other multidimensional data structure available in [1]. Keeping that in view, we describe the NNS operations independently and comprehensively. We implemented the LFkD-tree algorithm in Java. The code is available at [14].

2. LockFree-kD-tree Implementation

2.1. Design Overview of the LFkD-tree

The LFkD-tree is a *point kD-tree* in which each node that stores data is assigned at most one data-point. Typically, to partition $\mathbb{R}^d$, we use *axis-orthogonal hyperplanes* that are given by $x_i = c,\ 1 \leq i \leq d$. The structure and consequently the NNS performance of a kD-tree heavily depends on the *splitting rule* - the procedure to select the partitioning hyperplanes. In a sequential setting, to construct a kD-tree from static data, the partitioning hyperplanes are chosen to coincide with points that belong to the given dataset. In this approach, similar to an internal BST representation [7], each node is used for storing data. However, removing a node from an internal BST is costly, more so in a concurrent setting [5, 7]. With this in mind, we opt for an external BST representation [4, 6] to design the LFkD-tree. In this design, only *leaf-nodes* contain the data-points and *internal-nodes* route a traversal, see fig. 1 (b). More importantly, it gives us the flexibility to compute c and $i : 1 \leq i \leq d$ for a hyperplane $x_i = c$, which may not coincide with a data-point.

To compute the values of c and i, in the scenarios where the entire dataset is available beforehand, a number of splitting rules exist in the literature [9, 10]. These rules focus on the hierarchical partition of a *closed hyperrectangle* that covers the entire dataset, thereby not only tries to balance a kD-tree but also optimize its depth. For example, we can see such efforts in the *sliding-midpoint* rule of Mount and Arya [10], which is also used in the Python kD-Tree library [22]. The *median splitting* rule, was proposed by Friedmann et al. [9]. They suggested to choose i in a way such that a partitioning hyperplane is orthogonal to the axis of the cell along which its data points have the maximum spread, and c is chosen as the coordinate median of the points along that axis.

In a concurrent setting, where we do not have knowledge of the entire dataset in advance, the partitioning hyperplane needs to be computed dynamically and in a very localized fashion. For the LFkD-tree, we formulate a simple and practical splitting rule, namely the ***local-midpoint rule***, as given in section 2.2.

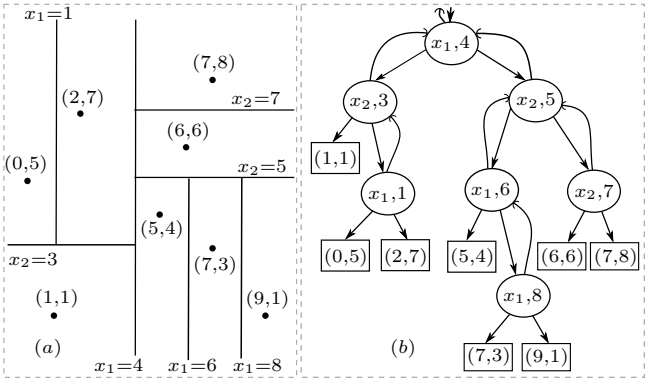

Figure 1: LFkD-tree Structure

A leaf-node of a LFkD-tree Υ, contains a unique data-point as its *key*, whereas, an internal-node corresponds to a partitioning hyperplane. Without any ambiguity, we denote a leaf-node containing key $k = \{k_i\}_{i=1}^d \in \mathbb{R}^d$ by $\mathtt{N}(k)$ (or $\mathtt{N}(\{k_i\}_{i=1}^d)$), and an internal node associated with a hyperplane $x_i = c$, by $\mathtt{N}(i, c)$. An internal-node has three *links* connected to its *left-child*, *right-child* and *parent*. We indicate the link emanating from a node N and terminating at a node M by $\mathsf{N} \rightsquigarrow \mathsf{M}$. Access to Υ is given by the *address* of a unique node *root*. A node N is said to be *present* in Υ ($\mathsf{N} \in \Upsilon$), if it can be *reached* following the links starting from the root.

For each internal-node $\mathtt{N}(i, c)$, Υ maintains the following invariants (for the nodes in the subtrees rooted at $\mathtt{N}(i, c)$): (i) a node $\mathtt{N}(\{k_i\}_{i=1}^d)$ belongs to the *left subtree*, if $k_i < c$, (ii) a node $\mathtt{N}(\{k_i\}_{i=1}^d)$ belongs to the *right subtree*, if $k_i \geq c$ and (iii) both subtrees are themselves LFkD-tree. (i) and (ii) together are called the *symmetric order* of the LFkD-tree. Figure 1 illustrates the structure of a subtree of a LFkD-tree corresponding to a sample 2-dimensional dataset.

2.2. Sequential Behaviour of the ADT Operations

LFkD-tree implements an abstract data type kD-Set. It provides operations `Add`, `Remove`, `Contains` and `NNS`. For each operation, we start with a *query* from the root, traverse down Υ, at an internal node decide left/right child using the symmetric order until arrived at a leaf-node.

To perform `Add`(a), $a \in \mathbb{R}^d$, if the query terminates at a leaf-node $\mathtt{N}(b)$, $b \in \mathbb{R}^d$, and $b = a$ (an

element-wise comparison of keys), Add(a) returns false. However, if $b \neq a$, we allocate a new internal-node N(i, c) with its child links connected to two leaf-nodes N(a) and N(b). If p(N(b)) was the parent of N(b) at the termination of query, we connect the parent link of N(i, c) to p(N(b)). We update the link p(N(b))$\rightsquigarrow$N(b) to point to N(i, c) and return true. To compute i and c, we employ the local-midpoint rule as given below.

Local-midpoint rule: *Let i, $1 \leq i \leq d$ be the index of the coordinate axis along which a and b have the maximum coordinate difference; if there are more than one such axis, select the one with the lowest index. Take the hyperplane as $x_i = \frac{a[i]+b[i]}{2}$.*

To perform Remove(a), if the key of the leaf-node, where the query terminates, is a i.e. N(a) $\in \Upsilon$, we modify the link from the *grandparent* of N(a), denoted by g(N(a)), to its parent, to connect the *sibling* of N(a): s(N(a)) to g(N(a)). Thereafter we return true. If N(a) $\notin \Upsilon$, Remove(a) returns false. To perform Contains(a), using a similar query we check whether N(a) $\in \Upsilon$ and return true or false accordingly.

The operation NNS(a) is non-trivial. On termination of the initial query, if we reach N(b) and $b = a$, clearly the nearest neighbour of a, available in the dataset stored in Υ, is a itself. However, if $b \neq a$, we take b as our *current best guess* and check whether the *other subtree* of p(N(b)) (the current subtree contains the single node N(b)) stores a *better guess*. Suppose that p(N(b)) = N(i, c). Now, any point on the *other side* of the hyperplane $x_i = c$ will be at least at a distance $|a_i - c|$ from the target point $\{a_i\}_{i=1}^{d}$. Therefore, if $|a_i - c| > ||a, b||_2$ (the Euclidean distance between a and b), we must prune the other subtree i.e. the one rooted at s(N(b)), otherwise we visit it in the *next iteration*. A subtree is ensured to be visited and *not re-visited* guaranteeing a traversal back to the root. At the termination of the iterative scan of Υ, the current best guess is returned as the nearest neighbour of a.

2.3. Lock-free Synchronization

As the basic structure of our LFkD-tree is based on an external BST, for the lock-free synchronization in the LFkD-tree, we build upon the lock-free BST algorithm of [6]. The fundamental idea of the design is a *lazy remove* procedure that is essentially based on a protocol of atomically injecting *operation descriptors* on the links connected to the node to be removed, and then modifying those links to disconnect the node from the LFkD-tree. If multiple concurrent operations try to modify a link simultaneously, they synchronize by *helping* one of the pending operations that would have successfully injected its descriptor.

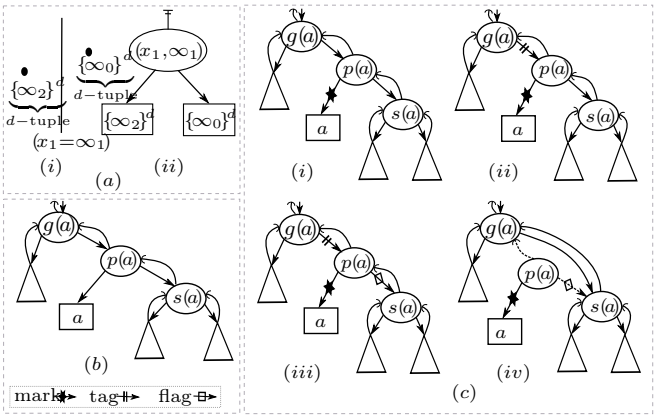

Figure 2: LFkD-tree Structure

More specifically, to Remove the node N(a), as shown in the fig. 2(b), we use CAS primitives to inject *operation descriptors* at the links p(N(a))$\rightsquigarrow$N(a), g(N(a))$\rightsquigarrow p$(N(a)) and p(N(a)) $\rightsquigarrow s$(N(a)), exactly in this order. We call these descriptors mark, tag and flag respectively. An operation descriptor works as an *information source* about the steps already performed in Remove(a) and thus a concurrent operation, if obstructed at a link with descriptor, *helps* by performing the remaining steps. In particular, a mark at a link indicates that the next step would be to inject a tag at the link g(N(a))$\rightsquigarrow p$(N(a)), whereas, a tag indicates that the next step is to inject the descriptor flag at the link p(N(a))$\rightsquigarrow s$(N(a)). Finally, a flag indicates the completion of steps of injecting operation descriptors and thereafter the required link updates are done. The *helping mechanism* ensures that the concurrent Add and Remove operations do not violate any invariant maintained by the LFkD-tree. The steps of a Remove operation are shown in Figure 2(c). An Add operation uses a single CAS to update the target link only if it is free from any operation descriptor, otherwise it helps the pending Remove. A Contains or NNS operation does not perform help.

We call the CAS step, which injects a mark at p(N(a))$\rightsquigarrow$N(a), the *logical remove* of a. After this step, a Contains(a) that reads p(N(a))$\rightsquigarrow$N(a) returns false. Accordingly, Add(a) helps to complete the pending Remove(a), if it reads p(N(a))$\rightsquigarrow$N(a) with a mark descriptor, and then reattempts its own steps. The helping mechanism guarantees that a

logically removed node will be eventually detached from the LFkD-tree.

To realize the atomic step to inject an operation descriptor, we replace a link using a CAS with a single-word-sized packet of a link and a descriptor. Given a pointer delegates a link, a well-known method in C/C++ to pack extra information with a pointer in a single memory-word is *bit-stealing*. In a x86/64 machine, where memory allocation is aligned to a 64-bit boundary, three least significant bits in a pointer are unused. The three operation descriptors used in our algorithm fit over these bits.

For ease of exposition, we assume that a memory allocator always allocates a variable at a new address and thus an ABA[2] problem does not occur. For lock-free memory reclamation in a C/C++ implementation of the algorithm, a method such as one based on reference counting [23] can be used. Whereas, traditionally a Java implementation uses the JVM garbage collector. Furthermore, to avoid null pointers at the beginning of an application, we use a subtree containing an internal-node and two leaf-nodes which work as *sentinel nodes*. See fig. 2(a). The keys in the sentinel nodes maintain $\infty_0 > \infty_1 > \infty_2 > k_i$, $1 \leq i \leq d$, for any data point $\{k_i\}_{i=1}^{d}$ stored in the LFkD-tree. The sentinel internal-node $\mathtt{N}(1, \infty_1)$ works as the root of the LFkD-tree and the entire dataset is stored in its left subtree.

3. Linearizable Add, Remove and Contains Operation

3.1. Description of the Add, Remove and Contains operations

First, we present the node-structures in the LFkD-tree, which will help in the subsequent discussion. The classes **INode** and **LNode**, which represent an internal- and a leaf- node respectively, are shown in lines 1 and 3 in Algorithm 1. Every **INode**, in addition to the fields i and c that represent the associated hyperplane, has three pointers lt, rt and pr that delegate the *left-child*, *right-child* and *parent* links, respectively. An **LNode** contains only an array k to represent a data-point $k=\{k_i\}_{i=1}^{d} \in \mathbb{R}^d$. The node-pointer root, line 5, delegates address of the sentinel node $\mathtt{N}(1, \infty_1)$. As a convention, if x is a *field* of a class C, we use pc·x to indicate the field x of an instance of C pointed by pc; and, the type of a pointer to an instance of C is indicated by C∗. Note that, **INode** and **LNode** inherit **Node**.

```
1  class INode { ▷ A subclass of Node.
     long i; double c;
     Node* lt, rt, pr;
2  }
3  class LNode { ▷ A subclass of Node.
     double[] k;
4  }
5  root := INode*(1, ∞1, null, null, null);
   root·lt := LNode*({∞2}^d);
   root·rt := LNode*({∞0}^d);
   ▷ Return a child-direction.
   Dir(Node* N(i,c)·ref, double[] k)
6    return k[i] < c ? L : R;
     ▷ Directions - L: left, R: right;
     implemented as a boolean.
   ▷ Return a child-pointer.
   Child(Node* pa, dir cD)
7    return cD = L ? pa·lt : pa·rt;
   ChCAS(Node* pa, Node* exp, Node* new, dir
     cD)
8    if (cD = L) and pa·lt = exp then
9      return CAS(pa·lt·ref, exp, new);
10   else if (cD = R) and pa·rt = exp then
11     return CAS(pa·rt·ref, exp, new);
12   else return false;
   ▷ The method Search() is called by Add, Remove
   and Contains operations.
   Search(Node* pa, Node* a, double[] k)
13   while Ptr(a)·class ≠ LNode do
14     pa := Ptr(a); a := Child(pa, Dir(pa, k));
15   return ⟨pa, a⟩;
   ▷ Crates a new internal-node.
   NewNode(Node* a, Node* b, Node* p)
16   ka := a·k; kb := b·k;
17   i := {i : 1≤i≤d and
       |ka[i]−kb[i]|≥{|ka[j]−kb[j]|}_{j=1}^{d}};
     ▷ Local-midpoint rule is applied.
18   c := (ka[i]+kb[i])/2;
19   left := (ka[m] < kb[m] ? a : b);
20   right := (ka[m] > kb[m] ? a : b);
21   return INode(m, c, left, right, p);
```

Algorithm 1. The node structure and *inlined* methods in the LFkD-tree

The method Search(), line 13 to 15, which performs a query, returns the pointers to the leaf-node and its parent, where the query terminates. The method NewNode() allocate a new i.

The operation Add, line 23 to 24, calls AddNode() to get the pointer to the node and its parent, either added by itself or already present there, containing its query key, and the result of addition accordingly. Thereafter, Add calls the method Sync(), line 24, and outputs the result. We describe Sync() in Section 4.

The method AddNode(), line 25 to 38, attempts

[2] ABA is an acronym to indicate a typical problem in a CAS-based lock-free algorithm: a value at a shared variable can change from A to B and then back to A, which can corrupt the semantics of the algorithm.

to add a new node in the LFkD-tree. It starts with calling `Search`(), line 27. If the returned leaf-node-pointer `a` is found containing `mark`, it indicates that the node containing the query key is logically removed, and therefore, the method `Help`() is called to help the concurrent pending `Remove` operation, line 37. Otherwise, the node pointed by `a` is checked whether it contains the query key, line 29, and if found, false is returned, line 30. `AddNode`() also outputs the descriptor-free pointers to the leaf-node and its parent where the query terminated. However, if the leaf-node did not contain the query-key, it is checked whether `a` has the descriptor `flag`, which indicates a pending `Remove` of the *sibling* of the node pointed by `a`; and if `flag` is found, `Help`() is called, line 31. Only in the case `a` is descriptor-free, the method `NewNode`() is called to allocate a new node, and a `CAS` executed in the method `ChCAS`(), called at line 35, modifies `a` to add the new node. On success of the `CAS` it returns `true`. The method `Help`() is described in Section 3.2.

```
22 Add(double[] k)
23   ⟨pa, a, result⟩ := AddNode(k);
24   Sync(pa, a); return result;
   AddNode(double[] k)
25   pa := root; a := pa·lt;
26   while true do
27     ⟨pa, a⟩ := Search(pa, a, k);
28     if !IsMark(a) then
29       if k = Ptr(a)·k then
30         return ⟨Ptr(pa), Ptr(a), false⟩;
31       if IsFlag(a) then pa := Help(pa, a);
32       else
33         n := LNode(k); cD := Dir(pa, k);
34         newNd := NewNode(a, n·ref, pa);
35         if ChCAS(pa, a, newNd·ref, cD) then
36           return ⟨newNd·ref, n·ref, true⟩;
37     else pa := Help(pa, a);
38     a := Child(pa, Dir(pa, k));
   Remove(double[] k)
39   pa := root; a := pa·lt;
40   while true do
41     ⟨pa, a⟩ := Search(pa, a, k);
42     if !IsMark(a) then
43       if k ≠ Ptr(a)·k then return false;
44       if IsFlag(a) then pa := Help(pa, a);
45       marker := Mark(a); cD := Dir(pa, k);
46       else if ChCAS(pa, a, marker, cD) then
47         Help(pa, a); return true;
48     else return false;
49     a := Child(pa, Dir(pa, k));
   Contains(double[] k)
50   pa := root; a := pa·lt;
51   ⟨pa, a⟩ := Search(pa, a, k);
52   if !IsMark(a) then
53     Sync(Ptr(pa), Ptr(a));
54     return k = Ptr(a)·k ? true : false;
55   else return false;
```

Algorithm 2. The `Add`, `Remove` and `Contains` operations in LFkD-tree

The `Remove` operation, line 39 to 49, performs query in a similar way calling `Search`(), line 41. At the return of `Search`(), if `a` is found to have `mark`, it indicates that even if the query key k was present in the LFkD-tree, it was already logically removed and therefore `Remove` returns false, line 48. If `a` is free of `mark`, we check if the node pointed by `a` contains the query key, and if not, `Remove` returns false, line 43. However, if the pointer `a` is found to have the descriptor `flag`, it indicates a pending `Remove` of the sibling of the node pointed by `a`, and therefore we call the method `Help`() to perform helping steps. After return of `Help`(), the steps are reattempted. Finally, if `a` was descriptor-free, `mark` is injected on it via the method `ChCAS`(), line 46, and if it succeeds, the `Help`() is called to take further steps and `true` is returned, line 47.

A `Contains`, line 50 to 55, calling `Search`(), returns `true` only if the pointer `a` does not have `mark` and the query key matches at the leaf-node pointed by `a` at line 54; else it returns false, line 55. Similar to `Add`, `Contains` also calls `Sync`(), which will be explained in Section 4.

3.2. The Helping steps

In algorithm 3, the method `Help`(), line 56 to 61, is called at a pointer to a leaf-node which had been injected with either the descriptor `mark` or `flag`. Therefore, it first decides the type of descriptor, and then accordingly calls either `HelpMrk`(), line 60, or `HelpFlg`(), line 61.

The method `HelpMrk`(), line 62 to 65, first calls `ApndTag`() to fix the $g(\mathtt{N}(a))$, pointed by `ga`. And then calls `HelpTag`() to complete the remaining steps of `Remove`. To distinguish between the `tag` put by the `Remove` of left and right child of $p(\mathtt{N}(a))$, we use two types of `tag`: `ltag` and `rtag`. In the method `ApndTag`(), line 75 to 88, if the link was found already tagged, the type of `tag` (`ltag` or `rtag`) is read using the method `TagDir`. And, if the link was found to be tagged by a `Remove` of the other child of $p(\mathtt{N}(a))$, first that `Remove` is helped and then we reattempt, line 81, otherwise we return `ga`, line 80. However, if the link $g(\mathtt{N}(a))\rightsquigarrow p(\mathtt{N}(a))$ is found `flagged`, line 84, it indicates a pending `Remove` of $s(p(a))$ and therefore we help it before reattempt. On successfully `tagging` the link $g(\mathtt{N}(a))\rightsquigarrow p(\mathtt{N}(a))$, we return the pointer `ga`, line 86. Also, if $g(\mathtt{N}(a))$ is found not connected with $p(\mathtt{N}(a))$,

```
Help(Node* pa, Node* a)
56  cD := (a·k[pa·i] < pa·c) ? L : R;
57  if IsFlag(a) then
58  | ga := pa·pr; sa := Child(pa, !cD);
59  | pD := (a·k[ga·i] < ga·c) ? L : R;
60  | return HelpFlg(ga, pa, sa, pD);
61  else return HelpMrk(pa, a, cD);
HelpMrk(Node* pa, Node* a, dir cD)
62  ga := ApndTag(pa, a, cD);
63  pD := Dir(ga, a·k); pl := Child(ga, pD);
64  if Ptr(pl) = pa then HelpTag(ga, pl, pD);
65  return ga;
ApndFlg(Node* pa, dir sD)
66  while true do
67  | sa := Child(pa, sD);
68  | if IsMark(sa) then return sa;
69  | else if IsFlag(sa) then return Ptr(sa);
70  | else if IsTag(sa) then HelpTag(pa, sa, sD);
71  | else if ChCAS(pa, sa, Flag(sa), sD) then
72  | | return sa;
HelpTag(Node* ga, Node* pl, bool pD)
73  pa := Ptr(pl); sD := (TagDir(pl) = L ? R : L);
74  HelpFlg(ga, pa, ApndFlg(pa, sD), sD);
ApndTag(Node* pa, Node* a, dir cD)
75  while true do
76  | ga := pa·pr; pD := Dir(ga, a·k);
77  | pl := Child(ga, pD);
78  | if Ptr(pl) = pa then
79  | | if IsTag(pl) then
80  | | | if TagDir(pl) = cD then return ga;
81  | | | else HelpTag(ga, pl, pD);
82  | | else if IsFlag(pl) then
83  | | | grGa := ga·pr;
84  | | | HelpFlg(grGa, ga, pa, Dir(grGa, a·k));
85  | | else if ChCAS(ga, pl, Tag(pl, cD), pD) then
86  | | | return ga;
87  | | else if pl = a then pa := ga;
88  | | else return ga;
HelpFlg(Node* ga, Node* pa, Node* sa, dir pD)
89  if Ptr(pl := Child(ga, pD)) = pa then
90  | if Ptr(sa)·pr = pa then
91  | | CAS(Ptr(sa)·pr·ref, pa, ga);
92  | ChCAS(ga, pl, sa, pD);
93  return ga;
```

Algorithm 3. `Help()` Method of Algorithm 2

we return `ga`, line 88, and `Remove` operation terminates because it indicates the completion.

The method `HelpTag()`, line 73 to 74, reads the direction of the child whose `Remove` had tagged the link $g(\mathtt{N}(a))\rightsquigarrow p(\mathtt{N}(a))$ (represented by pl), line 73, `flags` the (sibling) link calling `ApndFlg()` and finally calls `HelpFlg()` to perform the remaining steps, see line 74.

In `ApndFlg()`, line 66 to 72, if the link $p(\mathtt{N}(a))\rightsquigarrow s(\mathtt{N}(a))$ (represented by `sa`) was found `marked`, line 68, we return this link as it was, because it is guaranteed that the `Remove` operation that `marked` this link, will perform helping before reattempting its `CAS` to put a `tag` in the method `ApndTag()`. In that case, the `marked` link is further carried to the method `HelpFlg()` and connected to $p(\mathtt{N}(a))$. If $p(\mathtt{N}(a))\rightsquigarrow s(\mathtt{N}(a))$ is found flagged, we return $s(\mathtt{N}(a))$, represented by the value of `sa` without any descriptor i.e. `Ptr`(sa), line 69. On a successful `CAS` to `flag` the link, we return address of $s(\mathtt{N}(a))$ represented by `sa`, line 72.

Finally, the method `HelpFlg()`, line 89 to 92, if required, connects the `pr` pointer of $s(\mathtt{N}(a))$ to $g(\mathtt{N}(a))$, see line 91. And lastly, node a is detached from the LFkD-tree by connecting $s(\mathtt{N}(a))$, represented by sa, to $g(\mathtt{N}(a))$ using a `CAS` at line 92.

4. Linearizable Nearest Neighbour Search

In this section, we begin with the algorithm that addresses the case where concurrent `NNS` operations have coinciding target points. We build on it to present the algorithm for general cases without any restriction. However, first we discuss the linearization scenario to present a precise picture of the challenge that our design addresses.

4.1. Linearization argument

As an illustration, consider Figure 3. In this example, an `NNS`(6.4) operation op_1 by thread T_1 is concurrent with modification operations by thread T_2 in the LFkD-tree. T_2 completes operation op_2: `Add`$(6, 4)$, to a sub-tree that has already been traversed by T_1, then proceeds to complete operation op_3: $Add(5.5, 4)$ to a sub-tree that is yet to be traversed by T_1. Thus, T_1 observes the operation op_3 but not op_2, even though, to T_1, $op_2 \rightarrow op_3$ (op_2 precedes op_3). In case op_1 returns $(5.5, 4)$ as the nearest neighbour, then the operations op_1, op_2 and op_3 can not be linearized as explained in the Section 1.1. Thus, op_2 essentially needs to *report* its modification to op_1, after completing its own steps.

Suppose that op_2 got delayed after adding a new node $\mathtt{N}(6, 4)$ to the LFkD-tree and could not report it to op_1. If a concurrent `Contains` operation, say op_4 by thread T_3, reads node $\mathtt{N}(6, 4)$ and later makes modifications to the tree that are observable to op_1 and thus linearizable before op_1. Similarly, operations op_4 and op_1 can not be ordered sequentially without violating linearizability.

Therefore, op_4 also needs to report its output to op_1. Now, given that op_2, op_3 amd op_4 are made to report their modifications to op_1, we need to change

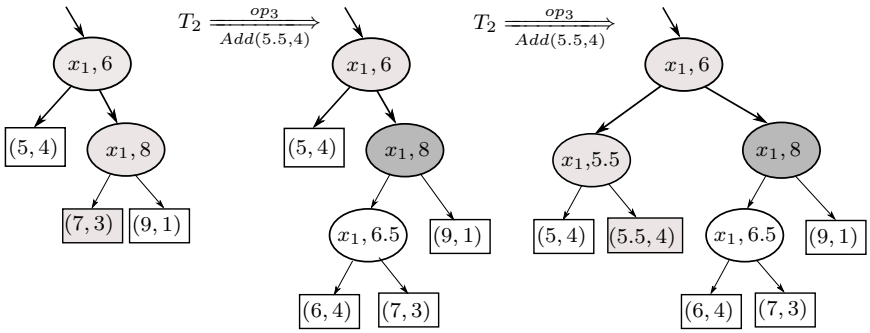

Figure 3: Illustration of modification operations concurrent with an NNS$(6, 4)$ operation. Light shaded nodes denote nodes currently on the traversal path of NNS and the dark shaded nodes denote roots of sub-trees that have been pruned.

```
94  class Nebr {▷ Neighbour
      Node* a; double d;
    }
95  class NbrClctr{▷ Neighbour-collector
      double[] tgt; bool isAct;
      Nebr* col, rep; NbrClctr* next;
96  }
97  ncp := NbrClctr*(null, false, null, null, null);

    NNS(double[] k)
98    pa := root; a := pa·lt; hi := {∞0}^d; lo := {−∞0}^d;
99    ⟨pa, a⟩ := Seek(pa, a, k, hi, lo);
100   dst := IsMark(a) ? ∞ : ||k, a·k||2;
101   if dst ≠ 0 then
102     return NNSync(pa, a, dst, k, hi, lo);
103   else {Sync(pa, a); return k;}

    Collect(Node* pa, Node* a, double[]
      k, double[] hi, double[] lo, double
      dst, NbrClctr* nn)
104   while pa ≠ Ptr(root) and dst ≠ 0 do
105     ⟨pa, a⟩ := NextGuess(pa, a, dst, k, hi, lo);
106     if ChkValid(pa, a) then
107       dst := AdNebr(a, nn, col);
108   return nn;
```

Algorithm 4. Linearizable NNS operations with single target point in LFkD-tree

the linearization point of op_1. To maintain the order, we put the linearization point of op_1 just after reading reports made by concurrent operations before returning the result of the iterative scan.

Certainly, we need to be careful about unnecessary reporting, which may potentially degrade performance. As an illustration, suppose that op_2 and op_3 both got delayed after their linearization. Now, if invocation of op happened after that, op is guaranteed to read N, if N contained the nearest neighbour of the target point. But, if in between the linearization of op_3 and invocation of op, a concurrent Remove operation removed N, op will certainly not read it, and a reporting may render the linearization point of op to be shifted to even before its invocation, which is undesired. To avoid this situation, before every reporting, we first ascertain whether the node to be reported is logically removed by calling the method IsMark().

4.2. Concurrent NNS with coinciding target point

When concurrent NNS operations have coinciding target points, they can output same result by adopting a single atomic step, which is performed during the lifetime of one of them, as the linearization point for each of them. The real-time order amongst them can be taken as the order of any predecided step, for example, their invocation steps. Thus, essentially they require a single *iterative* scan. Basically, it is similar to the linearizable snapshot algorithm of [19]. The pseudo-code of the algorithm is given in Algorithm 4.

The class **Nebr**, line 94, represents a packet of a data-point, as contained in a leaf-node pointed by the node-pointer a, and its distance, given as d, from the target point of an NNS. The class **NbrClctr**, line 95, represents a *neighbour-collector*: the platform for collecting and reporting the nearest neighbour. **NbrClctr** contains pointers to two **Nebr** instances: col points to one that contains collected data-point during iterative scan by an NNS operation and rep points to one that contains a data-point reported by a concurrent operation, in addition to the target point tgt. It also contains a boolean isAct, which if set true, implies an *active* neighbour-collector; and a neighbour-collector-pointer nxt, which is used in Algorithm 7. The LFkD-tree is augmented with a pointer ncp, line 97, initialized to point to an *inactive* neighbour-collector.

4.2.1. The coordinated iterative scan

The operation NNS, line 98–103, starts with calling the method Seek(), line 99, to perform the ini-

tial query to arrive at a leaf-node. The process of a query down the kD-tree is described in Section 4.3. If the pointer to the leaf-node a does not contain the descriptor mark, which indicates that the node pointed by a is not logically removed, and if the query key k matches at the leaf-node, which is checked by the distance between k and the key at the leaf-node, k itself is the nearest neighbour available in the dataset and NNS returns, line 103. Otherwise, NNS calls NNSync() to perform further steps and returns the nearest neighbour, line 102. The arrays hi and lo are used to support non-recursive traversal, described in Section 4.3. NNSync() and called methods thereof are described here.

The method NNSync(), line 109–120, starts with checking whether ncp points to an active neighbour-collector, and if it does not, it allocates a new active neighbour-collector and attempts a CAS to modify ncp to point to the new one, line 114. In case ncp was pointing to an active neighbour-collector, the current best guess of nearest-neighbour, as contained in the leaf-node, is attempted to be added to that. On discovering or allocating and adding an active neighbour-collector, the method Collect() is called to perform a *coordinated iterative scan*, line 119.

Collect(), line 104–107, calls the method NextGuess(), line 105, to perform next iteration that can better the current best guess of the nearest neighbour. NextGuess() performs a classical non-recursive traversal in the kD-tree. The non-recursive method improves the traversal in a concurrent setting, where the kD-tree can change its structure during the traversal. We describe NextGuess() in Section 4.3.

Before attempting to add the new guess, contained in a leaf-node, to the neighbour-collector using the method AdNebr(), it is always checked whether the leaf-node is logically removed by calling the method ChkValid(). Please note that, given a (possibly stale) pointer to a leaf-node, we can not directly check whether it was logically removed. Therefore, we also supply the pointer to the parent and thus the method ChkValid(), line 152–156, gets the latest pointer to the leaf-node considering the fact that a new internal-node may get added between the parent of the leaf-node and the leaf-node to be reported.

AdNebr(), line 126–135, is called to add a collected or reported neighbour to an active neighbour-collector. It calls the method NearNbr(), shown in line 136–139, which returns a new neighbour only if the distance of the "newly guessed neighbour" is less than the distance of the already collected or reported neighbours to the neighbour-collector.

After the iterative scan, the method Deactivate() is called by NNSync() at line 120. Deactivate(), line 141–142, other than setting the IsAct to false, also injects a descriptor finish at both the neighbour-pointers of the neighbour-collector using the method BlockNebr(). BlockNebr(), line 146–151, performs a CAS to replace a neighbour-pointer with one that has the descriptor finish over it, see lines 149 and 150. It ensures that each of the concurrent NNS operations using the same neighbour-collector have the matching view of it after linearization. The method IsFinish() returns true when called on a neighbour-pointer with descriptor finish. Thus, AdNebr() can not add a new neighbour in a neighbour-collector if the corresponding pointer is injected with finish, see line 128.

Finally, the method Process(), line 143–145, is called by NNSync() to select the better candidate between the reported and the collected neighbours of the target point, which is returned to the caller NNS to output. Note that, once a neighbour-collector is *deactivated* by an NNS, the method AdNebr() returns 0, line 135. This in turn, immediately terminates the **While** loop in Collect() at line 104. Thus, as mentioned in Section 1.2, we can observe that the *coordination* among the concurrent iterative scans at the same neighbour-collector helps a delayed NNS to complete faster.

4.3. The Non-recursive Traversal

The main tool of a non-recursive traversal for the iterative scan is to keep track of an (orthogonal) axis aligned bounding box (AABB) of the points in the subtrees, both visited and pruned. An AABB is described by its two corner points. We use the variables hi and lo throughout the algorithms to represent the two corner points. Initially, to begin the query in the operation NNS, the corner points are taken as $\{\infty_0\}^d$ and $\{-\infty_0\}^d$, see line 98 in Algorithm 4, which cover the entire dataset.

The method Seek(), line 6 to 163, which is called by NNS for the initial query at line 99 in algorithm 4, starts with the initial AABB as described by the two arrays hi and lo with their initial values, and performs a query absolutely similar to the method Search() to arrive at a leaf-node. At the termination of Seek(), the arrays AABB represent the

```
NNSync(Node* pa, Node* a, double dst,
  double[] k, double[] hi, double[] lo)
109 | while true do
110 | | on := ncp;
111 | | if on·isAct = false then
112 | | | cN := Nebr*(a, dst);
113 | | | nn := NbrClctr*(k, true, cN, cN, null);
114 | | | if CAS(ncp·ref, on, nn) then break;
115 | | else
116 | | | if ChkValid(pa, a) then
117 | | | | dst := AdNebr(a, on, col);
118 | | | nn := on; break;
119 | nn := Collect(pa, a, dst, k, hi, lo, nn);
120 | Deactivate(nn); return Process(nn);
Report(Node* a, NbrClctr* nn)
121 | AdNebr(a, nn, rep);
Sync(Node* pa, Node* a)
122 | if ncp·isAct then
123 | | ⟨d, nb⟩ := NearNbr(a, ncp);
124 | | if nb ≠ null and ChkValid(pa, a) then
125 | | | Report(a, ncp);
AdNebr(Node* a, NbrClctr* nn, bool nt)
  ▷nt (Neighbour-type): col or rep.
126 | while true do
127 | | nbr := (nt = col) ? nn·col : nn·rep;
128 | | if nn·isAct and !IsFinish(nbr) then
129 | | | ⟨dst, nb⟩ := NearNbr(a, nn);
130 | | | if nb = null then return dst;
131 | | | if nt = col then
132 | | | | res := CAS(nn·col·ref, nbr, nb);
133 | | | else res := CAS(nn·rep·ref, nbr, nb);
134 | | | if res then return dst;
135 | | else return 0;
NearNbr(Node* a, NbrClctr* nn)
136 | distTgt := ||a·k, nn·tgt||2;
137 | col := nn·col; rep := nn·rep;
138 | if distTgt<col·d and distTgt<rep·d then
139 | | return ⟨distTgt, Nebr*(a, distTgt)⟩;
140 | else return ⟨distTgt, null ⟩;
Deactivate(NbrClctr* nn)
141 | BlockNebr(nn, col);
142 | nn·isAct := false; BlockNebr(nn, rep);
Process(NbrClctr* nn)
143 | if nn·rep·d < nn·col·d then
144 | | return nn·rep·a;
145 | else return nn·col·a;
BlockNebr(NbrClctr* nn, bool nt)
  ▷nt (Neighbour-type): col or rep.
146 | nbr := (nt = col) ? nn·col : nn·rep;
147 | while !IsFinish(nbr) do
148 | | if nt = col then
149 | | | CAS(nn·col·ref, nbr, Finish(nbr));
150 | | else CAS(nn·rep·ref, nbr, Finish(nbr));
151 | | nbr := nt = col ? nn·col : nn·rep;
ChkValid(Node* pa, Node* a)
152 | k := a·k; ch := Child(pa, Dir(pa, k));
153 | while Ptr(ch)·class ≠ LNode do
154 | | ch := Ptr(Child(ch, Dir(ch, k)));
155 | if IsMark(ch) then return false;
156 | return ch = a ? true: false;
```

Algorithm 5. Linearizable NNS operations with single target point in LFkD-tree

```
Seek(Node* pa, Node* a, double[] k, double[]
  hi, double[] lo)
157 | cD := (a·k[pa·i] < pa·c) ? L : R;
158 | while Ptr(a)·lt ≠ null do
159 | | pa := Ptr(a); cD := Dir(pa, k);
160 | | a := Child(pa, cD);
161 | | if cD = L then hi[pa·i] := pa·c;
162 | | else lo[pa·i] := pa·c;
163 | return ⟨pa, a⟩;
NextGuess(Node* pa, Node* a, double
  dst, double[] k, double[] hi, double[] lo)
164 | cD := (a·k[pa·i] < pa·c) ? L : R;
165 | leafKey := a·k;
166 | while pa ≠ root do
167 | | if cD = L then ntVsted := (pa·c≥hi[pa·i]);
168 | | else ntVsted := (pa·c≤lo[pa·i]);
169 | | if |pa·c − k[pa·i]| < dst and ntVsted then
170 | | | cD := (cD = L ? R : L); a := Child(pa, cD);
171 | | | Seek(pa·ref, a·ref, cD·ref, k, hi)lo;
172 | | | leafKey := a·k;
173 | | | if (leafdst := ||k, leafKey||2) < dst then
174 | | | | if !IsMark(a) then {dst := leafdst; break;}
175 | | else
176 | | | a := pa; pa := pa·pr; cD := Dir(pa, leafKey);
177 | | | if cD = L then
178 | | | | if pa·c > hi[pa·i] then hi[pa·i] := pa·c;
179 | | | else
180 | | | | if pa·c < lo[pa·i] then lo[pa·i] := pa·c;
181 | return ⟨pa, a⟩;
```

Algorithm 6. Non-recursive traversal

bounding box that covers every data-point that can be in the sub-tree of the parent of the leaf-node, where it terminates, which has the same direction as the leaf-node with respect to its parent. We follow the convention that an array is always passed by reference and therefore any modification at any element in a method call persists even after the return of the method call. Thus, at the return of `Seek`(), if the query point did not match at the key of the leaf-node, we go to perform further iterations using the method `NextGuess`() with the current bounding box which represents the rectangular region of the Euclidean space that we have covered.

The method `NextGuess`(), line 164 to 181, performs an iteration for a better guess of the nearest neighbour given the distance of the current guess from the target point. We input the pointers to the current leaf-node and its parent along with the AABB described by its two corners. The first step is to find the direction of the current sub-tree and then decide whether the other sub-tree of the parent is visited or not, see lines 164, 167 and 168. Basically we check whether the axis-orthogonal hyperplane associated with the parent node is beyond the AABB. Having done that, we check whether the

unvisited AABB on the other side of the hyperplane should be visited by checking its distance from the target point and comparing it with the current distance as input, see line 169. Now, if we need to visit the other sub-tree, the method Seek() is called to perform the query and update AABB, line 171, else we traverse back to root. When we traverse back to root, the AABB is widened to cover both sub-tree rooted at an internal node, see lines 178 and 180.

Thus, the method Collect() repeatedly calls NextGuess() to perform an iterative scan of the LFkD-tree, see line 105 in Algorithm 4.

4.3.1. The reporting methods:

Add or Contains operations after completion, use the method Sync(), line 122–125, to synchronize with concurrent NNS operations. Sync() first checks the status of the neighbour-collector, then calls NearNbr() to create a neighbour. If the point to be reported is not better than the current best guess available, NearNbr() returns null and then Sync() returns without any change. Otherwise, it checks whether the leaf node with the point to be reported is logically removed by calling the method ChkValid(), and then calls the method Report(), which in turn calls AdNebr() to add the reported neighbour, line 121.

4.4. A general Concurrent NNS with multiple target points

To allow multiple concurrent NNS with non-coinciding target points to progress together, we need to have as many active neighbour-collectors as the number of different target points. Essentially, we need to have a dynamic list of neighbour-collectors. In this list, before adding a new neighbour-collector, an NNS must scan through it so that if there was already an active neighbour-collector with a matching target point, coordination among the concurrent iterative scans with coinciding target points can be achieved. For each of the operations in the LFkD-tree to be lock-free, we ensure the lock-freedom of this list as well. Hence, we augment the LFkD-tree with a single-word CAS based lock-free list of neighbour-collectors.

The linearization points remain as before: the concurrent NNS with coinciding target points share an atomic step during the lifetime of one of them as their linearization point with some order among themselves; other operations linearize as described earlier.

The pseudo-code of the algorithm is given in Algorithm 7, in which every method is absolutely same as that in Algorithm 4, except NNSync() and Sync(). The list is initialized with two sentinel nodes pointed by tail and head, with head·nxt set as tail, as given in lines 182 and 183. A new neighbour-collector is added to this list at one of the ends only, which is just before the node pointed by tail. The method of maintaining this list is similar to the lock-free linked-list of Harris [24], except the fact that no addition happens anywhere in the middle of the list. Removal of a neighbour-collector, say one pointed by c, takes two successful CAS steps: first we inject a mark descriptor at the c·nxt using a CAS and then modify the pointer p·nxt to n with a CAS, if p and n happened to be the pointers to the predecessor and successor, respectively, of the neighbour-collector pointed by c. We use the method Mark() to get a word-sized packet of a neighbour-collector-pointer and the descriptor mark, whereas, the method Ptr() masks the descriptor off such a packet and does not change a neighbour-collector-pointer. Adding a neighbour-collector takes a single successful CAS similar to [24].

The method NNSync(), line 184–201, as called by NNS after the initial query in Algorithm 4, starts with traversing the list. We maintain an **enum** variable mode that indicates the stages of NNSync(). Initially, the mode is INIT. During the traversal, if an active neighbour-collector with matching target point is found, the mode is changed to COLLECT and traversal terminates, line 194. Otherwise, the traversal terminates in the mode INIT itself. On the termination of the traversal in the mode INIT, it is checked whether the neighbour-collector, where traversal terminated (in this case c), is already logically removed, line 196, and if it is, a CAS is attempted to detach it from the list and the traversal is restarted, line 197.

After that, if the mode is INIT or COLLECT, the method Finalize() is called. Finalize(), line 221–229, if called in the mode INIT, allocates a new neighbour-collector by calling the method Allocate(), otherwise uses the input neighbour-collector. If Allocate() could not add a new neighbour-collector, it returns null and the entire process restarts from scratch with a fresh traversal. After successfully adding a new neighbour-collector to the list or asserting that it needs to use an existing one, Finalize() calls the methods Collect() and Deactivate() similar to those in Algorithm 4. On deactivating the neighbour-collector, the method Clean() is called to remove it from the

```
182 tail := NbrClctr*(null, false, null, null, null);
183 head := NbrClctr*(null, false, null, null, tail);

    NNSync(Node* pa, Node* a, double dst,
      double[] k, double[] hi, double[] lo)
184 | nn := null; mode := INIT;
185 | retry:
186 | while true do
187 | | p := null; c := head; n := c·nxt;
188 | | while Ptr(n) ≠ tail do
189 | | | if n = nn and mode = CLEAN then
190 | | | | if (val := Clean(c, nn)) ≠ null then
191 | | | | | return val;
192 | | | | else goto retry;
193 | | | else if k = n·tgt and n·isAct then
194 | | | | nn := n; mode := COLLECT; break;
195 | | | else {p := c; c := n; n := n·nxt;}
196 | | if mode = INIT and IsMark(n) then
197 | | | CAS(p·nxt·ref, c, Ptr(n)); goto retry;
198 | | if mode ≠ CLEAN then
199 | | | ⟨val, mode⟩ := Finalize(pa, a, dst, k,
      | | | hi, lo, p, c, mode);
200 | | | if val ≠ null then return val;
201 | | else return Process(nn);

    Allocate(Node* a, double dst, double[] k, NbrClctr*
      c)
202 | cNb := Nebr*(a, dst);
203 | nn := NbrClctr*(k, true, cNb, cNb, tail);
204 | if CAS(c·ref, on, nn) then return nn;
205 | else return null;

    Clean(NbrClctr* pre, NbrClctr* nn)
206 | nxt := nn·nxt;
207 | while !IsMark(nxt) do
208 | | CAS(nn·nxt·ref, nxt, Mark(nxt));
209 | | nxt := nn·nxt;
210 | if CAS(pre·nxt·ref, nn, Ptr(nxt)) then
211 | | return Process(nn);
212 | else return null;

    Sync(Node* pa, Node* a)
213 | n := head·nxt;
214 | while n ≠ tail do
215 | | if n·isAct then
216 | | | nb := NearNbr(a, n);
217 | | | if nb ≠ null and ChkValid(pa, a) then
218 | | | | Report(a, n);
219 | | | else break;
220 | | else n := Ptr(n·nxt);

    Finalize(Node* pa, Node* a, double dst, double[]
      k, double[] hi, double[] lo, NbrClctr* p, NbrClctr*
      c, enum md)
221 | if md = COLLECT then nn := c; pre := p;
222 | else if md = INIT then
223 | | nn := Allocate(a, dst, k, c); pre := c;
224 | | if nn ≠ null then mode := COLLECT;
225 | if md = COLLECT then
226 | | nn := Collect(pa, a, dst, k, hi, lo, nn);
227 | | Deactivate(nn); md := CLEAN;
228 | | if (val := Clean(pre, nn)) ≠ null then
229 | | | return ⟨val, md⟩;
```

Algorithm 7. Linearizable NNS with multiple distinct target points in LFkD-tree

list and return the value of the nearest neighbour.

Clean(), line 206–212, performs the two CAS steps to remove the neighbour-collector and calls the method Process(), line 211, to compute the nearest neighbour. However, if after injecting mark, it could not modify the nxt pointer of the predecessor, it returns null, which again causes a fresh traversal in the mode CLEAN in Finalize(). A traversal in mode CLEAN, if finds the deactivated neighbour-collector, calls the method Clean(), line 191, to redo the remaining steps and return the nearest neighbour. If the traversal terminates in the mode CLEAN, that implies that a concurrent NNS would have detached the deactivated neighbour-collector and therefore Process() is called to finish, line 201.

4.4.1. Approximate Nearest Neighbour Search

Practitioners prefer better query latency at the cost of exact solution in various applications that require a nearest neighbour search, which is commonly known as approximate-Nearest Neighbour (ANN) [25, 26, 11, 27]. Consider a target point $q{=}\{q_i\}_{i=1}^{d} \in \mathbb{R}^d$ of the NNS, given $\epsilon > 0$, we say that a point k^* is the $(1+\epsilon)$-ANN of q if

$$dist(k^*, q) \leq (1+\epsilon) dist(k, q),$$

where k is the *true* nearest neighbour to q.

Generally, in a hierarchical multidimensional data structure like kD-tree, ANN algorithms relax the pruning criterion so that an NNS operation visits lesser number of subsets and thereby it speeds up the performance. Implementing ANN in a concurrent hierarchical multidimensional data structure does not impact the design-complexity as long as we follow the same consistency framework.

5. Correctness and Lock-freedom

We present a detail proof of the presented algorithm. The proof structures as defining the key notions, presenting the invariants of the data structure and finally proving the correctness in terms maintenance those invariants and then satisfying the requirements of linearizability and lock-freedom.

Shared Memory System: We consider an *asynchronous shared memory system* $\mathcal{U}$ which comprises a set of word-sized *objects* $\mathcal{V}$ and a finite set of processes $\mathcal{P}$ and supports *primitives* read, write and CAS (compare-and-swap). $\mathcal{U}$ guarantees that the primitives are *atomic* i.e. they take effect instantaneously at an indivisible time-point [28]. Each object $v \in \mathcal{V}$ has a unique *address*, commonly known as a *pointer to* v, denoted by $v{\cdot}$ref.

CAS(v·ref, exp, new) compares the value of v with exp and on a match updates it to new in a single atomic step and returns true; else it returns false without any update at a. Let $|\mathcal{P}| = n$. Processes $p \in \mathcal{P}$ communicate by accessing the objects $v \in \mathcal{V}$ using a primitive. A *configuration* $\mathcal{U}_t$ of $\mathcal{U}$ specifies the value of each of $v \in \mathcal{V}$ and the state (values of local variables, etc.) of each of $p \in \mathcal{P}$ at time t. The *initial configuration* $\mathcal{U}_0$ represents the initial value of each of $v \in \mathcal{V}$ and the initial state of each of $p \in \mathcal{P}$.

Abstract Data Type: The *multidimensional* data indicates the set of points $a \in \mathbb{R}^d$ and *distance* refers to Euclidean distance $||a, b||_2 \ \forall\, a, b \in \mathbb{R}^d$. Let $F^{\mathbb{R}^d}$ be the set of all countably finite subsets of $\mathbb{R}^d$. Let $k \in \mathbb{R}^d$ and $\mathbb{A} \in F^{\mathbb{R}^d}$. Let $\mathcal{B}$:={true, false}. An *abstract data type* (ADT) kDSet is specified as a set of mappings $\mathcal{M}$ = {Add, Remove, Contains, NNS} as defined below:

Definition 1 (***ADT Operations***). *1.* $\mathit{Add} : \mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathcal{B} \times F^{\mathbb{R}^d}$ *s.t.* $\mathit{Add}(k, \mathbb{A}) = (\mathsf{true}, \mathbb{A} \cup k)$ *if* $k \notin \mathbb{A}$ **and** $\mathit{Add}(k, \mathbb{A}) = (\mathsf{false}, \mathbb{A})$ *if* $k \in \mathbb{A}$.

2. Remove:$\mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathcal{B} \times F^{\mathbb{R}^d}$ *s.t.* $\mathit{Remove}(k, \mathbb{A}) = (\mathsf{true}, \mathbb{A}/k)$ *if* $k \in \mathbb{A}$ **and** $\mathit{Remove}(k, \mathbb{A}) = (\mathsf{false}, \mathbb{A})$ *if* $k \notin \mathbb{A}$.

3. $\mathit{Contains} : \mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathcal{B} \times F^{\mathbb{R}^d}$ *s.t.* $\mathit{Contains}(k, \mathbb{A}) = (\mathsf{true}, \mathbb{A})$ *if* $k \in \mathbb{A}$ **and** $\mathit{Add}(k, \mathbb{A}) = (\mathsf{false}, \mathbb{A})$ *if* $k \notin \mathbb{A}$.

4. $\mathit{NNS} : \mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathbb{R}^d \times F^{\mathbb{R}^d}$ *s.t.* $\mathit{NNS}(k, \mathbb{A}) = (a^*, \mathbb{A})$ *where* $a^* \in \mathbb{A} \wedge ||a^*, k||_2 \leq ||a, k||_2 \ \forall a \in \mathbb{A}$.

Data Structure: A *LFkD-tree* Υ stores points from a dataset $\mathbb{A} \in F^{\mathbb{R}^d}$. The *state* of Υ in configuration $\mathcal{U}_t$, denoted Υ_t, stores points from $\mathbb{A}_t \in F^{\mathbb{R}^d}$. For an unbounded and dynamic design, Υ is constructed using *nodes* and *links* that are assembled of the objects $v \in \mathcal{V}$. In Section 2.1, we described the structure of the LFkD-tree in detail. The access of Υ is availed by root - the address of a fixed *sentinel* node. The *left (right) -subtree* of a node N, denoted by N·L (N·R), is the set of nodes comprising of the left (right) -child and all its descendants.

Operation Descriptors: An *operation descriptor* is a boolean variable. A link represented by a pointer, which occupies a single object $v \in \mathcal{V}$, is called to be *injected with a descriptor* des if a *test* for des on the link returns true. A descriptor des can be mark, flag, ltag or rtag. We call a link clean if it is not injected with any descriptor. At any time $t \geq 0$, a node N is said to be *present* in Υ_t, denoted by $\mathsf{N} \in \Upsilon_t$, if it can be *reached* following links starting from root and the link that connects its parent to itself is not injected with the descriptor mark. If Υ_t stores $\mathbb{A}_t$ then $\mathsf{N} \in \Upsilon_t \implies \mathsf{N{\cdot}ref{\cdot}ky} \in \mathbb{A}_t$.

Implementation: An *implementation* $\mathcal{I}_\mathcal{O}$ of kDSet is an algorithm, which implements mappings $\mathcal{O} \subseteq \mathcal{M}$ using *operations* on Υ. We call the implementation *full* if $\mathcal{O} = \mathcal{M}$, otherwise it is called *partial*. We assign the operations same name as its corresponding mapping. Thus, a mapping $op(k, \mathbb{A})$, where $op : \mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathcal{B} \times F^{\mathbb{R}^d}$, $k \in \mathbb{R}^d$ and $\mathbb{A} \in F^{\mathbb{R}^d}$, is implemented by an operation $op(k)$ that outputs true or false and makes appropriate changes in Υ storing $\mathbb{A}$. An NNS$(k, \mathbb{A})$ is implemented by NNS(k), which outputs a point $a^* \in \mathbb{R}^d$ according to the mapping definition.

Operation Steps: A process $p \in \mathcal{P}$ performs an operation *op* as a set of *steps*. Often we group a subset of steps in *op* as a method, which is *called* from inside of *op*. A *step* $s = \langle v, g, h, p \rangle$, where g and h are the values of the object v before and after the execution of s, comprises at most one execution of a primitive and can contain some calculations over process-local variables of p. The *execution-point* of s is the point on a real time-line where its atomic primitive takes effect. We denote the *invocation* and *response* steps of *op* by $s_i(op)$ and $s_r(op)$, respectively. The execution-points of $s_i(op)$ and $s_r(op)$, denoted by $t^i(op)$ and $t^r(op)$, are called the *invocation point* and *response point*, respectively. $\mathcal{I}_\mathcal{O}$ specifies the initial configuration $\mathcal{U}_0$.

Execution History: An *execution* α of $\mathcal{I}_\mathcal{O}$ is a (finite or infinite) sequence of steps performed by the processes $p \in \mathcal{P}$, starting from $\mathcal{U}_0$. A *history* $\mathcal{H}$ of α is its subsequence consisting of the invocation and response steps. A *subhistory* of $\mathcal{H}$ is its subsequence. A *process subhistory* of $\mathcal{H}$, denoted by $\mathcal{H}|_p$ is its subsequence containing steps executed by a $p \in \mathcal{P}$. We call histories $\mathcal{H}$ and $\mathcal{H}'$ *equivalent*, denoted $\mathcal{H} \equiv \mathcal{H}'$, if $\forall p \in \mathcal{P}$, $\mathcal{H}|_p = \mathcal{H}'|_p$. In $\mathcal{H}$, a response step of an operation *op matches* an invocation step if the two are performed by the same process. A history is called *sequential*, if the first step is an invocation and every invocation step, except possibly the last one, follows by a matching response step. We assume that every history $\mathcal{H}$ is *well-formed*: $\forall p \in \mathcal{P}$, $\mathcal{H}|_p$ is sequential. An operation in a history is effectively the *pair* of its invocation and response steps. Let op_1 and op_2 be two operations in $\mathcal{H}$. We call op_1 *precedes* op_2 in $\mathcal{H}$, denoted $op_1 \underset{\mathcal{H}}{\rightarrow} op_2$, if $t^r(op_1) < t^i(op_2)$. We call

two operations op_1 and op_2 *concurrent* in $\mathcal{H}$, if neither precede the other. $\mathcal{H}$ is called *concurrent* if it contains at least one pair of concurrent operations.

Extension and Completion of History: We call an invocation s *pending* in $\mathcal{H}$, if $\mathcal{H}$ does not contain a matching response to it. An *extension* of $\mathcal{H}$, denoted $ext(\mathcal{H})$, is obtained by appending matching response steps to every pending invocation in $\mathcal{H}$. A *completion* of $\mathcal{H}$, denoted by $complete(\mathcal{H})$, is obtained by either (a) dropping all the pending invocation steps from $\mathcal{H}$, or (b) extending $\mathcal{H}$, or (c) dropping some of the pending invocation steps and extending the remaining $\mathcal{H}$ (this case will be clear after we define the linearization point below).

Consistent Sequential History: A *sequential specification* of $\mathcal{I}_{\mathcal{O}}$ is a set of sequential histories. Let $s_i(op), s_r(op) \in \mathcal{S}$, where $\mathcal{S}$ is a sequential history. Let $\Upsilon_{t^i(op)}$ and $\Upsilon_{t^r(op)}$ be the states of Υ at $t^i(op)$ and $t^r(op)$, which store the datasets $A_{t^i(op)}$ and $A_{t^r(op)}$, respectively. We call the operation *op consistent* with respect to the ADT kDSet in $\mathcal{S}$ if the output arguments at the response and $A_{t^r(op)}$ satisfy the corresponding mapping definition of kD-Set. The sequential history $\mathcal{S}$ is *consistent* if each operation in it is consistent.

Definition 2 (***Linearizability:***)**.** *A history $\mathcal{H}$ is linearizable if $\exists \mathcal{H}_e = ext(\mathcal{H})$ and a consistent sequential history $\mathcal{S}$ s.t. (a) $complete(\mathcal{H}_e) \equiv \mathcal{S}$ and (b) $op_1 \underset{\mathcal{H}_e}{\longrightarrow} op_2 \implies op_1 \underset{\mathcal{S}}{\rightarrow} op_2$. We call an implementation $\mathcal{I}_{\mathcal{O}}$ linearizable if every execution history of $\mathcal{I}_{\mathcal{O}}$ is linearizable.*

The most common approach to prove linearizability is: (a) define *linearization point* of each operation *op* as the execution-point of a step, called *linearization*, which should be between the invocation and response point of *op* then (b) in an arbitrary history $\mathcal{H}$ append appropriate response (in any arbitrary order) of all the operations which have performed their linearization, then (c) drop the invocation steps without a matching response to obtain $complete(\mathcal{H})$, and (d) construct a sequential history $\mathcal{S}$ by arranging the invocation-response pair of operations according to their linearization points. It is easy to argue that $complete(\mathcal{H}) \equiv \mathcal{S}$. And, finally, show that the constructed sequential history $\mathcal{S}$ is consistent.

Below we list out the linearization points of the operations as the following:

Definition 3 (***Linearization points***)**.** *1. For a successful* `Add` *operation, it is at line 9 or line 11 in the method* `ChCAS`()*, which is called at line 35 in the method* `AddNode`() *and which in turn was called by* `Add`*.*

2. *For a successful* `Remove` *operation, it is at line 9 or line 11 in the method* `ChCAS`()*, which is called at line 46 in* `Remove`*.*
3. *For an unsuccessful* `Add` *and a successful* `Contains` *operation it is at line 14 in the method* `Search`() *called from these operations.*
4. *For an unsuccessful* `Contains` *and* `Remove` *operation, it can be either just after the linearization point of a concurrent* `Remove` *operation or at the invocation point of these operations.*
5. *For an* `NNS` *operation, if it returns a data-point which was contained in a collected-neighbour, the linearization point is at line 159 in algorithm 6 in the method* `Seek`() *called from the* `NNS`*.*
6. *For an* `NNS` *operation, if it returns a data-point which was contained in a reported-neighbour, the linearization point is just after the linearization point of either* `Contains` *or* `Add` *that reported the neighbour.*

It is easy to observe in algorithm 2 that these linearization points are in between $t^i(op)$ and $t^r(op)$ for respective $op \in \mathcal{O} = \{\texttt{Add}, \texttt{Remove}, \texttt{Contains}, \texttt{NNS}\}$.

Now with that, given any concurrent execution history $\mathcal{H}$ of an implementation $\mathcal{I}_{\mathcal{O}}$, where $\mathcal{O} \subseteq \{\texttt{Add}, \texttt{Remove}, \texttt{Contains}, \texttt{NNS}\}$, we form an equivalent sequential history $\mathcal{S}$ by following the steps as described above. And thus it remains to be shown that such a sequential history will be consistent.

To do that, we essentially show that the invariants of the LFkD-tree, as stated in Section 2.1 are maintained, and the sequential specifications as described in Section 2.2 are satisfied by the consistent operations. Because the implementation of the lock-free list of neighbour-collectors is orthogonal to the implementation of the LFkD-tree, we also need to show that the invariants of the list, as stated in section 4.4, are maintained by the `NNS` operations. Therefore, here we state the invariants and present some observations and lemmas which help us to show that the invariants are maintained.

Given a LFkD-tree Υ, for every internal-node $\texttt{N}(i, c)$ and a leaf node $\texttt{N}(\{k_i\}_{i=1}^{d})$, Υ maintains the following invariants:

Invariant 1. *A node $\texttt{N}(\{k_i\}_{i=1}^{d})$ belongs to the left subtree, if $k_i < c$.*

Invariant 2. *A node* $\mathtt{N}(\{k_i\}_{i=1}^{d})$ *belongs to the right subtree, if* $k_i \geq c$.

Invariant 3. *A node* $\mathtt{N}(\{k_i\}_{i=1}^{d})$ *belongs to the right subtree, if* $k_i \geq c$.

A LFkD-tree state Υ_t that satisfies the Invariants 1 to 3 is called a *valid state*. Now, for the list of the neighbour-collectors, we denote a neighbour-collector by $\mathtt{NC}(\{k_i\}_{i=1}^{d})$ if the target point that it contains is $\{k_i\}_{i=1}^{d}$. A neighbour-collector list maintains following invariant:

Invariant 4. *In the list there can not be two neighbour-collectors* $\mathtt{NC}(\{k_i\}_{i=1}^{d})$ *and* $\mathtt{NC}(\{j_i\}_{i=1}^{d})$ *such that* $k_i = j_i \ \ \forall\ i\ :\ 1 \leq i \leq d$.

To prove that the above invariants are maintained throughout the algorithms, we provide the following observations and lemmas.

Observation 1. *The fields* k *and* i *are never changed in a* **Node**.

Observation 2. *Any link in a LFkD-tree is updated only using a* CAS.

Observation 3. *The sentinel nodes are never removed.*

Observation 4. *The* pr *pointer of the node* root *is never dereferenced.*

Going through the pseudo-code we can observe that once we allocate a node, we never call any store step on the fields k and i and any pointer update is done using a CAS. The choice of keys in the sentinel nodes verifies the third observation. The pr pointer of an internal node is dereferenced only if a Remove operation on any of its children is called. Thus Observation 3, implies Observation 4.

Lemma 1. *In each call of* Dir(), *line 6, variable* $\mathtt{N}(i,c)\cdot\mathsf{ref}$ *represents a pointer which is* clean *and points to an internal-node and thus is not* null.

Lemma 2. *In each call of* Child(), *line 7, pa is* clean *and points to an internal-node and thus is not* null.

Lemma 3. *In each call of* ChCAS(), *line 8 to 12, pa is* clean *and points to an internal-node, whereas new is* clean *and points to a leaf-node; thus pa and a are both not* null.

Lemma 4. *In each call of* Search(), *line 13, pa is* clean *and points to an internal-node, whereas a is* clean *and points to a node (internal or leaf); thus both are not* null.

Lemma 5. *In each call of* Search(), *line 13, pa and a satisfy* $a = pa\cdot\mathsf{lt} \mid pa\cdot\mathsf{rt}$.

Lemma 6. *In each call of* HelpMrk(), *line 62, pa is* clean *and points to an internal-node, whereas a is* clean *and points to a leaf-node; thus both are not* null.

Lemma 7. *In each call of* HelpFlg(), *line 89, ga and pa are* clean *and point to two different internal-nodes, whereas sa is eitherpoints to a leaf-node and thus are not* null.

Lemma 8. *In each call of* HelpTag(), *line 73, ga is* clean *and points to an internal-nodes, whereas pl is either* ltag *or* rtag *and points to an internal-node and thus are not* null.

Lemma 9. *In each call of* ApndTag(), *line 75, pa and a are* clean. *pa points to an internal-nodes, whereas a points to a leaf-node and thus both are not* null.

Lemma 10. *In each call of* ApndFlg(), *line 66, pa is* clean *and points to an internal-node and thus is not* null.

Lemma 11. *A pointer once injected with a descriptor* mark, flag, ltag *or* rtag *is not injected with any descriptor ever after.*

Lemmas 1 to 10 provide a base to prove that at no point an implementation of the presented algorithm faces a segmentation fault due to the dereferencing of a null pointer during the operations Add, Remove and Contains. To prove these lemmas we inspect the pseudo-code in Algorithms 2 and 3. At each call of the utility methods we find that the inputs to the utility methods follow the requirements of these lemmas. A listing of the lines of the pseudo-code containing call of these methods verifies this claim. The statements of this set of lemmas is what we need to prove the next set of lemmas which provides the verified base for postconditions of the LFkD-tree operations.

Lemma 12. *At the termination of* Search *at line 15,*

(a) pa *points to an internal-node and is* clean.
(b) a *points to a leaf-node and can be either* clean *or* mark *or* flag.
(c) pa *and* a *satisfy* $a = pa\cdot\mathsf{lt} \mid pa\cdot\mathsf{rt}$.
(d) $a\cdot\mathsf{k}[pa\cdot\mathsf{i}] \geq pa\cdot\mathsf{c} \implies a = pa\cdot\mathsf{rt}$.
(e) $a\cdot\mathsf{k}[pa\cdot\mathsf{i}] < pa\cdot\mathsf{c} \implies a = pa\cdot\mathsf{lt}$.

Following from Lemmas 4 and 5, the **while** loop ensures that the variable a always points to one of the child-pointers of the node pointed by pa; this validates Lemma 12 (a), (b) and (c).

Now, Lemma 11 provides that if the `CAS` steps are orderly in a `Remove` operation, it does not result in a malformation of the LFkD-tree. Also, for an `Add` operation, because the single `CAS` that it requires can not happen over a link with descriptor, an incorrect addition is avoided.

Now, the keys in the sentinel nodes vacuously prove the following Lemma 13, which provides base condition for an induction to prove Theorem 1.

Lemma 13. *Initially, the LFkD-tree consisting of the sentinel nodes satisfies the invariants as stated in section 2.1.*

Now we are prepared to prove Theorem 1. We use induction to prove it. Using Lemma 13, when no update has happened, the nodes in the LFkD-tree satisfy the invariants. It is straightforward to observe that no `Contains` or `NNS` operation involves a write (`CAS`) step and therefore they do not change the state of the LFkD-tree. From Lemma 12, at the end of every call to `Search`, which satisfies the symmetric order of the LFkD-tree, a `CAS` to `Add` does not violate the Invariants 1 to 3. For a `Remove` operation, after the `CAS` to logically removing the node i.e. `mark CAS`, the order of `CAS` do not let any update operation let the node reappear in the LFkD-tree following Lemma 11.

Thus if the state of the LFkD-tree was consistent before the application of an update operation, it remains so after its linearization. Using induction, Theorem 1 follows.

Theorem 1. *At any time $t \geq 0$ the LFkD-tree state Υ_t is a valid state.*

Now considering the neighbour-collector-list, its semantics are absolutely same as those of Harris's lock-free linked list [24] and which was further improved my Micheal [29]. A very sophisticated proof of the state change and thus validity of the list algorithm was provided by Micheal [29]. The invariant maintained our list, Ivariant 4, can be proved along the same lines and we skip the detail here. Now, we prove the linearizability of the implementation $\mathcal{I}_{\mathcal{M}}$ as given below.

Theorem 2. *(Correctness) The operations `Add`, `Remove`, `Contains` and `NNS` are linearizable with respect to the abstract data type kDSet.*

Proof. We show that a sequential history $\mathcal{S}$ obtained by following the steps: (a) in an arbitrary history $\mathcal{H}$ append appropriate responses (in any arbitrary order) of all the operations which have performed their linearization steps as defined in definition 3, (b) drop the invocation steps without a matching response to obtain *complete*($\mathcal{H}$), and (c) construct $\mathcal{S}$ by arranging the invocation-response pair of operations according to their linearization points, is consistent.

Let $\mathcal{S}_n$ be a sub-history of $\mathcal{S}$ that contains the *first* n complete operations. Let $\mathbb{A}_n$ be the dataset which was added to the LFkD-tree by the successful `Add` operations in $\mathcal{S}_n$. Let $\mathbb{B}_n$ be the dataset which was removed from the LFkD-tree by the successful `Remove` operations in $\mathcal{S}_n$. Let $\mathbb{C}_n = \mathbb{A}_n / \mathbb{B}_n$. We use (strong) induction on n to show that $\mathcal{S}_n$ is consistent $\forall\ n \geq 1$.

Suppose that $\mathcal{S}_n$ is consistent $\forall\ n\ :\ 1 \leq n \leq i$. Let the $(i+1)^{th}$ operation in $\mathcal{S}_n$ be $op(k)$, where $k \in \mathbb{R}^d$. Then for $\mathcal{S}_{i+1}$ we prove the following:

1. Let $op(k)$ be an `Add` operation.
   (a) Let $op(k)$ returns `true`. We show that if $op_1(k)$ is an `Add` operation such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $op_1(k)$ returns `true` then $\exists$ a `Remove` operation $op_2(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $op_2(k)$ returns `true`.
   Suppose there does not exist such a `Remove` operation. Now, following Lemma 12, at the termination of `Search`(), line 14 in algorithm 2, $pa \rightsquigarrow a$ is a leaf-node pointer. Now using the construction of $\mathcal{S}_i$ and definition 3-(1), at the linearization of op, it performed a successful `CAS` at the link $pa \rightsquigarrow a$ which must have been `clean`. Using the same argument op_1 also performed a successful `CAS` at the link $pa \rightsquigarrow a$ which must have been `clean`. Now because op_1 linearized before op, the set of nodes that the `Search`() called from op, terminates at, by the consistency of $\mathcal{S}_i$ op must find k being the key at that leaf-node. Now unless the link $pa \rightsquigarrow a$ was already injected with the descriptor `mark`, op would not have continued beyond the termination of `Search`() and reading the descriptor at it and thereby returning `false`. Therefore, there must have been a `Remove` operation which marked the link $pa \rightsquigarrow a$ before op read and thus it had the linearization point before that of op. This is a contradiction.
   (b) Let $op(k)$ returns `false`. We show that $\exists$ an `Add` operation $op_1(k)$, which returns `true`, such

that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $\nexists$ a `Remove` operation $op_2(k)$, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

Suppose the contrary. Then at the termination of `Search`(), line 14 in algorithm 2, by definition 3-(3) the link $pa \rightsquigarrow a$ is `clean` and $a.\mathsf{k} = k$. But, following (a) as above and the consistency of $\mathcal{S}_i$, there must exist an $op_1(k)$ in $\mathcal{S}_i$ which returns `true` and that does not precede an $op_2(k)$ which returns `true`- which contradicts our assumption.

Now, it is easy to see that after the linearization of an `Add` operation that returns `true`, the node added by it is reachable from `root` following the links and thus that node belongs to the LFkD-tree which in turn implies that $k \in \mathbb{C}_{i+1}$. Thus, combining this fact with (a) and (b) together, the mapping definition of `Add`, definition 1-(1), is satisfied. Thus, `Add` is consistent in $\mathcal{S}_{i+1}$.

2. Let $op(k)$ be a `Remove` operation.
   (a) Let $op(k)$ returns `true`. We show that if $op_1(k)$ is a `Remove` operation, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\exists$ an `Add` operation $op_2(k)$, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$. We use similar argument as given in (1) to prove it.
   (b) Let $op(k)$ returns `false`. We show that one of the following is `true`:
   i. If $op_1(k)$ is a `Remove` operation, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\nexists$ an `Add` operation $op_2(k)$, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   Suppose the contrary is true. Then, because $op_1(k)$ return `true`, by the construction of $\mathcal{S}_{i+1}$ and the definition of the linearization point definition 3-(2), either a leaf-node does not exist with key k or the link to it is injected with `mark`. Now if that is the case and op also returns `true`, then there must have been a link to a leaf-node with key k which was `clean`. But that was possible only if an `Add` existed before op, which added a leaf-node with key k. This contradicts our claim.
   ii. There $\nexists$ an `Add` operation $op_1(k)$, which returns `true`, and $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   We can observe that at the linearization of $op(k)$, the link to the leaf-node with key k gets injected with `mark` and thus after that $k \notin \mathbb{C}_n$. Combining this fact with (a) and (b) satisfies the sequential specification of `Remove`- definition 1-(2). Thus, `Remove` is consistent in $\mathcal{S}_{i+1}$.
3. Let $op(k)$ be a `Contains` operation.
   (a) Let $op(k)$ returns `true`. We show that $\exists$ an `Add` operation $op_1(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $\nexists$ a `Remove` operation $op_2(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   The arguments are similar to (1)(b) above.
   (b) Let $op(k)$ returns `false`. We show that one of the following is `true`:
   i. If $op_1(k)$ is a `Remove` operation, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\nexists$ an `Add` operation $op_2(k)$, which returns `true`, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   ii. There $\nexists$ an `Add` operation $op_1(k)$, which returns `true`, and $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   The arguments are similar to (2)(b) above. Combining (3)(a) and (3)(b), `Contains` is consistent in $\mathcal{S}_{i+1}$.
4. Let $op(k)$ be an `NNS` operation that returns k^*. We show that (a) there $\exists$ $op_1(k^*)$ such that $op_1(k^*) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and (b) if there $\exists$ $op_1(k^{**})$, which returns `true`, where op_1 is either `Add` or `Contains` and $||k^{**},\ k||_2 < ||k^*,\ k||_2$ such that $op_1(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then there $\exists$ a `Remove` operation $op_2(k^{**})$, which returns `true`, such that $op_1(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   To prove (a), it is easy to see that if such an `Add` did not exist preceding op then at the linearization of op it can not read a leaf-node containing k^*. Therefore, (a) is `true`.
   Now, for (b), suppose the contrary is `true`. Thus, if there did not exist a `Remove` operation op_2 then at the linearization of op, which is either at the termination of the method `Seek`() called by itself or at the termination of the method `Search`() called by reporting `Contains` or at the `CAS` step performed by a reporting `Add` operation, the leaf-node containing k^{**} must have been connected by a `clean` link. But then either op would have read the `clean` link to the leaf-node with k^{**} or the operation reporting to it would have done the same. Thus the method `Process`() that is called by `NNS` before its return, by virtue of $||k^{**},\ k||_2 < ||k^*,\ k||_2$, would have returned k^{**} which in turn would have been returned as the nearest neighbour of k by op. Which is a contradiction. Thus, `NNS` is consistent in $\mathcal{S}_{i+1}$.

By (1) to (4), $\mathcal{S}_{i+1}$ is consistent whenever $\mathcal{S}_n$ is consistent $\forall n : 1 \leq n \leq i$. Thus, using induction, $\mathcal{S}_n$ is consistent for every positive integer n. □

Theorem 3. *(Lock-freedom) The LFkD-tree operations* `Add`*,* `Remove`*,* `Contains` *and* `NNS` *are lock-free and thus the presented algorithm implements a lock-free LFkD-tree.*

Proof. We take the `NNS` operation separately because it also involves the steps related to the lock-free list. By the description of the algorithm, a non-faulty thread performing a `Contains` will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of `Add` operations would have successfully completed adding new nodes making the implementation lock-free. So, in the context of `Add`, `Remove` and `Contains`, it will suffice to prove that the modify operations are lock-free.

Suppose that a process $p \in \mathcal{P}$ performs a modify operation *op* on a valid state of LFkD-tree Υ_t and takes infinite steps and no other modify operation completes after that. Now, if no modify operation completes then Υ_t remains unchanged forcing p to retract every time it wants to execute its own modification step on Υ_t. This is possible only if every time p finds the injection point of *op* with descriptor `mark`, `flag`, `ltag` or `rtag`. This implies that a `Remove` operation is pending. It is trivial to observe in the method `Add` that if it gets obstructed by a concurrent `Remove`, then before retrying after recovery from failure, it helps the pending `Remove` by executing all the remaining steps of that. We can also observe that whenever two `Remove` operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so Υ_t changes. It is contrary to our assumption. Hence, by contradiction we show that no non-faulty process shall remain taking infinite steps if no other non-faulty process makes progress where the executed operation is either `Add` or `Remove`.

Now we consider an `NNS` with concurrent `Add`, `Remove` or `Contains` operations. We consider the case where concurrent `NNS` operations do not necessarily have coinciding target points; this case obviously covers the case when they do have coinciding target points. We can see that a `Remove` operation does not have to report to a concurrent `NNS` operation. Moreover, an `Add` or a `Contains` operation to perform a reporting, needs to first traverse through the unordered list and then possibly perform a `CAS` if required to report. Now, unless the number of `NNS` operations keep on increasing infinitely, the total length of the unordered list will be finite and thus the traversal path for an `Add` or a `Contains` operation to report will be finite. Now, at each neighbour-collector, where the reporting is required, if a `CAS` to report fails, that implies that a concurrent `Contains` or `Add` operation succeeds. Similarly, when a `CAS` by a `NNS` operation fails, it indicates that a `CAS` by a concurrent `NNS` operation succeeded. Finally, a `CAS` to add a new neighbour-collector only indicates that either a new neighbour-collector by a concurrent `NNS` has been successfully added or a `NNS` operation has terminated. In case of a `CAS` failure to add a new neighbour-collector, a `NNS` operation always helps a concurrent pending `NNS` operation before reattempting, in case it finds the link with descriptor `mark`. It shows that in all cases at least one non-faulty thread succeeds with respect to execute a `NNS` operation concurrent to any other LFkD-tree operation. Thus we arrive at Theorem 3. □

This concludes the proof of the presented algorithm.

6. Experimental Evaluation

6.1. Experiment Setup

We implemented the LFkD-tree algorithm in Java using run-time type identification (RTTI). We used the library objects `AtomicReferenceFieldUpdater` to perform `CAS`. The test environment comprised a dual-socket server with a 2.0GHz Intel (R) Xeon (R) E5-2650 with 8 physical cores each (32 hardware threads in total with hyper-threading enabled). The server has 64 GB of RAM, runs Ubuntu 13.04 Linux (Kernel version: 3.8.0-35-generic x86_64) with Java HotSpot (TM) 64-Bit Server VM (build 25.60-b23), and we compiled all the implementations with `javac` version 1.8.0_60.

Data Structures: For experimental evaluation, in addition to our designs, we considered two other implementations that support `NNS`. The implementations in the evaluation are:

1 Levy-Kd: An implementation of kD-tree of [30] by Levy [31] that supports `Remove` operations (we could not find any other Java implementation of a kD-tree with `Remove`). To allow for concurrent access, we augmented the implementation with

coarse-grained `ReadWrite`[3] lock.

2 LFKD: Our implementation of the LFkD-tree with `NNS`.

3 A-LFKD: Our implementation of the LFkD-tree with *approximate*-`NNS` ($\epsilon = 2$).

4 PH-tree: A multi-dimensional storage and indexing data structure by Zäschke *et al.* [15] that supports `Remove` operations. Similar to *Levy-Kd*, we add coarse-grained `ReadWrite` lock to allow for concurrency.

Workload and Methodology: We run each test for 5 seconds and measured throughput as the total number of operations per microsecond executed by all threads in this time duration. We run each experiment in a separate instance of the JVM, starting off with a 2-second "warm-up" period to allow the Java HotSpot compiler to initialize and optimize the running code. During this warm-up phase, we performed random Add, Remove and Contains operations, and then flushed the tree. At the start of each execution, the data structure is pre-filled with keys in the selected key-range.

To simulate the variation in contention and tree structure, we chose following combination of workload configurations: i) dataset space dimension $\in \{2, 3, 4, 5\}$, ii) distribution of (`Add-Remove-NNS`) $\in \{(05, 05, 90), (25, 25, 50), (40, 40, 20)\}$, and iii) number of threads $\in \{1, 2, 4, 8, 16, 32\}$.

We did not include `Contains` operations in experiment because essentially it would increase the proportion of exact-match `NNS`. All executions use the same set of randomly generated points for the selected workload characteristics. The graphs present average of throughput over 6 runs of each experiment.

6.2. Datasets

We performed evaluation using a 2D real-world dataset and a set of synthetic benchmarks. For the real-world dataset, we used the United States Census Bureau 2010 TIGER/Line KML [32] dataset that consists of polylines describing map features of the United States of America. TIGER/Line is a standard dataset used for benchmarking spatial databases [15]. For this evaluation, we extracted points representing the mainland, resulting in $18.4 * 10^6$ unique 2-d points, with x-y coordinates that lie between -124.85 $\leqslant x \leqslant$ -66.89 and 24.40 $\leqslant y \leqslant$ 49.38.

[3]A *ReadWriteLock* consists of a pair of locks: *read* lock may be held by multiple readers as long as the write lock is free, *write* lock is exclusive)

To investigate more variable workloads, two synthetic datasets were utilized. The SKEWED data simulates datasets in which different dimensions may have varying distributions. The SKEWED(α) dataset contains uniformly distributed points which fall within 0.0 and 1.0 in every dimension that have been skewed in the y-dimension. For each point in the dataset, the y value is replaced with the value y^α, for example in the 2-dimension case, each point (x, y) is replaced with (x, y^α). In the fig. 5(a), we show examples for SKEWED(1) which is intuitively uniform distribution in all dimensions. SKEWED(3) and SKEWED(6) are shown in the fig. 5(b) and fig. 5(c), respectively .

The CLUSTER dataset [15] is an extension of a synthetic dataset previously described by Arge *et al.* [33]. In this evaluation we used clusters of 1000 points evenly spaced on a horizontal line. Each of the clusters is filled with evenly distributed points and stretches 0.00001 in every dimension. Figure 5(d) depicts an example of the CLUSTER dataset with 49 points per cluster. The line of clusters falls within (0.0, 1.0) along the x-axis and is parallel to every other dimensional axis with a 0.5 offset.

6.3. Observations and Discussion

The Figures 4, 6 and 7 show the performance of the implementations for TIGER/Line, SKEWED and CLUSTER datasets respectively. In Figure 6 and 7, each row represents a combination of the range of the number of unique keys (k=N, N being the maximum) and the associated workload distribution while each column the dimensionality of key (d=dimension).

In all of experiments, LFKD and A-LFKD have better throughput performance (in million operations per second) compared to both the PH-tree and the Levy-Kd, even in single thread cases, for all workload distributions. The performance significantly scales up with increasing thread count. This shows that our implementation is both lightweight and scalable. As we increase the key dimension, the performance degrades for workloads dominated by the `NNS`. This degradation with increasing key dimensions is expected in kD-trees due to the *curse of dimensionality* [1]. This performance pattern is identical for different key ranges. However, the LFKD still achieve speed-up over the single threaded implementations.

We further observe that, as expected, A-LFKD outperforms LFKD in `NNS` dominated workload, the

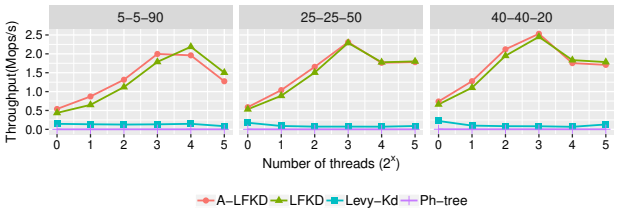

Figure 4: Performance on the 2-D TIGER/Line dataset.

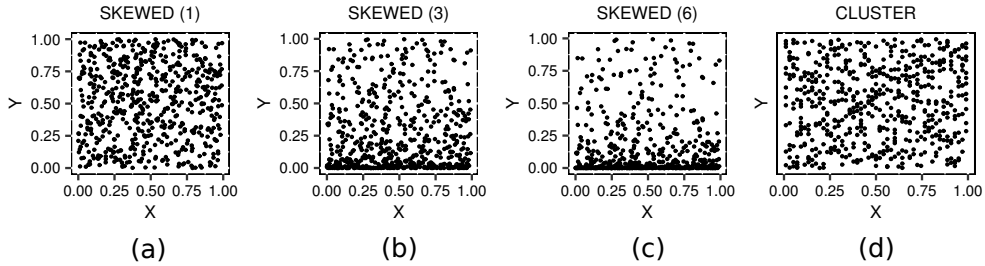

Figure 5: Synthetic dataset.

performance benefit increases with increasing dimensionality of the data set that brings the increased load of iterative scan. This can be explained by early termination of the iterative scan in the A-`NNS` in A-LFKD which prunes parts of the tree with are otherwise traversed by the `NNS` in LFKD.

For the TIGER/Line dataset, in a single thread case, both LFKD and LFKD(SC) perform at least 2.5× better than Levy-Kd, and, it goes up to 19× in the `NNS` dominated workload. Additionally, the PH-tree outperforms the Levy-Kd only for workloads that do not involve `NNS` (00% `NNS`, 50% `Add` and 50% `Remove`).

We observe that for `NNS` dominated workload (90% `NNS`, 5% `Add` and 5% `Remove`), the A-LFKD achieves speed-ups up to 66× for SKEWED and up to 150× for CLUSTER datasets over the sequential implementations. These observations can be partially attributed to the local-midpoint rule, which carries the essence of the sliding-midpoint-splitting rule of [10] that targets the extreme cases such as a CLUSTER dataset, to a concurrent setting.

For a mixed workload (50% `NNS`, 25% `Add` and 25% `Remove`), the performance of LFkD-tree degrades by increasing key dimension. The absolute throughput figures are higher for the `NNS` dominated workload in lower dimensions than in mixed workloads. This is because the modify operations incur higher synchronization (conflicts, expensive atomic operations, and helping) overhead. However in higher dimensions, the throughput of the `NNS` is lower as the number of visited nodes increases tremendously with dimension.

7. Conclusion and Future Work

For a large number of applications, which require a multidimensional data structure supporting dynamic modifications along with nearest neighbour search, research community has largely focused on improving the design of sequential data structures. Parallel implementations of the sequential designs speed up loading of and NNS on a fully loaded data structure. Thus, they do not address the issue of dynamic modifications in the datasets. On the other hand, the concurrent data structure research is primarily confined to one-dimensional problems.

Our work is the first to extend the concurrent data structures to problems covering multidimensional datasets. We introduced LFkD-tree, a lock-free design of kD-tree, which supports linearizable nearest neighbour search operations with concurrent dynamic addition and removal of data. We provided a sample implementation which shows that the LFkD-tree algorithm is highly scalable.

Our method to implement linearizable nearest neighbour search is generic and can be adapted to other multidimensional data structures. We plan to design lock-free data structures which are suitable for nearest neighbour search in high dimensions, for example, the ball-tree [34]. We also plan to extend our work to k-nearest neighbour (kNN) search.

References

[1] H. Samet, Foundations of multidimensional and metric data structures, Morgan Kaufmann, 2006.

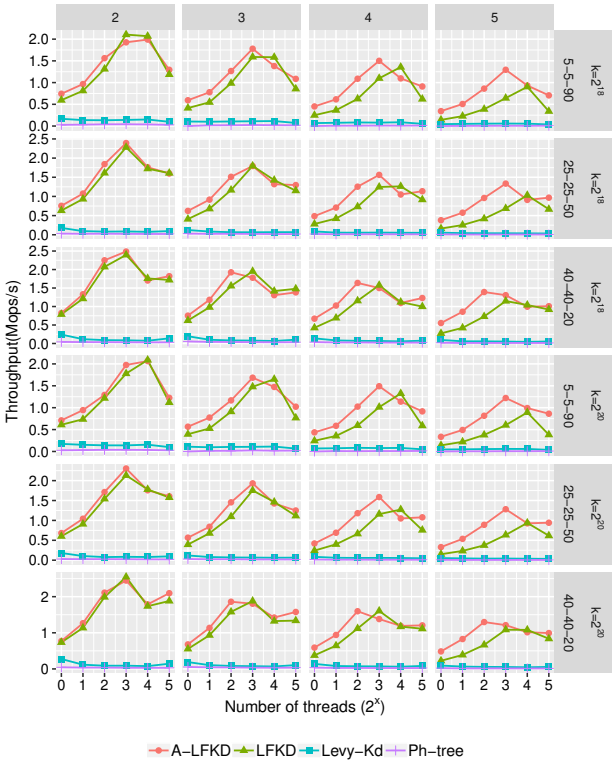

Figure 6: Performance on the SKEWED(6) dataset.

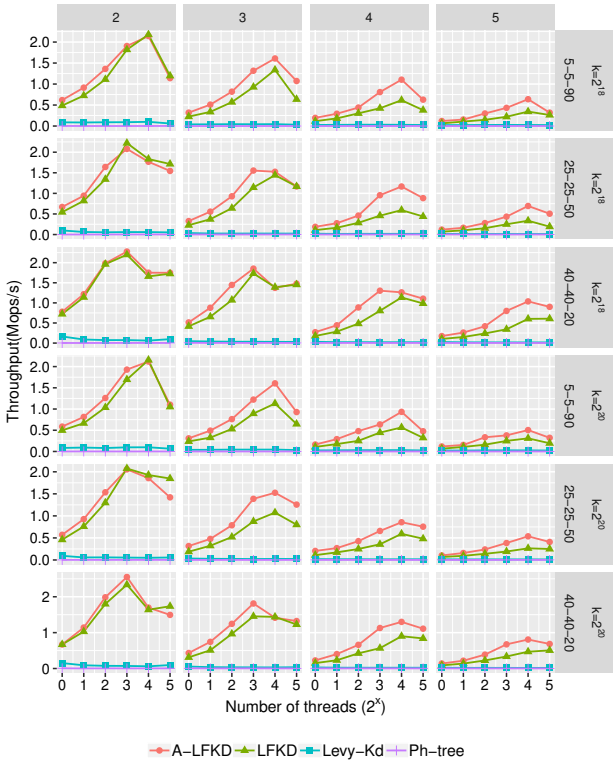

Figure 7: Performance on the CLUSTER dataset.

[2] M. P. Herlihy, J. M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems (TOPLAS) 12 (3) (1990) 463–492.

[3] H. Sundell, P. Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, J. Parallel Distrib. Comput. 65 (5) (2005) 609–627.

[4] F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel, Non-blocking binary search trees, in: 29th PODC, 2010, pp. 131–140.

[5] S. V. Howley, J. Jones, A non-blocking internal binary search tree, in: 24th SPAA, 2012, pp. 161–171.

[6] A. Natarajan, N. Mittal, Fast concurrent lock-free binary search trees, in: 19th PPoPP, 2014, pp. 317–328.

[7] B. Chatterjee, N. Nguyen, P. Tsigas, Efficient lock-free binary search trees, in: 33rd PODC, 2014, pp. 322–331.

[8] J. L. Bentley, Multidimensional binary search trees used for associative searching, CACM 18 (9) (1975) 509–517.

[9] J. H. Friedman, J. L. Bentley, R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, Transactions on Mathematical Software (TOMS) 3 (3) (1977) 209–226.

[10] D. M. Mount, S. Arya, Ann: a library for approximate nearest neighbor searching, `http://www.cs.umd.edu/~mount/ANN/`.

[11] S. Arya, H.-Y. A. Fu, Expected-case complexity of approximate nearest neighbor searching, SIAM Journal on Computing 32 (3) (2003) 793–815.

[12] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time kd-tree construction on graphics hardware, ACM Transactions on Graphics (TOG) 27 (5) (2008) 126.

[13] J. Ichnowski, R. Alterovitz, Scalable multicore motion planning using lock-free concurrency, Robotics, IEEE Transactions on 30 (5) (2014) 1123–1136.

[14] B. Chatterjee, `CKDTree`, in: `https://tinyurl.com/hl3oktt`.

[15] T. Zäschke, C. Zimmerli, M. C. Norrie, The ph-tree: A space-efficient storage structure and multi-dimensional index, in: SIGMOD 2014, 2014, pp. 397–408.

[16] N. G. Bronson, J. Casper, H. Chafi, K. Olukotun, A practical concurrent binary search tree, SIGPLAN Not. 45 (5) (2010) 257–268.

[17] A. Prokopec, N. G. Bronson, P. Bagwell, M. Odersky, Concurrent tries with efficient non-blocking snapshots, in: Acm Sigplan Notices, Vol. 47, ACM, 2012, pp. 151–160.

[18] T. Brown, H. Avni, Range queries in non-blocking k-ary search trees, ArXiv abs/1712.05101.

[19] E. Petrank, S. Timnat, Lock-free data-structure iterators, in: Distributed Computing, Springer, 2013, pp. 224–238.

[20] B. Chatterjee, Lock-free linearizable 1-dimensional range queries, in: 18th ICDCN, ACM, 2017, pp. 9–18.

[21] K. Winblad, Faster concurrent range queries with contention adapting search trees using immutable data, ArXiv abs/1709.00722.

[22] A. M. Archibald, `KDTree`, in: `scipy.spatial.KDTree`.

[23] D. Detlefs, P. Martin, M. Moir, G. Steele, Lock-free reference counting, Distributed Computing 15 (2002) 255–271.

[24] T. L. Harris, A pragmatic implementation of non-blocking linked-lists, in: DISC, Vol. 1, Springer, 2001, pp. 300–314.
[25] M. Bern, Approximate closest-point queries in high dimensions, Information Processing Letters 45 (2) (1993) 95–99.
[26] S. Arya, D. M. Mount, Approximate nearest neighbor queries in fixed dimensions, in: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 271–280.
[27] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu, An optimal algorithm for approximate nearest neighbor searching fixed dimensions, Journal of the ACM 45 (6) (1998) 891–923.
[28] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (1) (1991) 124–149.
[29] M. M. Michael, High performance dynamic lock-free hash tables and list-based sets, in: SPAA 2002, 2002, pp. 73–82.
[30] A. W. Moore, Efficient memory-based learning for robot control, Tech. Rep. 209, University of Cambridge (1991).
[31] S. D. Levy, `KDTree`, in: `edu.wlu.cs.levy.CG.KDTree`.
[32] `https://www.census.gov/geo/maps-data/data/tiger.html`.
[33] L. Arge, M. D. Berg, H. Haverkort, K. Yi, The priority r-tree: A practically efficient and worst-case optimal r-tree, ACM Trans. Algorithms 4 (1) (2008) 9:1–9:30.
[34] T. Liu, A. W. Moore, A. Gray, New algorithms for efficient high-dimensional nonparametric classification, Journal of Machine Learning Research 7 (Jun) (2006) 1135–1158.