Concurrent Nearest-Neighbor Searching for Parallel Sampling-based Motion Planning in SO(3), SE(3), and Euclidean Spaces

Jeffrey Ichnowski and Ron Alterovitz

University of North Carolina at Chapel Hill, NC, 27599 USA,
{jeffi,ron}@cs.unc.edu

Abstract. This paper presents a fast exact nearest neighbor searching data structure and method that is designed to operate under highly-concurrent parallel operation on modern multi-core processors. Based on a kd-tree, the proposed method is fast, supports metric spaces common to robot motion planning, and supports nearest, k-nearest, and radius-based queries. But unlike traditional approaches using kd-trees, our approach supports simultaneous queries and insertions under concurrency, supports wait-free queries, and provides asymptotically diminishing expected wait-times for random concurrent inserts. We provide proofs of correctness under concurrency, and we demonstrate the proposed method's performance in a parallelized asymptotically-optimal sampling-based motion planner.

Keywords: nearest neighbors, concurrent data structure, sampling-based motion planning

1 Introduction

Nearest neighbor searching data structures are a fundamental building block for many algorithms in robotics. Algorithms such as sampling-based robot motion planners [7], typically need to repeatedly search and insert data into a nearest neighbor data structure, and thus their performance benefits from nearest neighbor operations that are fast. However, with the trend of modern CPUs towards increasing computational parallelism in the form of multiple processor cores, it is no longer sufficient for a data structure to just enable operations to be fast. To harness the full computational power of a multi-core processor, algorithms must also allow for concurrent operations across multiple cores without slowdown. Slowdown is unfortunately worsened by increasing parallelism when the data structure requires concurrent operations to *wait* for mutually exclusive access to data to ensure correct operation. A *concurrent* data structure, on the other hand, avoids this source of slowdown, by minimizing or eliminating the requirement for mutual exclusion and the associated wait time. In this paper we propose a concurrent data structure, and associated algorithms, for fast exact nearest neighbor searching that is geared towards robotics applications.

The proposed nearest neighbor data structure is based on a kd-tree [5], and thus it provides for fast insert and search operations on metric spaces important to many robotics applications—including Minkowski spaces (a common example being Euclidean), SO(3) [11], and Cartesian products thereof [20]. In the proposed data structure, as with kd-trees, branching nodes partition space into spatially separated sub-trees. Fast insertion of a new point (e.g., a robot configuration for a sampling-based motion planner) into the data structure comes from the ability to quickly traverse the partitions to an insertion point. Fast searches for a set of nearest neighbors to a query point come from the ability to use the partitions to confine traversal to a spatially relevant portion of the tree. With minor modifications to the searching algorithm, searches can also produce nearest neighbor sets that are bounded in cardinality or bounded to be within a radius from the query point.

The data structure we propose supports provably correct concurrent operations. This is in contrast to the traditional approach to kd-trees, in which concurrent operation without mutual exclusion leads to data structure corruption. Corruption occurs when concurrent operations interleave mutations that invalidate the computations of each other. The problem is only exacerbated by modern compilers and CPUs as they often automatically and unpredictably change the order of memory accesses to improve the performance of non-concurrent operations. For example, if one operation writes to memory location 'A' and then to 'B', a concurrent operation may see the change to 'B' before it sees the change to 'A'. While the reordered memory writes do not affect the correctness of the operation in which they occur, they may become problematic for the correctness of an operation running concurrently. An effective way to prevent corruption caused by interleaved mutations and reordering of memory writes, is to only allow one insert operation to happen at any moment in time by using a mutual exclusion locking mechanism. But, by definition, locking prevents concurrent operations, and thus all but one attempted concurrent insert operation will spend time waiting. When an algorithm spends time waiting instead of computing, it effectively slows down. To avoid this slowdown, the data structure we propose supports concurrent wait-free queries, and it also supports inserts that wait with asymptotic probability of zero.

In this paper we improve upon the performance of the concurrent nearest neighbor data structure that we introduced in prior work [10]. In that work, a lock-free kd-tree provided wait-free queries and lock-free inserts. The fast lock-free inserts of that approach reduced the likelihood of insert waiting, but come at the expense of increased search times due to imbalances in the resulting tree. In this paper, insert operations produce a more balanced tree resulting in faster queries, and we provide proofs of correct operation and the low probability of waits.

We embed the proposed method in a parallelized sampling-based motion planning algorithm to demonstrate its performance and ability to operate under concurrency on a 32-core computer. The improvements proposed in this paper double nearest-neighbor search performance when compared to our prior lock-free nearest neighbor search data structure, and lead to up to 30% faster convergence rates of the motion planner. Sampling-based motion planners parallelize well [1], but as the results show, contention over exclusive access to a non-concurrent nearest neighbor data structures can slow them down significantly. The concurrent data structure we propose allows the parallelized motion planner to find solutions and converge faster by generating orders of magnitude more samples when compared to locked data structures.

2 Related Work

Our proposed nearest neighbor searching approach loosely follows that of a kd-tree [5, 9, 19]. A kd-tree is a space-partitioning binary tree that splits branching nodes along axis-aligned hyperplanes in $\mathbb{R}^n$. When splitting hyperplanes occur at the median of values in the subtrees, it creates perfectly balanced trees. However, as originally proposed, kd-trees are limited to $\mathbb{R}^n$ with a Minkowski metric.

Yershova et.al. [20] extended the metric spaces supported by kd-trees to include SO(2), SO(3), and cartesian products thereof and with $\mathbb{R}^n$. The SO(3) partitions of this approach are along axis-aligned hyperplanes in $\mathbb{R}^n$. Our prior work [11] proposes partitioning SO(3) using hyperplanes that wrap around the 3-sphere manifold obtained from a quaternion representation of SO(3) rotations. While the data structure we propose in this paper works with either SO(3) partitioning scheme, we expand upon the latter to address special handling required when inserting values under concurrency.

Generalized nearest-neighbor approaches, such as the Geometric Near-neighbor Access Tree (GNAT) [6] only require a well-behaved metric and thus support a broader set of topologies than kd-trees. The generalized nature of such structures does not take advantage of knowlege of the underlying topology as kd-trees do, and thus may not be as efficient as kd-trees. Additional work is also required to make such structures support concurrent and wait-free operations.

Approximate nearest neighbor searching approaches gain search efficiency by sacrificing accuracy. Methods include locality sensitive hashing (LSH) [2] and randomized kd-trees [18]. Our focus is on exact nearest neighbor searching as the proofs of many sampling-based motion planners' asymptotic feasibility (e.g., RRT [16]) and asymptotic optimality (e.g., RRT* [13]) implicitly rely on the nearest neighbor structure being exact. However, if the trade-off of accuracy for speed is appropriate, methods such as those proposed by Arya et al. [3] and Beis et al. [4] shorten the kd-tree search process producing approximate results faster. We believe similar methods could be readily applied to our proposed method to allow for approximate nearest neighbor searching under concurrency.

Concurrent data structures, such as the binary tree proposed by Kung and Lehman [15], allow correct operation while avoiding contention by having threads lock only the node that they are manipulating. In prior work [10], we proposed a kd-tree that allows concurrent modification and searching while avoiding contention through the use of a *lock-free* atomic update. When inserting into this kd-tree, the algorithm makes partitioning choices at the leaf of the kd-tree based upon the bounds of the region and/or the value in the leaf. Empirically this approach works well for the random insertion order of the associated sampling-based planner. However, better search performance is possible with a balanced kd-tree as would be created by median splits. To better approximate median splits in this work, we incorporate the approach described by Sherlock et al. [17] that accumulates a predetermined number of values into leaves before performing a median split on the values within the leaf.

3 Problem Definition

The problem definition is stated in two key parts: (1) correct concurrent operation, and (2) nearest neighbor searching.

Correct Concurrent Operation requires that memory *writes* of one operation must not adversely affect the memory *reads* or *writes* of a concurrent operation, while minimizing the time concurrent operations wait on each other. Once an operation running on a CPU core inserts a point into the data structure, the inserted point will eventually be reachable to all other cores. Once an operation running on a CPU core reaches a point in the data structure, all subsequent operations on that core must continue to reach the point.

Nearest Neighbor Searching finds all the nearest neighbors of a query point. Let $\mathcal{C}$ be a topological space which is composed of the Cartesian product of one or more sub-topologies in $\mathbb{R}^n$ and SO(3). Let $\mathbf{q} \in \mathcal{C}$ be a single configuration in the topological space with components from each sub-topology, e.g., $\mathbf{q} = \{\mathbf{p}_i, \ldots, \mathbf{r}_j, \ldots\}$, with $\mathbf{p}_i \in \mathbb{R}^{n_i}$ and $\mathbf{r}_j \in S^3$ for each i and j. Each SO(3) component is specified using the coefficients of a unit quaternion representing a rotation in 3D space [14].

Let $d(\mathbf{q}_1, \mathbf{q}_2)$ be the distance between two configurations, such that it is the weighted sum of the distances of each sub-topology's component:

$$d(\mathbf{q}_a, \mathbf{q}_b) = \sum_i \alpha_i d^p_{\mathbb{R}^n}(\mathbf{p}_{a_i}, \mathbf{p}_{b_i}) + \sum_j \alpha_j d_{\mathrm{SO}(3)}(\mathbf{r}_{a_j}, \mathbf{r}_{b_j}),$$

where α_i and α_j are positive real weight values, $d^p_{\mathbb{R}^n}(\cdot,\cdot)$ is an L^p distance metric on $\mathbb{R}^n$, and $d_{\mathrm{SO}(3)}(\cdot,\cdot)$ is the length of the shorter of the two angles subtended along the great arc. Thus:

$$d^p_{\mathbb{R}^n}(\mathbf{p}_a, \mathbf{p}_b) = \Big(\sum_i^n |\mathbf{p}_{a,i} - \mathbf{p}_{b,i}|^p\Big)^{1/p}$$
$$d_{\mathrm{SO}(3)}(\mathbf{r}_a, \mathbf{r}_b) = \cos^{-1}|\mathbf{r}_a \cdot \mathbf{r}_b|.$$

If appropriate to the application, a similar effect to weighting the distance metric can also be obtained by scaling the $\mathbb{R}^n$ coefficients instead.

Given a set $\mathbf{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n\}$ where $\mathbf{q}_i \in \mathcal{C}$, and a query point $\mathbf{q}_{\mathrm{search}} \in \mathcal{C}$ for some topological space $\mathcal{C}$, the objective of *k-nearest neighbors search*, is to find the set $\mathbf{N} \subseteq \mathbf{Q}$, such that $|\mathbf{N}| = \min(k, |\mathbf{Q}|)$, and:

$$\max_{\mathbf{q}_i \in \mathbf{N}} d(\mathbf{q}_i, \mathbf{q}_{\mathrm{search}}) \leq \min_{\mathbf{q}_j \in \mathbf{Q} \setminus \mathbf{N}} d(\mathbf{q}_j, \mathbf{q}_{\mathrm{search}}),$$

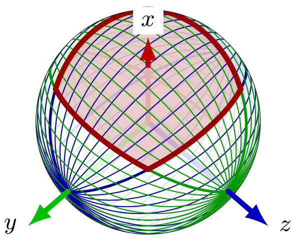


Fig. 1. Lower-dimensional analog of SO(3) partitioning scheme [11]. In SO(3), quaternions are partitioned into 4 non-overlapping bounded regions of a 3-sphere, with the negative axis mapped onto the positive axis due to the double-coverage property. The 2-sphere analog shown here is partitioned into 3 bounded region, with the x-centered bounded region highlighted in red. Within the bounded region, evenly separated partitioning hyperplanes are shown in green for one axis and in blue for the other.

where k is a positive integer. With $k = 1$ it thus finds the nearest neighbor.

The objective of *r-nearest neighbors search*, where r is a non-negative real value, is to find $\mathbf{N} \subseteq \mathbf{Q}$, such that:

$$\mathbf{N} = \{\mathbf{q}_i \mid d\left(\mathbf{q}_i \in \mathbf{Q}, \mathbf{q}_{\text{search}}\right) \leq r\}\,.$$

4 Method

The proposed method is based upon a kd-tree. A kd-tree is a binary tree data structure in which each branch successively partitions space by an axis-aligned hyperplane, and the leaf nodes contain the points to search. Searching a kd-tree for a query point begins at the root of the tree. When the search encounters a branch, it recurses to the child on the same side of the branch's splitting hyperplane as the query point. When the search encounters a leaf, it checks the distance between the leaf's point and the query point, and adds the point to the result set if the distance is small enough. When returning from recursion, the search then checks the distance between the query point and the closest point on the splitting hyperplane. If the distance between the points is small enough to be added to the result set, then the algorithm recurses to search the other child of the branch.

The partitioning approach for SO(3) [11], requires special handling for the top-level SO(3) branch (see lower dimensional analog in Fig. 1). Unlike other branches, this branch partitions space into 4 top-level *volumes*, one for each of the four components of a quaternion. (See `SO3Root` in Fig. 2). Once the algorithm has partitioned a value to a top-level SO(3) volume, the branches in the subtree are binary splits—similar to branches in $\mathbb{R}^n$, but with a hyperplane through the origin and defined by a constrained normal (see [11])

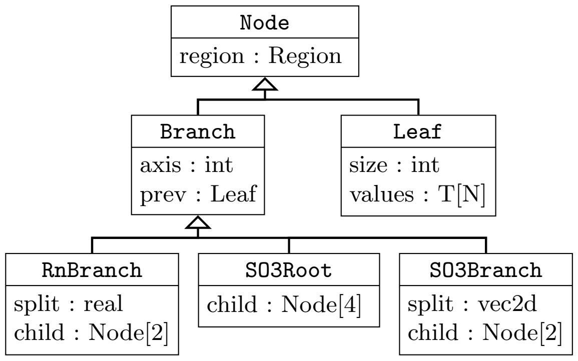


Fig. 2. Diagram of a possible node design needed to implement the proposed data structure. Each box represents a *type* of node that can be in the tree, with its name (top) and its data members (below the separating line). Data members are listed as *name : type*. Array types have their capacity listed in square brackets. Nodes inherit all members from their ancestors (shown with open arrows), thus all node types have a *region* data member. The three node types that inherit from `Branch` include a split *axis* and *prev* pointer to the Leaf node that the branch replaced. The root of an SO(3) subtree has four children, while the other branch types have a split plane definition and two children. The `Leaf` node has a current *size*, and fixed capacity (N) array of *values* of the type (T) stored in the data structure.

4.1 Data Storage

In previous work [10], we proposed a lock-free kd-tree that created a new branch every time a leaf was inserted. That approach has the benefit of making insertions quick and lock-free, but introduces an expense to search performance from two factors: (1) there is little information from which to choose a splitting hyperplane, leading to suboptimal tree-balancing, and (2) traversing a branch is more time consuming than a simple point-to-point distance check of a leaf. This performance issue is further exacerbated in algorithms that search more frequently than they insert (e.g., sampling-based motion planning algorithms such as [16, 13] that reject samples after checking the validity of paths to nearest neighbors). In the approach proposed herein, we address these two factors to improve search performance, by batching many points into leaves before splitting them into branches [17]. In our implementation, the leaf node's batch size is a fixed tunable number of the data structure.

4.2 Inserting Data

Inserting a value into a concurrent batched kd-tree (Alg. 1) starts at the kd-tree's root node (line 1) and traverses down the tree until it finds a leaf into which it will insert the new point. At each level of the tree, the current node is checked to see if it is a branch or a leaf. Empty trees and children are stored as leaf nodes with 0 size, and thus do not require special handling. When the

Algorithm 1 `INSERT`$(T, \mathbf{u})$

```
Require: $T$ is the kd-tree, $\mathbf{u}$ is the value to insert
 1: $p \leftarrow$ root of $T$
 2: loop
 3:     $n \leftarrow$ load($p$)
 4:     if $n$ is a branch then
 5:         update $n$'s region to contain $\mathbf{u}$
 6:         $p \leftarrow$ FOLLOW($n, \mathbf{u}$)
 7:     else if not try_lock($n$) then /* $n$ is a leaf */
 8:         relax CPU
 9:     else /* acquired lock on $n$ */
10:         $m \leftarrow$ load($n$.size)
11:         if $m <$ leaf capacity then
12:             update $n$'s region to contain $\mathbf{u}$
13:             append $\mathbf{u}$ to leaf $n$
14:             store($n$.size, $m + 1$)
15:             unlock($n$)
16:             return
17:         $c \leftarrow$ SPLIT($n, \mathbf{u}$)
18:         store($p, c$)
```

Algorithm 2 `FOLLOW`$(n, \mathbf{u})$

```
Require: $n$ is branch
1: if $n$ is SO(3) root then
2:     $i \leftarrow$ so3_volume_index($\mathbf{u}$)
3:     return $n$.child[$i$]
4: else if $n$ is SO(3) branch then
5:     return $n$.child[$H(\mathbf{u}[\text{axis}] \cdot n.\text{split})$]
6: else if $n$ is $\mathbb{R}^n$ branch then
7:     return $n$.child[$H(\mathbf{u}[\text{axis}] - n.\text{split})$]
```

algorithm encounters a branch node (line 4), it updates the branch node's region and traverses to the child under which the new value will be inserted. When the algorithm encounters a leaf it first attempts to lock the leaf (line 7) using a fast spin locking mechanism such as compare-and-swap (CAS) on a boolean flag. If the algorithm fails to lock the node, it issues an optional CPU-specific instruction (line 8) for efficient spin locking, and then it loops to try again. Once the algorithm successfully acquires the lock, it appends the value to the leaf if there is room (line 11), or splits the leaf (line 17) otherwise. When appending to a leaf, the algorithm ensures the new value is fully initialized before updating the leaf's size (line 14). The size update is a linearization point for making the inserted value reachable to other cores. When splitting the leaf, the algorithm replaces the leaf with the new branch (line 18), and then loops to insert the value into one of the branch's children.

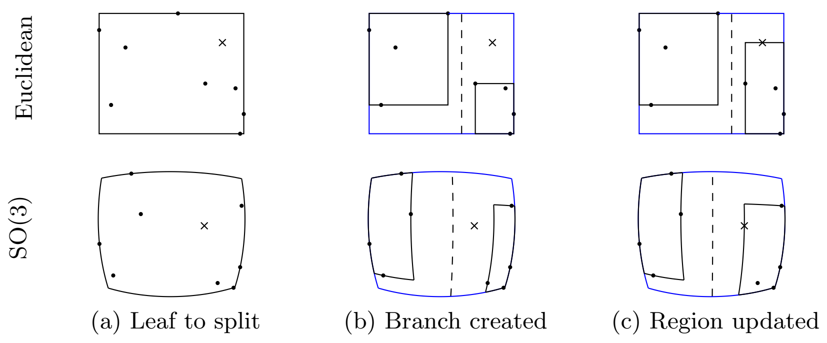


Fig. 3. Steps of splitting a leaf while operating under concurrency. In (a) the INSERT algorithm traversed to the leaf to add the 'x', finds the leaf is full, and thus calls SPLIT to create a branch. SPLIT partitions the branch along leaf's horizontal dimension resulting in the branch shown in (b). After SPLIT returns, INSERT then traverses to the right side, adds to the leaf, and updates the leafs region (c). During the SPLIT process, concurrent nearest neighbor searches traverse the old leaf. Once the INSERT replaces the leaf with the branch, searches will traverse the branch instead.

Algorithm 3 `SPLIT`$(n, \mathbf{u})$

```
Require: $n$ is a Leaf, $\mathbf{u}$ is the value to insert
 1: axis $\leftarrow$ best_axis($n$'s region)
 2: if axis is first SO(3) then
 3:     $c \leftarrow$ new SO3Root
 4:     for all $\mathbf{v} \in n$.values do
 5:         $i \leftarrow$ so3_volume_index($\mathbf{v}$)
 6:         append $\mathbf{v}$ to $c$.child[$i$]
 7:     return $r$
 8: else
 9:     $b_0, b_1 \leftarrow$ median_split($p$, axis)
10:     split $\leftarrow \frac{1}{2}$(max($b_0$.values) + min($b_1$.values))
11:     return new branch with axis, split, $b_0$, $b_1$
```

Traversing to Insertion Point FOLLOW (Alg. 2) implements the branch traversal required by the INSERT algorithm. When it encounters an SO(3) root node, it traverses to the child whose partition contains the sample. When it encounters an SO(3) branch or an $\mathbb{R}^n$ branch, it computes the signed distance between the point to insert and the splitting hyperplane. The sign of the distance selects the child using a Heaviside step function $H(\cdot)$ defined as:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0. \end{cases}$$

Splitting Leaf Nodes When inserting into a full leaf, the INSERT algorithm uses `SPLIT` (Alg. 3) to create a branch from the values in the full leaf. For an efficient kd-tree the splitting process will choose a partition that: (1) minimizes the maximum distance between points in the resulting subdivision and (2) divides the values into equal leaf nodes with the same number of elements (median split). To that end, the `SPLIT` algorithm first selects the best axis for partitioning as the one with the greatest extent between region bounds of the leaf. The region bounds are maintained by INSERT. For $\mathbb{R}^n$ axes, the extent is the difference between the minimum and maximum along each dimension of the bounds. For SO(3) root nodes, the extent is $\pi/2$. For SO(3) branch nodes, the extent is the arccosine of the dot product of the minimum and maximum normalized bounds for the axis [11].

If the selected axis is the SO(3) root, the SPLIT algorithm creates a new SO3Root branch node, and copies the old leaf's values into the appropriate child of the SO3Root (lines 3 to 6). Otherwise, for the remaining axis types, `median_split` (line 9) partitions the values of the old leaf evenly into two new leaf nodes (see Fig. 3 (a) and (b)) using an efficient selection algorithm. The SPLIT algorithm returns with a new branch that is split halfway between the maximum of one child and the minimum of the other (line 11).

In the presence of concurrency, concurrent nearest neighbor searches will continue to traverse the old leaf until INSERT atomically replaces the old leaf with the new branch. This means that INSERT does not know if the old leaf is being concurrently accessed, and thus cannot release the memory associated with the leaf without risking a program error. The SPLIT algorithm presented here stores a reference to the old leaf to allow the memory associated with the leaf to be safely deallocated later.

4.3 Searching Operations

NEAREST (Alg. 4) implements k-nearest neighbor (with k as appropriate and $r = \infty$) and r-nearest neighbor (with $k = \infty$ and r as appropriate) searches. Traversal for searching for a nearest neighbor is similar to that of FOLLOW. The primary difference is that after searching one child of the branch, NEAREST may need to search the other children of a branch. The algorithms starts with a pointer n to the root node of the kd-tree, and an empty set N of nearest neighbors. It terminates recursion if the node's region (as maintained by INSERT) is too far away from the query point to be added to the nearest neighbor set. If the node is a leaf (lines 3 to 7), it iterates through each value in the leaf, updating the N as appropriate. Here it first loads the node's size, ensuring that it will only visit consistent values in the leaf based upon the linearization point in INSERT.

When traversing an SO3Root node, NEAREST navigates the search key's SO(3) axis-major volume first (lines 9 and 10). It then searches the remaining volumes in an arbitrary order (lines 11 to 12).

When traversing an SO3Branch node or RnBranch node (lines 15 to 19), the algorithm first traverses a child in the same order as FOLLOW does. After returning from recursion on that child, it then traverses the other child. By recursing on the closer child first, updates to N will cause the traversal on the farther child to terminate quickly on line 1.

Algorithm 4 `NEAREST`$(N, n, \mathbf{q}, k, r)$

```
Require: $N$ is the set of nearest neighbor result so far, $n$ is a pointer to the current
    node, $\mathbf{q}$ is the query, $k$ is the maximum $|N|$ to return, $r$ is the maximum radius
1: if $|N| < k$ or dist$(n.\text{region}, \mathbf{q}) \leq \min(r, \max N)$ then
2:     if $n$ is leaf then
3:         for all $i \in \{0, \dots$ load$(n.\text{size})\}$ do
4:             if $|N| < k$ or dist$(n.\text{values}[i], \mathbf{q}) < \min(r, \max N)$ then
5:                 if $|N| = k$ then
6:                     $N \leftarrow N \setminus (\max N)$
7:                 $N \leftarrow N \cup n.\text{values}[i]$
8:     else if n is SO(3) root then
9:         $i \leftarrow$ so3_volume_index$(\mathbf{q})$
10:        $N \leftarrow$ NEAREST$(N,$ load$(n.\text{child}[i]), \mathbf{q}, k, r)$
11:        for all $v \in \{0, 1, 2, 3\} \setminus i$ do
12:            $N \leftarrow$ NEAREST$(N,$ load$(n.\text{child}[v]), \mathbf{q}, k, r)$
13:    else
14:        if $n$ is SO(3) branch then
15:            $c \leftarrow H(\mathbf{q} \cdot n.\text{split})$
16:        else /* $n$ is $\mathbb{R}^n$ branch */
17:            $c \leftarrow H(\mathbf{q}[\text{axis}] - n.\text{split})$
18:        $N \leftarrow$ NEAREST$(N,$ load$(n.\text{child}[c]), \mathbf{q}, k, r)$
19:        $N \leftarrow$ NEAREST$(N,$ load$(n.\text{child}[1-c]), \mathbf{q}, k, r)$
20: return $N$
```

5 Correctness and Analysis

In this section we prove that NEAREST is wait-free and correct with concurrent INSERTS (lemma 2), and provide analysis on the probability that INSERT waits (lemma 4). Correct operation relies upon *linearizable* operations which appear to occur instantaneously at a *linearization point* from the perspective of concurrent operations. Thus, before the linearization point, the linearizable operation has not occurred, and after the linearization point, the operation has occurred—there is no intermediate point in which the operation partially occurs. We prove that INSERT is *linearizable* (lemma 1) and that once a value is reachable it remains reachable (lemma 3). The following proofs depend upon *release* and *acquire* ordering semantics where noted in the algorithms. These semantics ensure that all memory writes that happen before the release-ordered store (via `store`$(a, \cdot)$) become visible side-effects of an acquire-ordered load (via `load`(a)). Implementations must explicitly ensure this ordering.

Lemma 1. *The INSERT operation is linearizable.*

Proof. INSERT can modify a leaf in one of two ways: (1) by appending a value to a leaf, or (2) splitting the leaf into a branch. As such, there are two linearization point cases to make INSERT linearizable.

Case (1): INSERTs do not store new values until they have exclusive write access to a leaf, and thus no two INSERT operations will concurrently store a value into the same leaf. INSERT stores the new value one past the leaf's size limit before incrementing the size with a release-order store. Concurrent operations do not read values in a leaf past the leaf's size limit, thus storing the incremented size is the linearization point for this case.

Case (2): INSERT splits a leaf by replacing it with a new branch node with children populated from the values from the leaf. As INSERT locks the leaf before populating the branch's children, the same values will be present in both leaf and branch. INSERT replaces the pointer to the leaf with the pointer to the branch using a release-order store. Since concurrent operations will either load a pointer to the leaf before the store, or to the branch after the store, the store is the linearization point for this case.

Both cases have linearization points, and thus INSERT is linearizable. □

In case (2), unlike case (1), the leaf is not (necessarily) unlocked, as concurrent INSERT operations waiting for the leaf will load the new branch after the linearization point, and recurse to operate on a child of the new branch.

Lemma 2. *The NEAREST operation is wait-free, and concurrent INSERT operations do not cause incorrect operation.*

Proof. The NEAREST operation contains no blocking waits or retry loops, and thus will not wait on other operations. Correct operation under concurrency results from the two linearization points of NEAREST.

In case (1), when NEAREST visits a leaf, it first performs an acquire-order load of the leaf's size before iterating through the values in the leaf. As incrementing the size is the linearization point, NEAREST will only iterate through values in the leaf stored before the linearization point, and thus it will only traverse consistent data.

In case (2), when NEAREST recurses to search a child of a branch, it performs an acquire-order load of a pointer to the child. NEAREST will either load the pointer before or after the corresponding linearization point of INSERT. If NEAREST loads the child before the linearization point, it will recurse to visit the leaf. If NEAREST loads the child after the linearization point, it will recurse to visit the branch. All nodes remain valid once reached, including leaf nodes that have been replaced by branch nodes, thus NEAREST will operate correctly under concurrency. □

Lemma 3. *Once a value is reachable by NEAREST, the value will remain reachable to all subsequent NEAREST operations.*

Proof. A value is first reachable after linearization point case (1) of INSERT. The leaf in which the value resides remains reachable until linearization point case (2) of INSERT. After the linearization point case (2), all values from the original leaf reside in the child nodes of the branch that replaced the original leaf. The originally reachable value thus remains reachable before and after linearization point case (2), and the value will thus *always* remain reachable to subsequent NEAREST operations. □

Lemma 4. *With uniform random insertion, an INSERT operation waits, or causes a wait, with probability* $(1 - ((n-1)/n)^{p-1})$*, where* p *is the number of concurrent INSERT operations and* n *is the number of leaf nodes in the tree. INSERT asymptotically almost surely does not wait.*

Proof. An INSERT will loop, and thus effectively wait on line 7, if a concurrent INSERT had a successful `try_lock` on the *same* leaf. Leaf nodes represent a bounded subregion of the space with uniform distribution. We cast this as the generalized birthday problem, and follow its derivation. Let $P(A_n)$ be the probability that an INSERT concurrently updates a leaf of the same bounded subregion as any of the other $(p-1)$ concurrent INSERTs. This is equivalent to $1 - P(A'_n)$, where $P(A'_n)$ is the probability that no other INSERT concurrently updates the same bounded region. We compute $P(A'_n)$ as the joint probability that $(p-1)$ INSERT operations are updating different regions. Thus,

$$P(A_n) = 1 - P(A'_n) = 1 - \left(\frac{n-1}{n}\right)^{p-1}.$$

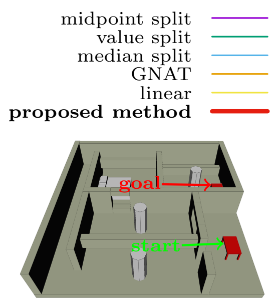
It follows that $\lim_{n\to\infty} P(A'_n) = 1$, and thus INSERT asymptotically almost surely does *not* wait. □

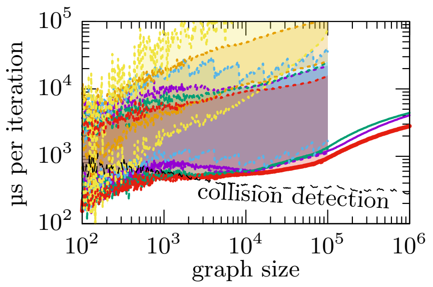
6 Results

We evaluate the proposed data structure by embedding it in PRRT* [10], a lock-free parallelized asymptotically optimal sampling-based motion planner. The data structure and planner implementations use the standard C++ atomic library [12] for memory operations that require release and acquire semantics. PRRT* uses the proposed data structure for concurrent insert, nearest, and k-nearest operations. We have PRRT* compute motion plans in two SE(3) rigid-body scenarios from OMPL [8] on a computer with four Intel x7550 2.0-GHz 8-core Nehalem-EX processors, using all 32 cores.

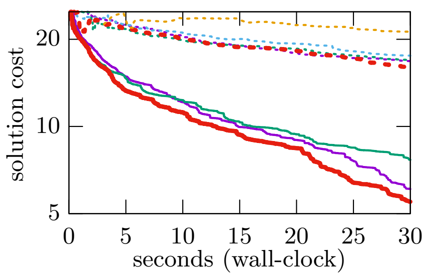
The experiments compare both concurrent and locked nearest neighbor data structures to show the benefit of using data structures designed for concurrency. Locking on the data structure makes use of an efficient reader/writer lock, under the observation that insertions are relatively fast and infrequent compared to time spent nearest neighbor searching. Thus the locked version of the data structure is exclusively write-locked when inserting, and shared read-locked when searching. This prevents searches from traversing an inconsistent data structure that would result from partial mutations and reordered memory writes of a concurrent insert. It also allows multiple concurrent searches that only block when there is a concurrent insert.

We compare our proposed method to the linear (brute-force) implementation included in OMPL, the GNAT implementation included in OMPL, the dynamically rebalanced median-split kd-tree from prior work [11], and the original lock-free kd-trees in PRRT*. The OMPL methods and the median-split kd-tree method are read/write locked. The concurrent methods are also evaluated in

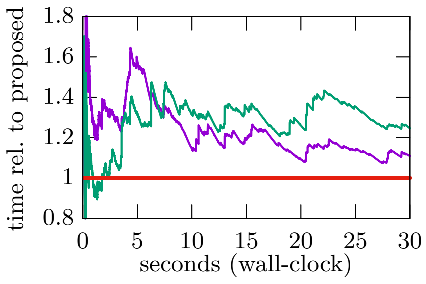




(a) Nearest neighbor time

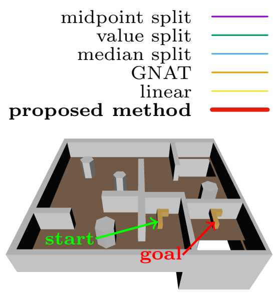


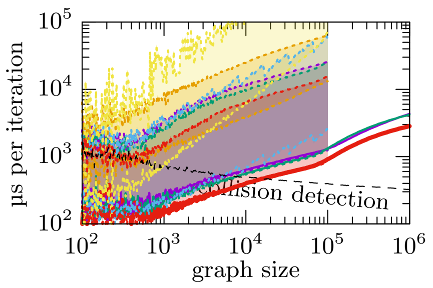
(b) PRRT* convergence



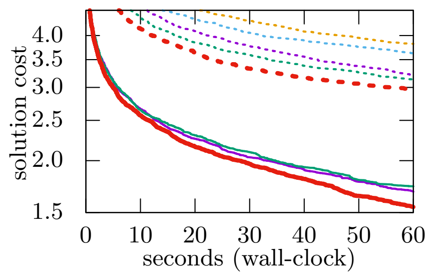
(c) PRRT* relative solve time

Fig. 4. The proposed data structure speeds up parallelized motion planning in the "Home" SE(3) scenario from OMPL. In this scenario, the motion planner finds a path for the red table to move from the lower right to the upper right configuration. The graph in (a) shows the time in milliseconds spent performing nearest neighbor operations (insert, nearest, and k-nearest) relative to the size of the nearest neighbor structure. To illustrate relative impact on overall planner performance, the graph also shows the time spent in collision detection, which is typically the other dominant time consumer in sampling-based motion planners. For the locked versions of the nearest-neighbor structures, the time spent waiting for the lock is shown in the shaded area—the lower boundary of the region is the time spent performing a nearest neigbor operation, and the height of the region is the time the planner must wait for the nearest neighbor operation including the lock. Locked structures (dotted lines) become prohibitively expensive to benchmark past a graph size of 10^5. With the concurrent data structures (solid lines) the total wall-clock time to generate a data structure with 10^6 points averaged between 1 and 2 minutes. The graph in (b) shows the average path cost relative to the estimated optimal path cost as it converges over wall-clock time. The graph in (c) shows the time relative to the proposed method to compute the same solution cost—the proposed method finds the same solution 10% to 30% faster than previous lock-free methods.

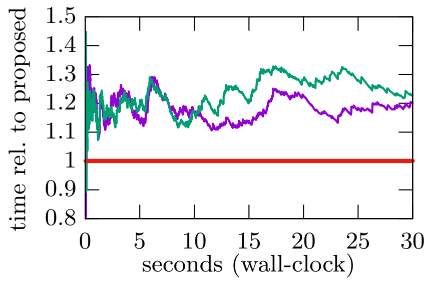




(a) Nearest neighbor time



(b) PRRT* convergence



(c) PRRT* relative solve time

Fig. 5. Speeding up planning on OMPL's"Cubicles" scenario. See description in Fig. 4.

locked form. The implementations of the locked versions of the kd-trees do not make use of memory-ordering operations, and thus run slightly faster in the absence of concurrency. In all experiments, the leaf nodes of the proposed method are configured to have a capacity of 8.

Figures 4 and 5 show two evaluated scenarios involving motion planning for a robot in SE(3). In both scenarios the motion planner must find, and asymptotically optimize, a path for a rigid body robot through a 3D environment. The topological space for nearest neighbor searching is thus $\mathbb{R}^3 \times \mathrm{SO}(3)$. We set the SO(3) distance scale factor to $\alpha_{\mathrm{SO}(3)} = 100$, and leave the $\alpha_{\mathbb{R}^3} = 1$ (the default). The $\mathbb{R}^3$ space extends for hundreds of units, so this makes the two sub-topologies approximately evenly weighted. This weighting has two effects: (1) it makes rotations more expensive, thus as the motion planner converges, the robot rotates less freely than otherwise, and (2) it ensures that the kd-tree splits both $\mathbb{R}^3$ and SO(3) axes.

The figures 4(a) and 5(a) show the time spent in nearest neighbor operations (both inserts and searches) per sampling iteration based upon the size of the PRRT* graph (which is equivalent to the number of points in the nearest neighbor data structure). These graphs show both the time spent searching (bottom line of shaded regions) and the time spent waiting on a lock (shaded regions). We generate a data structure sizes up to 10^5 with the locked versions, stopping then because it becomes too time consuming to continue to the next order of magnitude. The concurrent versions of the kd-tree continue to 1 million. In both graphs we observe that our proposed method performs better than alternatives, even under high concurrency, with roughly half the time (the graph is log scaled) spent compared to the best alternatives.

To demonstrate the relative impact on the motion planner, the graph includes the time spent in collision detection—which typically is the other most time consuming part of a sampling-based motion planner. From the graphs, we observe the time spent in collision detection shrinks as its computation time is a function of shrinking expected distance between random samples. We observe that nearest neighbor operations eventually dominate the per-iteration time.

The figures 4(b) and 5(b) show the overall effect on convergence rate of the asymptotically-optimal sampling-based planner. Due to the acceleration of each iteration, the motion planner is able to find lower-cost paths faster. The alternate presentation of the same data in 4(c) and 5(c), shows that the proposed method results in approximately 20% to 30% faster convergence of PRRT*.

7 Conclusion

This paper proposes and evaluates an exact nearest neighbor data structure that handles concurrent inserts and queries. Based on a kd-tree, the proposed data structure supports searching nearest neighbors on topologies relevant to robotics. The paper described how to support Cartesian products of an arbitrary number of Euclidean and SO(3) spaces with a distance metric that is the weighted sum of sub-topology components within the concurrent data structure.

In evaluation, a parallelized asymptotically-optimal sampling-based motion planner used the proposed data structure to accelerate motion planning. The proposed data structure enabled higher performance than the motion planner's default lock-free kd-tree. Furthermore, the faster performance relative to the lock-based alternatives demonstrates the importance of having a concurrent data structure in parallel processing algorithms such as sampling-based motion planning that depend heavily on nearest-neighbor searching.

In future work, we plan to explore using the proposed data structure with approximate nearest neighbor searching while maintaining its design for concurrency. Observing that the batching approach contributes significantly to performance, we also plan on exploring tuning the batch-size parameter. The batch-size might be optimized based upon static configuration, such as the topology; or on dynamic data, such as the leaf node's position in the tree and region in space.

Acknowledgments. This research was supported in part by the U.S. National Science Foundation (NSF) under Awards IIS-1149965 and CCF-1533844.

References

1. Amato, N.M., Dale, L.K.: Probabilistic roadmap methods are embarrassingly parallel. In: Proc. IEEE International Conference Robotics and Automation. pp. 688–694 (1999)

2. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on. pp. 459–468. IEEE (2006)
3. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM 45(6), 891–923 (Nov 1998)
4. Beis, J.S., Lowe, D.G.: Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In: Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on. pp. 1000–1006. IEEE (1997)
5. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM 18(9), 509–517 (Sep 1975)
6. Brin, S.: Near neighbor search in large metric spaces. In: Proc. 21st Conf. on Very Large Databases (VLDB). pp. 574–584. Zurich, Switzerland (1995)
7. Choset, H., Lynch, K.M., Hutchinson, S.A., Kantor, G.A., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press (2005)
8. Şucan, I.A., Moll, M., Kavraki, L.E.: The Open Motion Planning Library. IEEE Robotics and Automation Magazine 19(4), 72–82 (Dec 2012), http://ompl.kavrakilab.org
9. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS) 3(3), 209–226 (1977)
10. Ichnowski, J., Alterovitz, R.: Scalable multicore motion planning using lock-free concurrency. IEEE Transactions on Robotics 30(5), 1123–1136 (2014)
11. Ichnowski, J., Alterovitz, R.: Fast nearest neighbor search in SE(3) for sampling-based motion planning. In: Algorithmic Foundations of Robotics XI, pp. 197–214. Springer (2015)
12. ISO/IEC: ISO international standard ISO/IEC 14882:2017(E)—programming languages—C++. Standard, International Organization for Standards (ISO), Geneva, Switzerland (Dec 2017)
13. Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. Int. J. Robotics Research 30(7), 846–894 (Jun 2011)
14. Kuipers, J.B., et al.: Quaternions and rotation sequences, vol. 66. Princeton university press Princeton (1999)
15. Kung, H., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Transactions on Database Systems (TODS) 5(3), 354–382 (1980)
16. LaValle, S.M., Kuffner, J.J.: Rapidly-exploring random trees: Progress and prospects. In: Donald, B.R., Others (eds.) Algorithmic and Computational Robotics: New Directions, pp. 293–308. AK Peters, Natick, MA (2001)
17. Sherlock, C., Golightly, A., Henderson, D.A.: Adaptive, delayed-acceptance MCMC for targets with expensive likelihoods. Journal of Computational and Graphical Statistics 26(2), 434–444 (2017)
18. Silpa-Anan, C., Hartley, R.: Optimised kd-trees for fast image descriptor matching. In: Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on. pp. 1–8. IEEE (2008)
19. Sproull, R.F.: Refinements to nearest-neighbor searching in k-dimensional trees. Algorithmica 6(1-6), 579–589 (1991)
20. Yershova, A., LaValle, S.M.: Improving motion planning algorithms by efficient nearest-neighbor searching. IEEE Trans. Robotics 23(1), 151–157 (Feb 2007)