Parallel Batch-Dynamic kd-trees

by

Rahul Yesantharao

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author .....
Department of Electrical Engineering and Computer Science
December 12, 2021

Certified by .....
Julian Shun
Associate Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Parallel Batch-Dynamic kd-trees

by

Rahul Yesantharao

Submitted to the Department of Electrical Engineering and Computer Science
on December 12, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

kd-trees are widely used in parallel databases to support efficient neighborhood and similarity queries. Supporting parallel updates to kd-trees is therefore an important operation. In this paper, we present BDL-tree, a parallel, batch-dynamic implementation of a kd-tree that allows for efficient parallel k-NN queries over dynamically changing point sets. BDL-trees consist of a log-structured set of kd-trees which can be used to efficiently insert or delete batches of points in parallel with polylogarithmic depth. Specifically, given a BDL-tree with n points, each batch of B updates takes $O(B \log^2 (n + B))$ amortized work and $O(\log (n + B) \log \log (n + B))$ depth (parallel time). We provide an optimized multicore implementation of BDL-trees. Our optimizations include parallel cache-oblivious kd-tree construction and parallel bloom filter construction.

Our experiments on a 36-core machine with two-way hyper-threading using a variety of synthetic and real-world datasets show that our implementation of BDL-tree achieves a self-relative speedup of up to 34.8× (28.4× on average) for batch insertions, up to 35.5× (27.2× on average) for batch deletions, and up to 46.1× (40.0× on average) for k-nearest neighbor queries. In addition, it achieves throughputs of up to 14.5 million updates/second for batch-parallel updates and 6.7 million queries/second for k-NN queries. We compare to two baseline kd-tree implementations and demonstrate that BDL-trees achieve a good tradeoff between the two baseline options for implementing batch updates.

Thesis Supervisor: Julian Shun
Title: Associate Professor

Acknowledgments

Thank you to Yiqiu Wang for his invaluable guidance, patience, and mentorship throughout this project. I would also like to thank Professor Julian Shun and Dr. Laxman Dhulipala for all of their expertise and support in helping to guide and shape my research.

Thank you to all the friends I made at MIT, who made my time here beyond special. Thank you to my parents, who have always gone above and beyond in encouraging and supporting me in pursuing my passions. Finally, thank you to my sisters Pooja and Lekha, who amaze and inspire me every single day.

This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nearest neighbor search is used in a wide range of applications, such as in databases, machine learning, data compression, and cluster analysis. One popular data structure for supporting k-nearest neighbor (k-NN) search in low dimensional spatial data is the kd-tree, originally developed by Bentley [7], as it efficiently builds a recursive spatial partition over point sets.

There has been a significant body of work (e.g., [7, 8, 9, 3, 4, 15, 36, 37, 32]) devoted to developing better kd-tree variants, both in terms of parallelization and spatial heuristics. However, none of these approaches tackle the problem of parallelizing batched updates, which is important given that many real-world data sets are being frequently updated. In particular, in a scenario where the set of points is being updated in parallel, existing approaches either become imbalanced or require full rebuilds over the new point set. The upper tree in Figure 1-1 shows the baseline approach that simply rebuilds the kd-tree on every insert and delete, maintaining perfect spatial balance but adding overhead to the update operations. On the other hand, the lower tree in Figure 1-1 shows the other baseline approach, which never rebuilds and instead only inserts points into the existing spatial partition and marks deleted points as tombstones. This gives fast updates at the cost of potentially skewed spatial partitions.

Procopiuc et al. [32] tackled the problem of a dynamically changing point set with their Bkd-Tree, a data structure for maintaining spatial balance in the face of batched

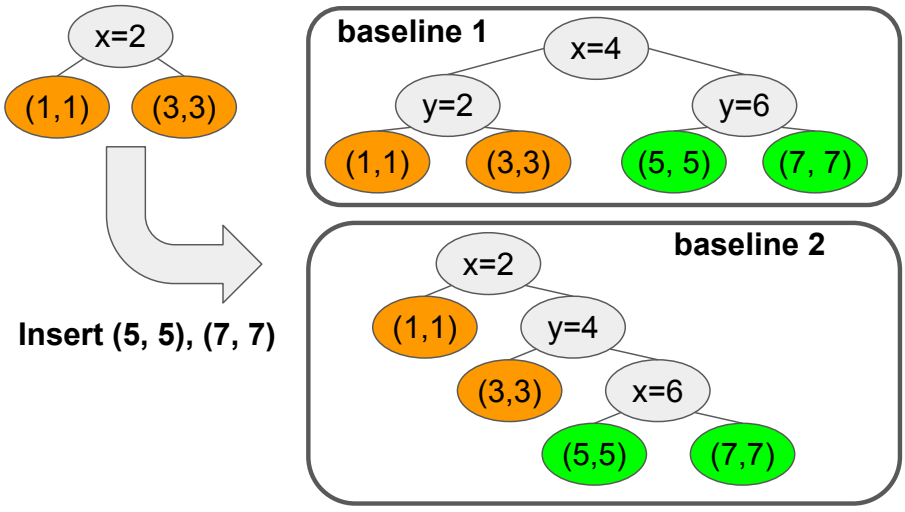


Figure 1-1: Difference in baseline update strategies when the 2-dimensional points $(5, 5)$ and $(7, 7)$ are inserted into a spatial-median kd-tree initially constructed on the points $(1, 1)$ and $(3, 3)$. The internal nodes are labeled with the splitting dimension and the coordinate of the split in that dimension. Baseline 1 rebuilds the kd-tree on every insertion and deletion, while baseline 2 does not rebuild the tree and instead inserts points into the existing spatial partition.

updates. However, it was developed for the external memory case and is not parallel. Similarly, Agarwal et al. [3] developed a dynamic cache-oblivious kd-tree, but it was not parallel and did not support batch-dynamic operations.

We adapt these approaches and develop BDL-tree, a new parallel, in-memory kd-tree-based data structure that supports batch-dynamic operations (in particular, batch construction, insertion, and deletion) as well as exact k-NN queries. BDL-trees consist of a set of exponentially growing kd-trees and perform batched updates in parallel. This structure can be seen in Figure 1-2. Just as in the Bkd-tree and cache-oblivious tree, our tree structure consists of a small buffer region followed by exponentially growing static kd-trees. Inserts are performed by rebuilding the minimum number of trees necessary to maintain fully constructed static trees. Deletes are performed on the underlying trees, and we rebuild the trees whenever they drop to below half of their original capacity. Our use of parallelism, batched updates, and our approach to maintaining balance after deletion are all distinct from the previous trees that we draw inspiration from. We show that given a BDL-tree with n points, each batch of B

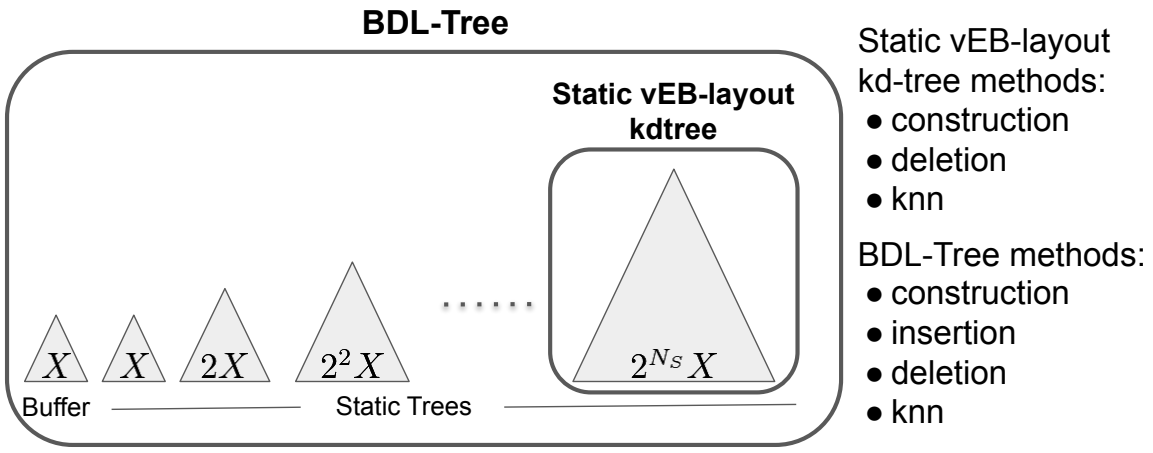


Figure 1-2: Logarithmic method used in BDL-tree, with N_s underlying static kd-trees and a buffer kd-tree of size X.

updates takes $O(B \log^2(n+B))$ amortized work and $O(\log(n+B) \log\log(n+B))$ depth (parallel time). As part of our work, we develop, to our knowledge, the first parallel algorithm for the construction of cache-oblivious kd-trees. Our construction algorithm takes $O(n \log n)$ work and $O(\log n \log\log n)$ depth. In addition, we implement parallel bloom filters to improve the performance of batch updates in practice. We also present a cache-efficient method for performing k-NN queries in BDL-tree. We show theoretically that BDL-trees have strong asymptotic bounds on the work and depth of its operations.

We experimentally evaluate BDL-trees by designing a set of benchmarks to compare its performance against the two baseline approaches described above, which we implemented using similar optimizations. First, we perform scalability tests for each of the four main operations, construction, batch insertion, batch deletion, and k-NN in order to evaluate the scalability of our data structure on many cores. On a 36-core machine with two-way hyper-threading, we find that our data structure achieves self-relative speedups of up to 35.4× (30.0× on average) for construction, up to 35.0× (28.3× on average) for batch insertion, up to 33.1× (28.5× on average) for batch deletion, and up to 46.1× (40.0× on average) for full k-NN. The largest dataset we test consists of 321 million 3-dimensional points. Then, we design a set of benchmarks that perform a mixed set of updates and queries in or-

der to better understand the performance of BDL-trees in realistic scenarios. We find that, when faced with a mixed set of batch operations, BDL-trees consistently outperforms the two baselines and presents the best option for such a mixed dynamic setting. Our source code implementing BDL-trees is publicly available at `https://github.com/rahulyesantharao/batch-dynamic-kdtree`.

Chapter 2

Preliminaries

2.1 kd-tree

The kd-tree, first proposed by Bentley [7], is a binary tree data structure that arranges and holds spatial data to speed up spatial queries. Given a set P of n d-dimensional points, the kd-tree is a balanced binary tree where each node represents a bounding box of a subset of the input points. The root node represents all of the points (and thus the tightest bounding box that includes all the points in P). Each non-leaf node holds a splitting dimension and splitting value that splits its bounding box into two halves using an axis-aligned hyperplane in the splitting dimension. Each child node represents the points in one of the two halves. This recursive splitting stops when the nodes hold some small constant number of points—these nodes are the leaves and directly represent the points.

2.1.1 k-NN Search using the kd-tree

Given a query coordinate q, a k-NN query finds the k nearest neighbors of q amongst elements of the kd-tree by performing a pruned search on the tree [22]. The canonical approach is to traverse the tree while inserting points in the current node into a buffer that maintains only the k nearest neighbors encountered so far. Then, entire subtrees can be pruned during the traversal based on the distance of the k-th nearest neighbor

found so far.

2.1.2 Dual-Tree k-NN

Besides the canonical approach explained above, there has been a lot of prior work on “dual-tree" approaches [16, 15, 24, 29], which provide speedups on serial batched k-NN queries. In particular, the dual-tree approach involves building a second kd-tree over the set of query points, and then exploring the two trees simultaneously in order to exploit the spatial partitioning provided by the kd-tree structure and reduce the overall work required to perform the search. This approach was theoretically very well-suited to our logarithmic tree structure because we have several individual kd-trees over which we would like to perform a k-NN search for the same set of query points. We can perform all of the searches with only a single tree built over the set of query points. We parallelized and tested this dual-tree approach in our work.

2.2 Batch-Dynamic Data Structures

The concept of a parallel batch-dynamic data structure has become popular in recent years [35, 18] as an important paradigm due to the availability of large (dynamic) datasets undergoing rapid changes. The idea is to batch together operations of a single type and perform them as a single batched update, rather than one at a time. This approach offers two benefits. First, from a usability perspective, it is often the case (especially in applications with a lot of data) that operations on a data structure can be grouped into phases or batches of a single type, so this restriction in the usage of the data structure does not significantly limit the usefulness of the data structure. Secondly, from a performance perspective, batching together operations of a single type allows us to group together the involved work and derive significantly more parallelism than otherwise might have been possible (while also avoiding the concurrency issues that might arise with batching together operations of varying types).

2.3 Baselines

In order to benchmark and test the performance of BDL-trees, we implement two parallel baseline kd-trees that use opposite strategies for providing batch-dynamism. In particular, the first baseline kd-tree simply rebuilds the tree after every batch insertion and batch deletion. With this approach, the tree is able to maintain a fully balanced structure in the face of a dynamically changing dataset, enabling consistently high performance for k-NN queries. This comes at the cost of reduced performance for updates. The second baseline kd-tree never rebuilds the tree and simply maintains the initial spatial partition, inserting points into and deleting points from the existing structure. This allows for extremely fast batch insertions and deletions, but could potentially lead to a skewed structure and cause reduced k-NN performance. The difference between these two baselines is graphically demonstrated in Figure 1-1. We demonstrate experimentally that BDL-tree achieves a balanced tradeoff between these two baseline options for batch-dynamic parallel kd-trees on which we are performing k-NN queries. It outperforms both in the dynamic setting where k-NN queries and batched updates are all being used.

2.4 Logarithmic Method

The logarithmic method [8, 9] for converting static data structures into dynamic ones is a very general idea. At a high level, the idea is to partition the static data structure into multiple structures with exponentially growing sizes (powers of 2). Then, inserts are performed by only rebuilding the smallest structure necessary to account for the new points. In the specific case of the kd-tree, a set of N_s static kd-trees is allocated, with capacities in the set $[2^0, 2^1, \ldots, 2^{N_s-1}]$, as well as an extra buffer tree with size 2^0. Then, when an insert is performed, the insert cascades up from the buffer tree, rebuilding into the first empty tree with all the points from the lower trees. If desired, the sizes of all of the trees can be multiplied by a buffer size X, which is some constant that is tuned for performance. This structure is illustrated in Figure 1-2. In the figure,

all of the trees shown are full; one can imagine that the tree with size 2^3X is empty, so the next insert would cause the buffer and trees 0, 1, and 2 to cascade up to it.

2.5 Computational Model

We use the standard work-depth model [14, 27] for multicore algorithms to analyze theoretical efficiency. The **work** of an algorithm is the total number of operations used and the **depth** is the length of the longest sequential dependence (i.e., the parallel running time). An algorithm with work W and depth D can be executed on p processors in $W/p + O(D)$ expected time [13]. Our goal is to come up with parallel algorithms that have low work and depth.

2.5.1 Parallel Primitives

We use the following parallel primitives in our algorithms.

- **Prefix sum** takes as input a sequence of values $[a_1, a_2, \ldots, a_n]$, an associative binary operator $\oplus$, and an identity i, and returns the sequence $[i, a_1, (a_1 \oplus a_2), ..., (a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1})]$ as well as the overall sum of the elements. Prefix sum can be implemented in $O(n)$ work and $O(\log n)$ depth [27].

- **Partition** takes an array A, a predicate function f, and a partition value p and outputs a new array such that all the values $a \in A, a < p$ appear before all the values $a \in A, a \geq p$. Partition can be implemented in $O(n)$ work and $O(\log n)$ depth [27].

- **Median partition** takes n elements and a comparator and partitions the elements based on the median value in $O(n)$ work and $O(\log n \log\log n)$ depth [27].

Chapter 3

Related Work

Numerous tree data structures for spatial search have been proposed in the literature, including the kd-tree [7], quadtree [20], ball tree [30], cover tree [10], and the R-tree [25]. Among them, the kd-tree is a well-known data structure proposed by Bentley [7] in 1975. It is simple and yet can handle many types of queries efficiently. Bentley showed that the kd-tree incurs $O(\log n)$ time for insertion and deletion of random nodes, and logarithmic average query time was observed in practice. The data structure has been extensively studied in the parallel setting. Shevtsov [33] et al. proposed fast construction algorithms for the kd-tree for ray tracing, by using binning techniques to distribute the workload among parallel processors. Agarwal [4] et al. proposed parallel algorithms for a series of spatial trees, including the kd-tree under the massively parallel communication (MPC) model. Wehr and Radkowski [36] proposed fast sorting-based algorithms for constructing kd-trees on the GPU. Zellmann [37] proposed algorithms for CPUs and GPUs for kd-trees used in graphics rendering.

The idea of decomposing a data structure into a logarithmic number of structures for the sake of dynamism has been proposed and used in many different scenarios. Bentley [8, 9] first proposed dynamic structures for decomposable search problems. Specifically, he proposed general methods for converting static data structures into dynamic ones with logarithmic multiplicative overhead in cost, using a set of static data structures with sizes given by increasing powers of two. This is a very impor-

tant idea, because kd-trees are generally packed into memory by their construction (both for convenience and for better memory utilization and locality) [7], so inserting into a kd-tree is generally a difficult problem. Beyond the memory issues, however, the more fundamental issue with inserting into a kd-tree is that inserts can cause imbalance in the tree, which is built according to an original set of points. Thus, too many inserts can cause the tree to become very imbalanced and cause queries to slow down. Agarwal et al. [4] designed cache-oblivious data structures for orthogonal range searching using ideas from the log-structured tree. This was the first application of the van Emde Boas layout [17], which was originally proposed for the cache-oblivious B-tree [6], to the kd-tree structure. This is a natural application because the layout itself can be applied to any balanced binary tree structure to achieve cache-oblivious traversals. Procopiuc et al. [32] proposed and implemented the Bkd-Tree, which uses the logarithmic method [8, 9] to maintain a balanced, dynamic kd-tree in external memory. O'Neil et al. [31] proposed the log-structured merge (LSM) tree, an efficient file indexing data structure that supports efficient dynamic inserts and deletes in systems with multiple storage hierarchies. Specifically, batches of updates are cascaded over time, from faster storage media to slower ones. A parallel version of the LSM tree has been implemented [34] by dividing the key-space independently across cores, and processing it in a data-parallel fashion. Our work, on the other hand, proposes a parallel batch-dynamic kd-tree, where each batch of updates can be processed efficiently in parallel, while supporting efficient k-NN queries.

We have recently discovered that, concurrent with our work, Dobson and Blelloch [19] developed the zd-tree, a data structure that also supports parallel batch-dynamic updates and k-NN queries. Their insight is to modify the basic kd-tree structure by sorting points based on their Morton ordering and splitting at each level based on a bit in the Morton ordering. For k-NN queries, they provide a root-based method, which is the traditional top-down k-NN search algorithm and is what we implement, as well as a leaf-based method, which directly starts from the leaves and searches up for nearest neighbors. The leaf-based method works and is faster when the query points are points in the dataset, as it saves the downward traversal to

search for the query points. They prove strong work and depth bounds for construction, batch updates, and k-NN queries assuming data sets with bounded expansion constant and bounded ratio. Without these assumptions, however, our bounds would be at least as good as theirs.

In Dobson and Blelloch's implementation, they optimize k-NN queries by presorting the query points using the Morton ordering to improve cache locality. Their algorithm assumes that there is a maximum bounding box where all future data will fit into, while our algorithm does not make this assumption. Their implementation discretizes the `double` coordinates into 64-bit integers in order to perform Morton sort on them. As a result, they can only use at most $64/d$ bits for each of the d dimensions. Directly extending the implementation to higher dimensions would either lead to larger leaf sizes or using larger-width coordinates for the Morton sort, either of which could add overheads. Their experiments consider only 2D and 3D datasets, while we test on higher dimensional datasets as well. For 2D and 3D datasets, the running times that they report seem to be in the same ballpark as our times (their k-NN queries for constructing the k-NN graph are much faster than ours due to the presorting described above), after adjusting for number of processors and dataset sizes, but it would be interesting future work to experimentally compare the codes on the same platform and datasets, including higher-dimensional ones. It would also be interesting to integrate some of their optimizations into our code, and to combine the approaches to build a log-structured version of the zd-tree.

Chapter 4

BDL-tree Algorithms

In this chapter, we introduce BDL-tree, a parallel batch-dynamic kd-tree implemented using the logarithmic method [8, 9] (discussed in Section 2.4). BDL-trees build on ideas from the Bkd-Tree by Procopiuc et al. [32] and the cache-oblivious kd-tree by Agarwal et al. [3]. The structure is depicted in Figure 1-2.

We implement the underlying kd-trees in an BDL-tree as nodes in a contiguous memory array, where the root node is the first element in the array. The kd-trees are built using the van Emde Boas (vEB) [5, 17, 3] recursive layout. Agarwal [3] et al. show that this memory layout can be used with kd-trees to make traversal cache-oblivious, although dynamic updates on a single tree become very complex. However, in the logarithmic method, the underlying kd-trees themselves are static, and so we are able to sidestep the complexity of cache-oblivious updates on these trees and benefit from the improved cache performance of the vEB layout. For the buffer region of the BDL-tree, we use a regular kd-tree, laid out like a binary-heap in memory (i.e., nodes are in a contiguous array, and the children of index i are $2i$, $2i + 1$). We will discuss the key parallel algorithms that we used in our implementation: construction, deletion, and k-NN on the underlying individual kd-trees (Section 4.1) and construction, insertion, deletion, and k-NN on BDL-tree (Section 4.2). Note that we do not need to support insertions on individual kd-trees, because our BDL-tree simply rebuilds the necessary kd-trees upon insertions. We use subscript S to denote algorithms on the underlying kd-trees, and subscript L to denote algorithms on the

Algorithm 1 Parallel vEB-layout kd-tree Construction

Input: Point Set P

Output: kd-tree over P, laid out with the vEB layout on a contiguous memory array of size $2|P| - 1$.

1: **procedure** BUILDVEB$_{\text{S}}(P)$

2: Allocate $2|P| - 1$ nodes in contiguous memory. The tree nodes will be laid out in this space.

3: BUILDVEBRECURSIVE$_{\text{S}}(P,\ 0,\ 0,\ \lfloor \log(|P|) \rfloor + 1,$ BOTTOM$)$

4: **procedure** BUILDVEBRECURSIVE$_{\text{S}}(Q,\ idx,\ c,\ l,\ t)$
idx: current node index in the memory array
c: current dimension to split on
l: number of levels to build
t: whether we are building the top or bottom of a tree

5: If we hit the base case $n = 1$, then we construct a node at idx. If t is TOP, then we perform a parallel median partition on Q in dimension c and record this split as an internal node. Otherwise, we create a leaf node that represents the points in Q.

6: Compute $l_b = \left\lceil \left\lceil \frac{l+1}{2} \right\rceil \right\rceil$ and $l_t = l - l_b$ (vEB layout).

7: Recursively build the top half of the tree with BUILDVEBRECURSIVE$_{\text{S}}(Q,\ idx,\ c,\ l_t,$ TOP$)$.

8: Compute $idx_b = idx + 2^{l_t} - 1$ as the offset where the top half of the tree was just laid out.

9: Construct the 2^{l_t} lower subtrees in parallel with BUILDVEBRECURSIVE$_{\text{S}}(Q_i,\ idx_i,\ (c+n_t) \mod d,\ l_b,\ t)$ where Q_i is the subarray of points that are held by the parent of this subtree and idx_i is the index at which this subtree is to be placed (precomputed with a parallel prefix sum).

full logarithmic data structure.

4.1 Single-Tree Parallel Algorithms

4.1.1 Parallel vEB Construction

The algorithm for parallel construction of the cache-oblivious kd-tree is shown in Algorithm 1. The function itself is recursive, and so the top level BUILDVEB$_{\text{S}}$ function allocates space on line 2 and calls the recursive function BUILDVEBRECURSIVE$_{\text{S}}$. Refer to Figure 4-1 for a graphical representation of this construction.

The recursive function BUILDVEBRECURSIVE$_{\text{S}}$ maintains state with 5 parameters: a point set Q, a node index idx, a splitting dimension c, the number of levels to

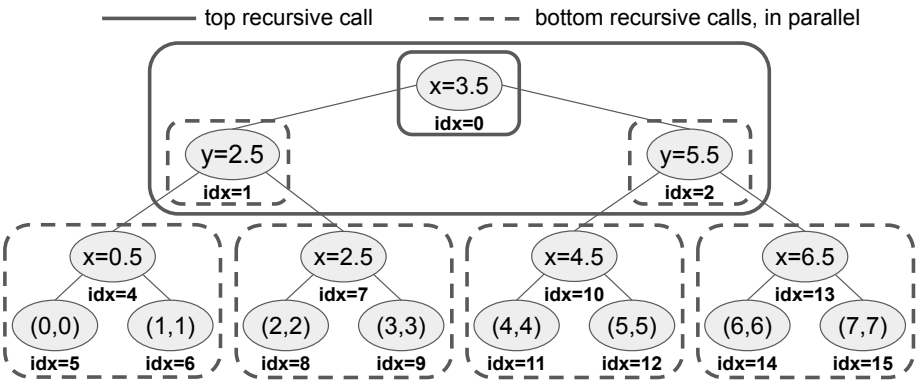


Figure 4-1: Constructing a vEB kd-tree in parallel over 8 2-dimensional points. Note that the top 3 nodes are placed before the remaining 4 bottom subtrees are built in parallel.

build l, and whether it is building the top or bottom of the tree (indicated by t). On line 5, we check for the base case—if the number of levels to build is 1, then we have to construct a node. If this is the top of a tree, then this node will be an internal node, so we perform a parallel median partition in dimension c and save it as an internal node. On the other hand, if this is the bottom of the tree, we construct a leaf node that holds all the points in Q. Lines 6–9 form the recursive step. In accordance with the exponential layout [3], we have to first construct the top "half" of the tree and then the bottom "half". Therefore, on line 6, we compute the number of levels l_b in the bottom portion as the hyperceiling[1] of $\frac{l+1}{2}$ and the remaining number of levels l_t in the top portion of the tree as $l - l_b$. On line 7, we recursively build the top half of the tree. Then, on line 8, we note that because the top half of the tree is a complete binary tree with l_t levels, it will use $2^{l_t} - 1$ nodes. Therefore, we compute $idx_b = idx + 2^{l_t} - 1$, the node index where the bottom half of the tree should start because the trees are laid out consecutively in memory. Finally, on line 9, we construct each of the 2^{l_t} subtrees that fall under the top half of the tree, each with l_b levels. Each of these trees falls into a distinct segment of memory in the array, and so we can perform this construction in parallel across all of the subtrees by precomputing the starting index idx_i for each of the 2^{l_t} subtrees.

[1]The hyperceiling of n, denoted as $\lceil\lceil n \rceil\rceil$ is the smallest power of 2 that is greater than or equal to n, i.e., $2^{\lceil \log n \rceil}$.

We trace this process on an example in Figure 4-1, in which BUILDVEB$_S$ is called on a set P of 8 points. This spawns a call to BUILDVEBRECURSIVE$_S$($P[0:8]$, 0, 0, 4, BOTTOM). On line 6, we will compute $l_b = 2$ and $l_t = 2$, and on line 7, we spawn a recursive call to BUILDVEBRECURSIVE$_S$($P[0:8]$, 0, 0, 2, TOP). This call is shown as the solid box around the top 3 nodes in Figure 4-1. In this call, we will hit one further level of recursion before laying out the 3 nodes in indices 0, 1, 2. Then, the original recursive call will proceed to line 8, where it will compute $idx_b = 3$ as the index to begin laying out the $2^{l_t} = 4$ bottom subtrees. Finally, on line 9, we will precompute that the starting indices for the 4 bottom subtrees are $(idx_0, idx_1, idx_2, idx_3) = (3, 6, 9, 12)$. This results in 4 parallel recursive calls, shown in the 4 lower dashed boxes in Figure 4-1. Each of these recursive calls internally has one more level of recursion to lay out their 3 nodes.

Theorem 1. *The cache-oblivious kd-tree with a vEB layout can be constructed over n points in $O(n \log n)$ work and $O(\log n \log\log n)$ depth.*

Proof. The work bound is obtained by observing that there are $O(\log n)$ levels in the fully-constructed tree, and the median partition at each level takes $O(n)$ work, giving a total of $O(n \log n)$ work. For the depth bound, at each recursive step we first build an upper tree with size $O(\sqrt{n})$, and then construct the lower trees in parallel, each with size $O(\sqrt{n})$. Further, we use an $O(\log n)$-depth prefix sum to compute idx_i at every level except the base case and an $O(\log n \log\log n)$-depth median partition in the base case. Overall, this results in $O(\log n \log\log n)$ depth. □

After the vEB-layout kd-tree is constructed, it can be queried as a regular kd-tree—the only difference is the physical layout of the nodes in memory. The correctness of this recursive algorithm can be seen through induction on the number of levels. In particular, we form two inductive hypotheses:

- BUILDVEBRECURSIVE$_S$(Q, idx, c, l, TOP) creates a contiguous, fully-balanced binary tree with l levels rooted at memory location idx. Furthermore, this binary tree consists of internal kd-tree nodes that equally split the point set Q in half at each level.

- BUILDVEBRECURSIVE$_S(Q, idx, c, \lceil \log |Q| \rceil + 1,$ BOTTOM) creates a contiguous kd-tree with l levels rooted at memory location idx.

The base cases, with $l = 1$, for these inductive hypotheses are explicitly given on line 5. Then, the inductive step follows easily by noting that the definition of hyperceiling implies that the recursive calls on line 9 are all sized such that $l_b = \lceil \log |Q|_i \rceil + 1$.

4.1.2 Parallel Regular Construction

Algorithm 2 Parallel kd-Tree Construction

Input: Point Set P

Output: kd-tree over P, laid out with a binary-heap layout on a contiguous memory space of size $2|P| - 1$.

1: **procedure** BUILD$_S(P)$
2: Allocate $2|P| - 1$ nodes in contiguous memory. The tree nodes will be laid out in this space.
3: BUILDRECURSIVE$_S(P,\ 0,\ 0)$
4: **procedure** BUILDRECURSIVE$_S(Q,\ idx,\ c)$
idx: current node index
c: current dimension to split on
5: If we hit the base case $|Q| = 1$, then we construct a leaf node to represent Q at the current index and return.
6: Otherwise, perform a parallel median partition on Q in dimension c. Then, place an internal node at the current index to represent this split.
7: Compute $med = |Q|/2$. Then, construct the left and right subtrees in parallel with

$$\begin{aligned} &\text{BUILDRECURSIVE}_S(Q[:|Q|/2],\ 2idx + 1,\ (c+1) \mod d) \\ &\text{BUILDRECURSIVE}_S(Q[|Q|/2:],\ 2idx + 2,\ (c+1) \mod d) \end{aligned}$$

The algorithm for parallel construction of a regular kd-tree is shown in Algorithm 2. The function itself is recursive, and so the top level BUILD$_S$ function allocates space on line 2 and calls the recursive BUILDRECURSIVE$_S$.

The recursive function BUILDRECURSIVE$_S$ maintains state with 3 parameters: the set of points Q, the current node idx, and the current splitting dimension c. On line 5, we check the base case where only a single point is left in Q—in this case, we construct a leaf node at idx to represent Q and return. Lines 6–7 represent

Algorithm 3 Parallel *kd*-Tree Deletion

Input: Point Set P

1: **procedure** ERASE$_{\text{S}}$(P)
2: ERASERECURSIVE$_{\text{S}}$(P, 0)
3: **procedure** ERASERECURSIVE$_{\text{S}}$(Q, idx)
idx: current node index
4: If the current node is a leaf node, mark any points in the leaf node that are also in Q as deleted. If all of the points in the current leaf are deleted, return NULL. Otherwise, return the current idx.
5: Otherwise, perform a parallel partition on Q around the split represented by the current node. Let Q_l, Q_r be the resulting left and right arrays, respectively, after the partition.
6: Then, recurse on the children in parallel with ERASERECURSIVE$_{\text{S}}$(Q_l, idx_l), ERASERECURSIVE$_{\text{S}}$(Q_r, idx_r), where idx_l and idx_r are the IDs of the left and right children, respectively.
7: If neither of the recursive calls return NULL, reset the left and right children to be the results of these calls and return the current node. If both of the recursive calls return NULL, return NULL. If one of the recursive calls returns NULL and the other does not, return the non-NULL node.

the recursive case. First, on line 6, we perform a parallel median partition of Q in dimension c and construct an internal node to represent this splitting plane. Then, on line 7, we construct the left and right subtrees in parallel.

Theorem 2. *The regular kd-tree with a binary heap-style layout can be constructed over n points in $O(n \log n)$ work and $O(\log^2 n \log\log n)$ depth.*

Proof. We can see the work bound by noting that there are $\log n$ levels in the fully-constructed tree, and the median partition at each level takes $O(n)$ work, giving a total of $O(n \log n)$ work. For the depth, note that there are $O(\log n)$ levels, and the depth at each level is $O(\log n \log\log n)$ for the median partition, resulting in an overall depth of $O(\log^2 n \log\log n)$. □

4.1.3 Parallel Deletion

The algorithm for parallel deletion from a single kd-tree is shown in Algorithm 3. The function itself is recursive, so the top level ERASE$_{\text{S}}$ calls the subroutine ERASERECURSIVE$_{\text{S}}$ on the root node on line 2.

The recursive function EraseRecursive$_S$ acts on one node at a time, represented by the index idx. On line 4, it checks for the base case—if the current node is a leaf node, it simply performs a linear scan to mark any points in the leaf node that are also in Q as deleted. Then, it returns NULL if the entire leaf was emptied; otherwise, it returns the current node idx. Lines 5–7 represent the recursive case. First, on line 5, we perform a parallel partition of Q around the current node's splitting hyperplane. We refer to the lower partition as Q_l and the upper partition as Q_r. On line 6, we recurse on the left and right subtrees in parallel, passing Q_l to the left subtree and Q_r to the right. Finally, line 7 updates the tree structure. We always ensure that every node has 2 children in order to flatten any unnecessary tree traversal. The return value of EraseRecursive$_S$ indicates the node that should take the place of idx in the tree (potentially the same node)—a return value of NULL indicates that the entire subtree rooted at idx was removed. So, if both the left and right child are removed, then we can remove the current node as well by returning NULL. On the other hand, if neither the left or right child are removed, then the subtree is still intact, and we simply reset the left and right child pointers of the current node and return the current node idx, indicating that it was not removed. Finally, if exactly one of the children was removed, then we remove the current node as well and let the remaining child connect directly to its grandparent—in this way, we remove an unnecessary internal splitting node. We do this by simply returning the non-NULL child, signaling that it will take the place of the current node in the kd-tree.

Theorem 3. *Deleting a batch of B points from a single kd-tree constructed over n points can be done in $O(B \log n)$ work and $O(\log B \log n)$ depth in the worst case.*

Proof. We can see the work bound by noting that each of the B points traverse down $O(\log n)$ levels as part of the algorithm. For the depth, note that in the worst-case the parallel partition at each level operates over $O(B)$ points at each level. Because parallel partition has logarithmic depth, this would result in a worst-case $O(\log B)$ depth at each of the $O(\log n)$ levels, giving the overall depth of $O(\log B \log n)$. □

4.1.4 Data-Parallel k-NN

Algorithm 4 Data-Parallel kd-Tree k-NN

Input: Point Set P, int k

Output: An array of arrays of the k nearest neighbors of each point in P.

1: **procedure** KNN(P)
2: Allocate a k-NN buffer buf_p and Call KNNSINGLEPOINT(0, p, k, buf_p) in parallel over all $p \in P$.
3: **procedure** KNNSINGLEPOINT(idx, p, k, buf)
4: If the current node is a leaf, add all points in the leaf to the buf_p and return.
5: Otherwise, recurse on the child c that p falls within based on the current splitting plane.
6: After the recursive call returns, if buf_p does not yet have k points, simply add all the points in the subtree of c's sibling, s.
7: On the other hand, if buf_p has k points, obtain the current radius estimate and call KNNPRUNE on s.
8: **procedure** KNNPRUNE(idx, p, k, buf, rad)
9: Call the box centered on p with side lengths $2 \cdot rad$ the query box.
10: Compute the box intersection of query box with the bounding box of the current node idx.
11: If the boxes are disjoint, return immediately - there are no potential closer neighbors in the subtree.
12: If the query box entirely contains the bounding box, add all the points in the subtree to buf - any of these points could potentially be a closer neighbor.
13: If the boxes otherwise overlap, recurse on the left and right children of idx if it is an internal node. Otherwise, add all of its points to buf.

We execute our k-NN searches in a data-parallel fashion by parallelizing across all of the query points in a batch. The k-NN search for each point is executed serially. We implement a "k-NN buffer", a data structure that maintains a list of the current k-nearest neighbors and provide quick insert functionality to test and insert new points if they are closer than the existing set. The data structure maintains an internal buffer of size $2k$. To insert a point, it simply adds that point to the end of the buffer. If the buffer is filled up, then it uses a serial selection algorithm to partition the buffer around the k-th nearest element and clears out the remaining k elements. This achieves a serial amortized $O(1)$ runtime (because the selection partition step is $O(k)$ and is only performed for every k insertions).

To implement batched k-NN on the kd-tree, we perform a k-NN search for each

individual point in parallel across all the points. We now describe the k-NN method ($\textsc{kNN}_{\text{S}}$) for a single point p. We first allocate a k-NN buffer for the point. Then, we recursively descend through the kd-tree searching for the leaf that p falls into. When we find this leaf, we add all of the points in the leaf to the k-NN buffer. Then, as the recursion unfolds, we check whether the k-NN buffer has k points. If it does not, we add all the points in the sibling of the current node to the k-NN buffer to try to fill up the buffer with nearby points as quickly as possible to improve our estimate of the k-th nearest neighbor. Otherwise, we use the current distance of the k-th nearest neighbor to prune subtrees in the tree. In particular, if the bounding box of the current subtree is entirely contained within the distance of the k-th nearest neighbor, we add all points in the subtree to the k-NN buffer. If the bounding box is entirely disjoint, then we prune the subtree. Finally, if they intersect, we recurse on the subtree. The pseudocode for this structure is shown in Algorithm 4.

Theorem 4. *For a constant k, k-NN queries over a batch of B points can be performed over a single kd-tree containing n points in worst-case $O(Bn)$ work and worst-case $O(n)$ depth.*

Proof. In the worst-case, we have to search the entire tree, of size $O(n)$, resulting in total work of $O(Bn)$ (due to the amortized $O(1)$ insert cost for k-NN buffers) and depth of $O(n)$, as the queries are done in parallel over the batch, but each search is serial. □

As noted by Bentley [7] and Friedman et al. [22], the work for a single nearest-neighbor query on a kd-tree is empirically found to be logarithmic in n, so the experimental runtime and scalability are much better than suggested by the worst-case bounds.

4.2 Batch-Dynamic Parallel Algorithms

This section describes our algorithms for supporting batch-dynamic updates on BDL-trees.

Algorithm 5 Parallel BDL-tree Batch Insertion

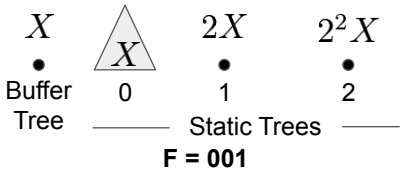
Input: Point Set P

1: **procedure** $\textsc{Insert}_{\text{L}}(P)$
2: Build an integer bitmask F that represents the static trees within the logarithmic tree structure that are currently filled using 1's, and the trees that are empty using 0's.
3: Compute $F_{new} = F + \frac{|P|}{X}$, where X is the buffer tree size. This is the new bitmask of trees that should be filled.
4: Based on the difference between F and F_{new}, determine which trees should be combined into larger trees.
5: Gather the relevant points and construct all the new trees in parallel using $\textsc{BuildvEB}_{\text{S}}$ (or $\textsc{Build}_{\text{S}}$ for the buffer tree).

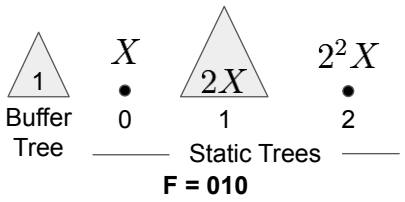
4.2.1 Parallel Insertion

Insertions are performed in the style of the logarithmic method [8, 9], with the goal of maintaining the minimum number of full trees within BDL-tree. Thus, upon inserting a batch B of points, we rebuild larger trees if it is possible using the existing points and the newly inserted batch. This is implemented as shown in Algorithm 5, and depicted in Figure 4-2.

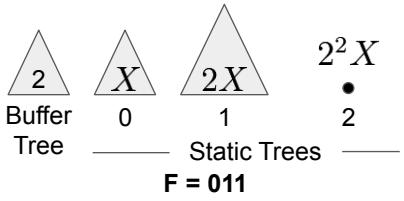
First, on line 2, we build a bitmask F of the current set of full static trees in the logarithmic structure. Then, on line 3, because the buffer kd-tree has size X, we can add $|P|/X$ to F to compute a new bitmask F_{new} of full trees that would result if we added $|P|$ points to the tree structure. As an implementation detail, note that we first add $|P| \mod X$ points to the buffer kd-tree—if we fill up the buffer kd-tree, then we gather the X points from it and treat them as part of P, effectively increasing the size of P by X. Then, on line 4, taking the bitwise difference between these two bitmasks gives the set of trees that should be consolidated into new larger trees—specifically, any tree that is set in F_{new} but not in F must be constructed from trees that are set in F but not in F_{new}. After determining which trees should be combined into new trees, on line 5 we construct all the new trees in parallel—in parallel for each new tree to be constructed, we deconstruct and gather all the points from trees that are being combined into it and then we construct the new tree over these points and any additional required points from P using Algorithm 1.



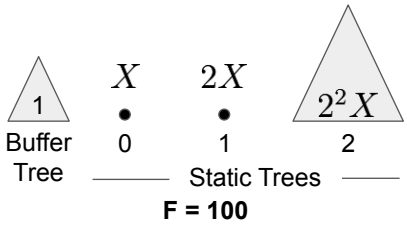
(a) Static tree 0 is full.



(b) Static tree 1 is full and buffer tree has 1 point.



(c) Static trees 0 and 1 are full and buffer tree has 2 points.



(d) Static tree 2 is full and buffer tree has 1 point.

Figure 4-2: A BDL-tree in various configurations with $X > 2$; starting from (a), inserting $X + 1$ points gives (b), then inserting $X + 1$ points gives (c), and then inserting $X - 1$ points gives (d).

Refer to Figure 4-2 for an example of this insertion method (suppose for this example that $X > 2$). In Figure 4-2a, the BDL-tree contains X points, giving a bitmask of $F = 1$ (because only the smallest tree is in use). If we insert $X+1$ points, then we put one node in the buffer tree and compute $F_{new} = 1 + \frac{X}{X} = 2$, and so we have to deconstruct static tree 0 and build static tree 1, as shown in Figure 4-2b. Then, if we insert $X + 1$ points again, then we again put one point in the buffer tree and compute $F_{new} = 2 + \frac{X}{X} = 3$, and so we simply construct tree 0 on the X new points (leaving tree 1 intact), as seen in Figure 4-2c. Finally, if we then insert $X - 1$ points, we note that this would fill the buffer up, so we take 1 point from the buffer and insert X points; then, $F_{new} = 3 + \frac{X}{X} = 4$, and so we deconstruct trees 0, 1 and construct tree 2, as seen in Figure 4-2d.

Algorithm 6 Parallel BDL-tree Batch Deletion

Input: Point Set P

1: **procedure** $\textsc{Erase}_{\text{L}}(P)$
2: In parallel, delete P from each of the underlying trees which is nonempty by calling $\textsc{Erase}_{\text{S}}(P)$ on each of these trees.
3: In parallel, gather the points from any trees that drop to below half of their original capacity into a set R.
4: Call $\textsc{Insert}_{\text{L}}(R)$ to reinsert these points into the log-tree structure.

4.2.2 Parallel Deletion

When deleting a batch of points, the goal is to maintain balance within the subtrees. Thus, if any subtree decreases to less than half of its full capacity, we move all the points down to a smaller subtree in order to maintain balance. As seen in Algorithm 6, this is implemented as a three-step process.

On line 2, we call a parallel bulk erase subroutine on each of the individual trees in parallel in order to actually erase the points from the trees. On line 3, we scan the trees in parallel and collect the points from all trees which have been depleted to less than half of their original capacity. Finally, on line 4, we use the $\textsc{Insert}_{\text{L}}$ routine to reinsert these points into the structure.

Theorem 5. *Given an BDL-tree that was created using only batch insertions and deletions, each batch of B updates takes $O(B \log^2 (n + B))$ amortized work and $O(\log (n + B) \log \log (n + B))$ depth, where n is the number of points in the tree before applying the updates.*

Proof. We first argue the work for only performing insertions starting from an empty BDL-tree. In the worst case, points are added to the structure one by one. Then, similarly to the analysis by Bentley [8], the total work incurred is given by noting that the number of times the i'th tree is rebuilt when inserting m points one by one is $O(2^{\log m - i})$. Then, summing the total work gives $O(\sum_{i=0}^{\log m} 2^i i 2^{\log m - i}) = O(m \log^2 m)$, where we use the work bound from Theorem 1. After inserting a batch of size B, we have $n + B$ points in the BDL-tree, and so the amortized work for the batch is $O(B \log^2(n + B))$. Now, if deletions occurred prior to a batch insertion, and the

Algorithm 7 Data-Parallel BDL-tree k-NN

Input: Point Set S

Output: An array of arrays of the k nearest neighbors of each point in S.

1: **procedure** $\text{kNN}_\text{L}(S)$
2: Allocate a k-NN buffer for each of the points in S.
3: For each nonempty tree in the BDL-tree, in serial, call the parallel subroutine $\text{kNN}_\text{S}(S)$ on the individual tree, passing the same set of buffers.
4: After all of the individual kNN_S calls are complete, gather and return the results from the k-NN buffers.

current BDL-tree has n points, there still must have been n previous insertions (since we started with an empty data structure), and so the work of this batch can still be amortized against those n insertions. We now argue the depth bound. When a batch inserted into the tree, the points from smaller trees can be gathered in worst-case $O(\log(n+B))$ depth (if all the points must be rebuilt) and the rebuilding process takes worst case $O(\log(n+B)\log\log(n+B))$ depth, using the result from Theorem 1.

The initial step of deleting the batch of points from each of the underlying kd-trees incurs $O(B\log^2 n)$ work (there are $O(\log n)$ kd-trees, each taking work $O(B\log n)$) and depth $O(\log B\log n)$. Then, collecting the points that need to be reinserted can be done in worst-case depth $O(\log(n+B))$ and the reinsertion takes $O(\log(n+B)\log\log(n+B))$ depth, from before. Overall, the depth is $O(\log(n+B)\log\log(n+B))$. The amortized work for reinserting points in trees that are less than half full is $O(B\log^2 n)$, as every point we reinsert can be charged to a deletion of another point, either from this batch or from a previous batch. This is because for a tree that is half full, there must be at least as many deletions from the tree as the number of points remaining in the tree. □

4.2.3 Data-Parallel k-NN

In the data-parallel k-NN implementation given in Algorithm 7, we parallelize over the set of points given to search for nearest neighbors. First, on line 2, we allocate a k-NN buffer for each of the points in S. Then, for each of the non-empty trees in

BDL-tree, we call the data-parallel k-NN subroutine on the individual tree, passing in the set S of points and the set of k-NN buffers. Because we reuse the same set of k-NN buffers for each underlying k-NN call, we eventually end up with the k-nearest neighbors across all of the individual trees for each point in S.

Theorem 6. *For a constant k, k-NN queries over a batch of B points over the n points in the BDL-tree can be performed in $O(Bn)$ work and $O(n)$ depth.*

Proof. These bounds follow directly from the bounds of the underlying individual k-NN calls. The k-NN routine on the i'th underlying tree, with size n_i, has worst-case work $O(Bn_i)$ and depth $O(n_i)$. Summing over all i gives the bounds. □

4.2.4 Parallel Dual-Tree k-NN

We also implement a parallel version of the dual-tree k-NN algorithm [16, 24, 15, 29]. As discussed in Section 2.1.2, this is well-suited in theory for our data structure because we have several individual kd-trees over which we would like to perform a k-NN search for the same set of query points. To implement this, we build a single kd-tree over the set of query points. Then, we perform a parallel dual-tree k-NN search on each of the underlying trees in order. The pseudocode for this algorithm in the single-tree case is given in [29]. We modify this work in two ways. First, we parallelize this algorithm by parallelizing the recursive traversal calls on the subtrees. Then, we perform this parallel dual-tree k-NN search on each of the underlying trees in order, one at a time, rather than simply performing it on a single tree. We tested this approach experimentally but found that it did not perform as well as the data-parallel approaches discussed above.

Chapter 5

Implementation and Optimizations

In this chapter, we describe implementation details and optimizations that we developed to speed up the BDL-tree in practice.

5.1 Parallel Bloom Filter

The bloom filter is a probabilistic data structure for testing set membership, which can give false positive matches but no false negative matches [12]. When erasing a batch of points from the BDL-tree, we need to potentially search for every point to be deleted within every individual underlying kd-tree. To mitigate this overhead, we use a bloom filter to prefilter points to be erased from each individual kd-tree that definitely are not contained within it. Specifically, we implemented a parallel bloom filter and added one to each individual kd-tree within the BDL-tree to track the points in that individual kd-tree. Before erasing an input batch from each individual kd-tree, we filter the batch with that kd-tree's bloom filter. This optimization adds some overhead to the construction and insertion subroutines, but provides significant benefits for the delete subroutine. To mitigate some of this overhead, we construct the bloom filters only when they are needed, rather than during construction.

5.2 Data-Parallel k-NN Structure

We tested three different variants of the data-parallel k-NN search in order to determine the best parallelization scheme. Each scheme consists of 2 nested for loops; one over the input points and one over the individual kd-trees. In the first variant, the outer loop is over the points and the inner loop is over the kd-trees. We only parallelize the outer loop, so we perform a k-NN search over all the trees for each point in parallel. In the second variant, we switch the loop order, so the outer loop is over the kd-trees and the inner loop is over points. We parallelize both the inner and outer loops, so we perform a k-NN search over each of the trees in a data-parallel fashion (i.e., search for the nearest neighbors of each point in parallel), but we also parallelize over the trees. In this case, because our k-NN buffer is not thread-safe, for each point we have to allocate a separate buffer for each tree and combine the results at the end, which adds some extra overhead. In the third variant, described in Section 4.2.3, the outer loop is over the kd-trees and the inner loop is over points, but in contrast to the second scheme, we only parallelize the inner loop over points. So, we perform a data-parallel k-NN over each of the individual kd-trees, one at a time. We found that the second scheme gives the most parallelism, but the third scheme has the fastest end-to-end running time, both serially and in parallel. This is due to better cache locality of the third method—each tree is completely processed before moving on to the next one. One other optimization we used in this third scheme was to process the underlying trees in order from largest to smallest. We experimentally found that the data-parallel k-NN search scaled better on large trees, and so this gave better radius estimates earlier (as opposed to processing the trees from smallest to largest).

5.3 Dual-Tree k-NN

When implementing the parallel dual-tree k-NN, we implemented and benchmarked three different variants. In the scheme presented in Section 4.2.4, we build a single

query kd-tree over all of the input points, and then perform a parallel dual-tree k-NN search on each of the underlying trees one at a time. However, we can also parallelize this search, performing the dual-tree k-NN on all the trees in parallel. This requires some modifications to our data structures. First off, because the k-NN buffers are not thread-safe, we have to allocate separate k-NN buffers for each individual tree and combine the results to a single set of k nearest neighbors at the end. In addition, in the dual-tree k-NN algorithm [16, 24, 15, 29], every node in the query tree tracks a radius field representing the current radius of the k-NN search of all points in the subtree rooted at that node. We can enable parallelism across the individual dual-tree k-NN searches either by making this field atomic, so that all the searches share a single radius estimate, or by making this field an array, so that each search has a separate radius estimate. Overall, this gives three different variations on the parallel dual-tree k-NN: the one presented in Section 4.2.4 (referred to as "non-atomic" because the radius estimate is stored as a non-atomic `double`), an "atomic" variant (using an atomic `double` for the radius estimate), and an "array" variant (using a `double` array to store the radius estimates). We found that overall, parallelizing across the trees with both the "atomic" and "array" approaches led to decreased performance and scalability due to the associated overheads and cache-behavior, as seen in Section 6.5.3.

5.4 k-NN Pruning Strategies

In implementing the k-NN approaches described above, we experimented with various different approaches as to how exactly to prune the trees while traversing. Specifically, we experimented with two different pruning options.

The first option applied to both the data-parallel and dual-tree k-NN approaches. When performing a k-NN traversal (both the dual-tree and data-parallel approaches), there are decisions to be made as to when we should update the current radius estimate. The tradeoff to make is that updating the radius estimate adds work overhead (as we must sort our current buffer of k nearest neighbors), but it could

potentially provide a tighter radius to more aggressively prune subtrees. So, we experimented with two different choices. In the first choice, we update the radius estimate every time we recurse down into a new subtree. In the second choice, we only update the radius estimate once each time we step up one recursion level in the overall traversal (i.e. at the start of each subtree traversal). In practice, we found that the second option was generally slightly faster, so we used this approach.

The second pruning variant we experimented with only applied to the data-parallel k-NN. As described in Section 4.1.4, when we are unrolling the traversal recursion and we have not yet found k neighbors, we simply add all the points in the sibling of the current node to the k-NN buffer in order to quickly find a k-th nearest neighbor estimate. Instead of doing this, we could also recurse on the sibling and be more careful with this process; this will theoretically provide a benefit with larger sibling nodes. In practice, we found that this approach either did not provide any benefit or caused a slight slowdown. This is due to the fact that this portion of the code is only executed rarely and very near to the leaves, where the subtrees are all very small. As a result, the overhead of pruning is higher than simply adding all the nodes to the k-NN buffer.

5.5 Parallel Splitting Heuristic

We implemented two different splitting heuristics for our kd-trees: splitting by object median and splitting by spatial median. Object median refers to a true median—at each split, we split around the median of the coordinates of the points in that dimension. On the other hand, for spatial median, we compute the average of the minimum and maximum of the coordinates of the points in the current dimension and use this as the splitting hyperplane. The spatial median is faster to compute, but leads to potentially less balanced trees as it is not strictly splitting the points in half at every level. We implement the object median with a parallel in-place sort [27, 11] and we implement the spatial median in two steps. First, we perform two parallel prefix sums, using min(), max() as the predicate functions, to compute the minimum

and maximum of the values, respectively. Then, we compute the spatial median as the average of these values and perform a parallel partition around this value.

5.6 Buffer Tree

We tested a simple linear array to serve as the buffer “tree" in the BDL-tree, rather than using a small kd-tree, but this severely impacted k-NN performance, as the k-NN work on a linear array scales as the product of the size of the array and the size of the query point set. It did, however, provide minor improvements in update speeds. Ultimately, this was not enough to offset the impact on k-NN performance.

5.7 Coarsening

We introduced a number of optimizations that were controlled by tunable parameters in our implementation. One key optimization was the coarsening of leaves, in which we let each leaf node represent up to 16 points rather than 1. This reduces the amount of space needed for node pointers and also improves cache locality when traversing the tree, as the points in a leaf node are stored contiguously. A second key optimization was the coarsening of the serial base cases of our algorithms, in which we switch from a parallel algorithm to a serial version for recursive cases involving subtrees with less than 1000 points in order to mitigate the overhead of spawning new threads. A third parameter was the buffer tree size, which we set to 1024. One coarsening optimization that we did not have time to attempt was to coarsen the k-NN recursion to stop above the leaves, rather than on the leaves. This could potentially provide benefits if the overhead of an extra level of recursion was larger than simply naively looping over a slightly larger subtree.

Chapter 6

Experiments

We designed a set of experiments to investigate the performance and scalability of BDL-tree and compare it to the two baselines described earlier.

1. **B1** is a baseline described in Section 2.3, where the kd-tree is rebuilt on each batch insertion and deletion in order to maintain balance. This allows for improved query performance (as the tree is always perfectly balanced) at the cost of slowing down dynamic operations.

2. **B2** is another baseline described in Section 2.3. It inserts points directly into the existing tree structure without recalculating spatial splits. This results in very fast inserts and deletes at the cost of potentially skewed trees (which would slow down query performance).

3. **BDL** is our BDL-tree described in Section 4. It represents a tradeoff between batch update performance and query performance. By maintaining a set of balanced trees, it is able to achieve good performance on updates without sacrificing the quality of the spatial partition.

We use the following set of experiments to measure the scalability of BDL-tree and compare its performance characteristics to the baselines.

1. Construction (Section 6.1): construct the tree over a dataset.

2. Insertion (Section 6.2.1): insert fixed-size batches of points into an empty tree until the entire dataset is inserted.

3. Deletion (Section 6.3.1): delete fixed-size batches of points from a tree constructed over the entire dataset until the entire dataset is deleted.

4. k-NN (Section 6.4.1): find the k-nearest neighbors of all the points in the dataset (i.e., compute the k-NN graph of the dataset).

We also designed the following microbenchmarks in order to better explore the design tradeoffs that BDL-tree makes when compared to the baselines.

1. Varying Batch Size (Sections 6.2.2 and 6.3.2): we vary the batch size of the operations used to fully insert or delete the point set to measure the impact of batch size on throughput.

2. Varying k after Batched Inserts (Section 6.4.2): we build a tree using a set of batched insertions and then measure the k-NN search performance with a range of k values to measure the impact of k on throughput and the impact of dynamic updates on k-NN performance.

3. Mixed Insert, Delete, and k-NN Searches (Section 6.4.3): we perform a series of batch updates (insertions and deletions) interspersed with k-NN queries to measure the performance of the data structures over time.

We run all of the experiments over BDL-tree and the two baselines. In addition, for the scalability and batch size experiments, we also compare the object median and spatial median splitting heuristics described in Section 5.5.

The experiments are all run on an AWS c5.18xlarge instance with 2 Intel Xeon Platinum 8124M CPUs (3.00 GHz), for a total of 36 two-way hyper-threaded cores and 144 GB RAM. Our experiments use all hyper-threads unless specified otherwise. We compile our benchmarks with the `g++` compiler (version 9.3.0) with the `-O3` flag, and use ParlayLib [11] for parallelism. All reported running times are the medians of 3 runs, after one extra warm-up run for each experiment.

We run the experiments over 9 datasets, consisting of 6 synthetic datasets and 3 real-world datasets. We use two types of synthetic datasets. The first is **Uniform** (**U**), consisting of points distributed uniformly at random inside a bounding hyper-cube with side length $\sqrt{n}$, where n is the number of points. The second is **VisualVar** (**V**), a clustered dataset with variable-density, produced by Gan and Tao's generator [23]. The generator produces points by performing a random walk in a local region, but jumping to random locations with some probability. For each of these two types, we generate them in 2D, 5D, and 7D, and for 10,000,000 points. We also use 3 real-world datasets: **10D-H-1M** [2, 26] is a 10-dimensional dataset consisting of 928,991 points of home sensor data; **16D-C-4M** [1, 21] is a 16-dimensional dataset consisting of 4,208,261 points of chemical sensor data; and **3D-C-321M** [28] is a 3-dimensional dataset consisting of 321,065,547 points of astronomy data. Due to time constraints, we only ran experiments on **3D-C-321M** using BDL-tree in parallel to demonstrate that BDL-tree can scale to large datasets.

6.1 Construction

In this benchmark, we measure the time required to construct a tree over each of the datasets. The results using an object median splitting heuristic are shown in Table 6.1a and the results using a spatial median splitting heuristic are shown in Table 6.1b. Figure 6-1a shows the scalability of the throughput on the 10M points 7D **Uniform** dataset.

As we can see from the results, **BDL** achieves similar or better performance both serially and in parallel than both **B1** and **B2**, and has similar or better scalability than both. With the object median splitting heuristic, it achieves up to 34.8× speedup, with an average speedup of 28.4×. We also note that the single-threaded runtimes are faster with the spatial-median splitting heuristic than with the object median splitting heuristic. This is expected, because spatial median only involves splitting points at each level compared with finding the median for object-median, hence it is less expensive to compute; however, we also note that the scalability for spatial

	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	20.6s	16.3s	14.4s	0.5s (40.0x)	3.7s (4.5x)	0.4s (34.5x)
2D-V-10M	20.5s	16.2s	14.2s	0.5s (40.3x)	3.6s (4.5x)	0.4s (34.8x)
5D-U-10M	23.3s	20.8s	16.3s	0.7s (35.2x)	4.8s (4.3x)	0.5s (30.3x)
5D-V-10M	22.8s	20.2s	15.8s	0.7s (34.9x)	4.6s (4.4x)	0.5s (29.2x)
7D-U-10M	27.0s	24.0s	17.1s	0.8s (33.0x)	5.4s (4.4x)	0.6s (27.5x)
7D-V-10M	26.2s	23.2s	16.5s	0.8s (33.5x)	5.3s (4.4x)	0.6s (26.6x)
10D-H-1M	1.5s	1.5s	2.8s	0.1s (23.7x)	0.5s (3.4x)	0.1s (23.8x)
16D-C-4M	13.5s	14.5s	11.0s	0.5s (25.2x)	3.8s (3.8x)	0.5s (20.4x)
3D-C-321M	–	–	–	–	–	20.4s

(a) Object median.

	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	10.7s	2.3s	5.3s	0.5s (23.8x)	1.6s (1.4x)	0.4s (13.5x)
2D-V-10M	10.9s	2.5s	5.5s	0.5s (22.8x)	1.7s (1.5x)	0.4s (13.5x)
5D-U-10M	13.7s	3.4s	6.7s	0.8s (18.0x)	2.3s (1.5x)	0.6s (10.7x)
5D-V-10M	14.4s	4.0s	7.1s	0.8s (17.4x)	2.6s (1.5x)	0.7s (10.3x)
7D-U-10M	16.8s	4.4s	7.1s	1.0s (17.2x)	2.9s (1.5x)	0.8s (9.3x)
7D-V-10M	17.3s	5.2s	8.3s	1.0s (16.9x)	3.3s (1.6x)	0.9s (9.2x)
10D-H-1M	1.4s	0.8s	3.2s	0.1s (11.0x)	0.5s (1.6x)	0.2s (15.9x)
16D-C-4M	10.0s	5.3s	7.3s	0.7s (14.0x)	3.0s (1.8x)	0.7s (10.2x)
3D-C-321M	–	–	–	–	–	20.8s

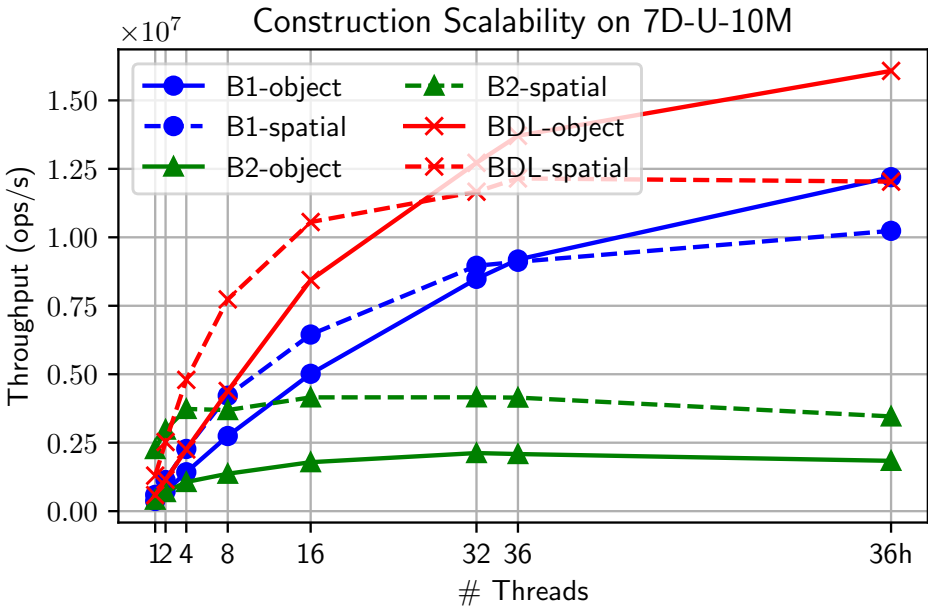
(b) Spatial median.

Table 6.1: Construction times (seconds) for a single thread (1) and 36 cores with hyper-threading (36h). The self-relative speedup for each implementation and dataset is shown in parentheses.

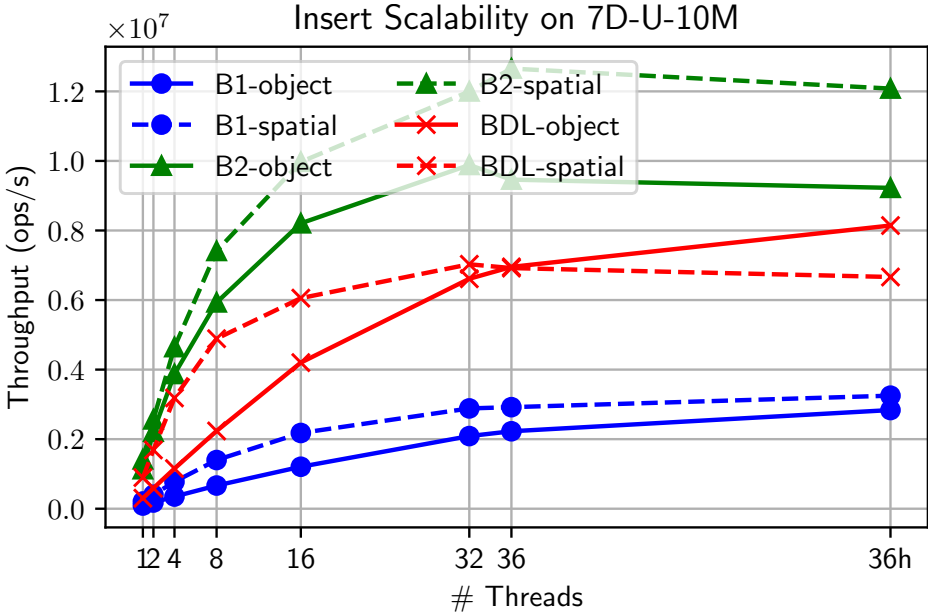
median is lower because there is less work to distribute among parallel threads.

6.2 Insertion

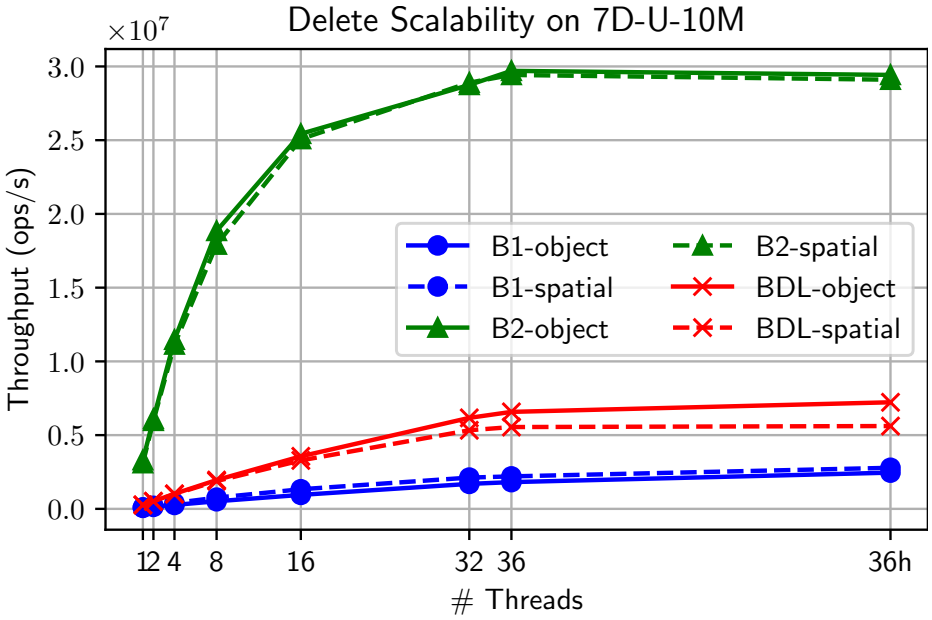
In this benchmark, we measure the performance of our batch insertion implementation as compared to the baselines. We split this experiment into two separate benchmarks, one to measure the full scalability and performance of our implementation, and one to measure the impact of varying batch sizes.



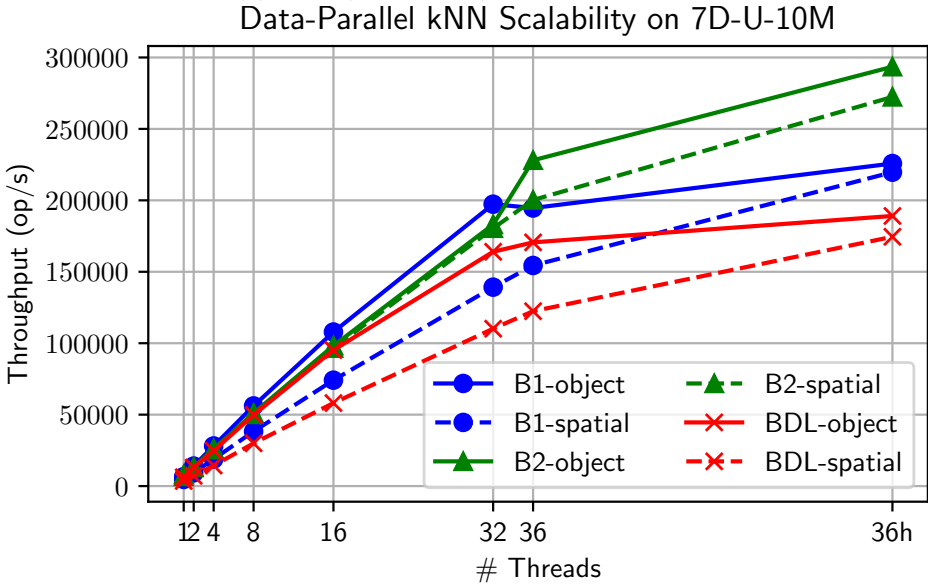
(a) Construction.



(b) 10% (1M points) Batch Insertion.



(c) 10% (1M points) Batch Deletion.



(d) Full (10M points) k-NN for $k = 5$.

Figure 6-1: Plot of throughput (operations per second) of batch operations over thread count for both object and spatial median implementations for the 7D-U-10M dataset.

6.2.1 Scalability

In this benchmark, we measure the time required to insert 10 batches each containing 10% of the points in the dataset into an initially empty tree for each of our two baselines as well as our BDL-tree. The results using object median and spatial median splitting heuristics are shown in Tables 6.2a, 6.2b, respectively. Figure 6-1b shows the scalability of the throughput on the 10M points 7D **Uniform** dataset.

We see that **B2** achieves the best performance on batched inserts—this is due to the fact that it does not perform any extra work to maintain balance and simply directly inserts points into the existing spatial structure. **BDL** achieves the second-best performance—this is due to the fact that it does not have to rebuild the entire tree on every insert, but amortizes the rebuilding work across the batches. Finally, **B1** has the worst performance, as it must fully rebuild on every insertion. Similar to construction, we note that the spatial median heuristic performs better in the serial case but has lower scalability. With the object median splitting heuristic, **BDL** achieves parallel speedup of up to 35.5×, with an average speedup of 27.2×.

6.2.2 Batch Size

In this benchmark, we measure the performance of our batch insertion implementation as the size of the batch varies from 1M points to 5M points. We repeatedly perform batched inserts of the specified size until the entire dataset has been inserted. We provide plots of the results for the 2D **Uniform**, 2D **VisualVar**, 7D **Uniform**, and 7D **VisualVar** datasets in Figures 6-2a 6-2b, 6-2c, 6-2d, respectively. The first striking result is that the throughput decreases for **B2** as the batch size increases, while the throughput increases for **B1** and **BDL**. This is due to the fact that the work that **B2** performs increases as the batch size grows—it has to do more work to compute spatial splits over larger batches, whereas for small batches it quickly computes the upper spatial splits on the first batch and never recomputes them. As a result, **B2** has best throughput at smaller batch sizes, but the worst at large batch sizes. For **B1** and **BDL**, note that the throughput increases as the batch

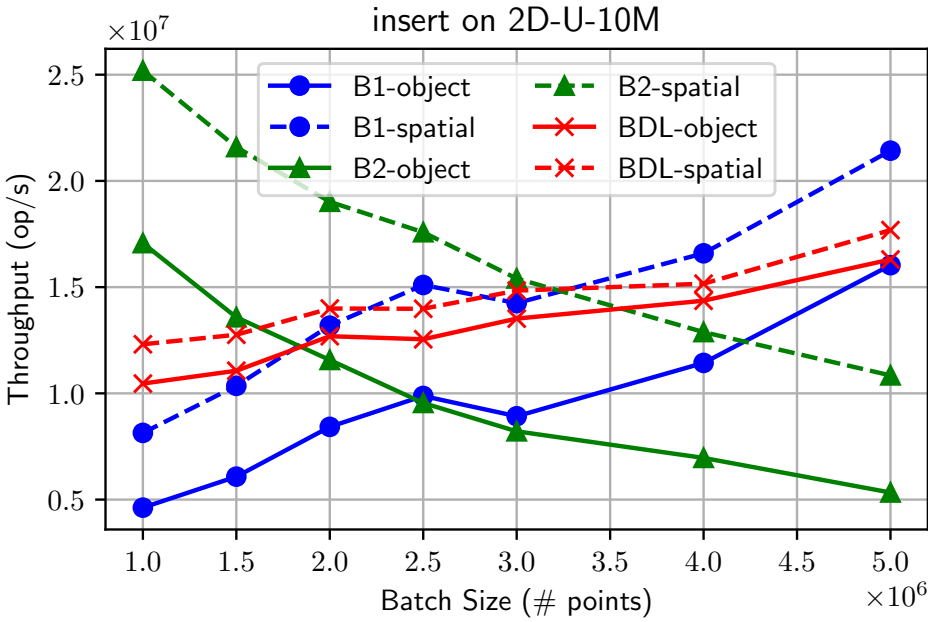
	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	86.1s	5.9s	24.8s	2.2s (39.5x)	0.6s (10.1x)	0.7s (35.4x)
2D-V-10M	86.1s	5.9s	24.4s	2.2s (39.6x)	0.6s (10.2x)	0.7s (35.5x)
5D-U-10M	97.9s	7.6s	29.2s	2.9s (33.6x)	0.9s (8.7x)	1.0s (30.3x)
5D-V-10M	94.5s	7.5s	28.1s	2.8s (33.2x)	0.9s (8.8x)	1.0s (29.3x)
7D-U-10M	109.7s	8.8s	33.0s	3.5s (31.1x)	1.1s (8.1x)	1.2s (26.9x)
7D-V-10M	106.1s	8.7s	31.7s	3.5s (30.7x)	1.1s (8.2x)	1.2s (25.6x)
10D-H-1M	7.9s	0.7s	1.7s	0.4s (22.1x)	0.1s (5.7x)	0.1s (16.3x)
16D-C-4M	66.2s	5.5s	21.1s	3.0s (22.2x)	0.9s (6.4x)	1.1s (18.3x)
3D-C-321M	–	–	–	–	–	20.7s

(a) Object median.

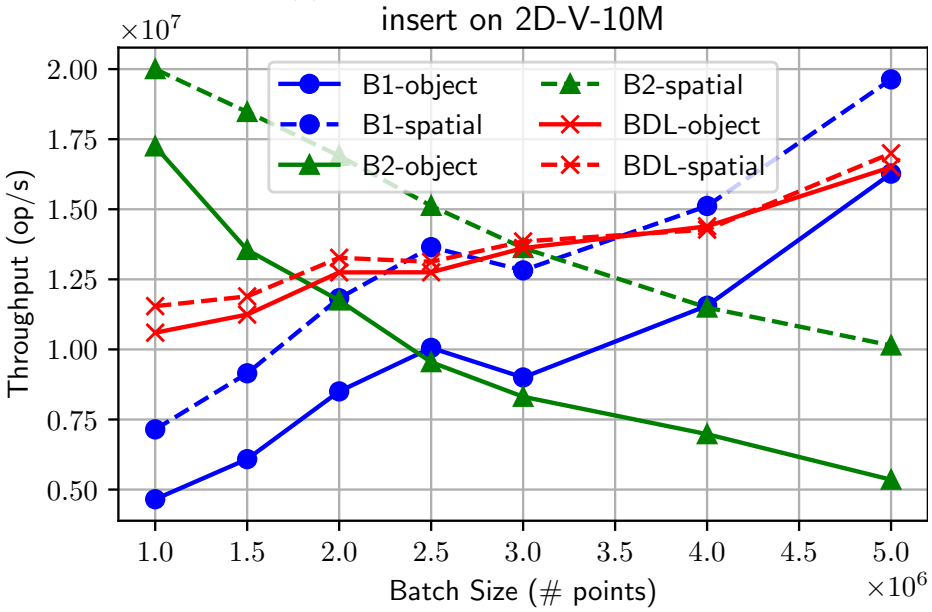
	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	32.5s	4.9s	5.2s	1.2s (26.1x)	0.4s (11.9x)	0.6s (9.2x)
2D-V-10M	33.9s	5.0s	5.8s	1.4s (24.0x)	0.5s (10.2x)	0.6s (9.2x)
5D-U-10M	40.7s	6.1s	8.8s	2.3s (17.5x)	0.7s (9.4x)	1.1s (8.1x)
5D-V-10M	44.5s	6.7s	10.0s	2.8s (15.6x)	0.8s (8.0x)	1.2s (8.0x)
7D-U-10M	48.1s	7.0s	11.2s	3.1s (15.6x)	0.8s (8.5x)	1.5s (7.4x)
7D-V-10M	52.6s	7.6s	12.5s	3.7s (14.4x)	1.0s (7.8x)	1.6s (7.6x)
10D-H-1M	6.3s	0.9s	1.3s	0.6s (10.2x)	0.2s (3.8x)	0.2s (6.6x)
16D-C-4M	42.9s	6.0s	12.9s	3.6s (11.8x)	1.0s (5.9x)	1.5s (8.5x)
3D-C-321M	–	–	–	–	–	15.7s

(b) Spatial median.

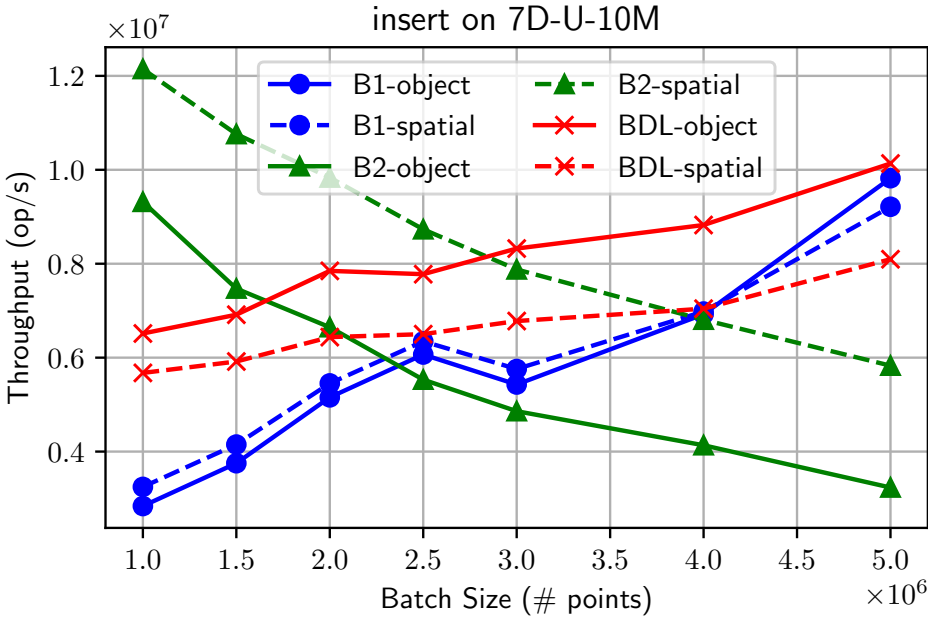
Table 6.2: Batch insertion times (seconds) for a single thread (1) and 36 cores with hyper-threading (36h). The self-relative speedup for each implementation and dataset is shown in parentheses. We insert batches of 10% of each dataset, starting from an empty tree until the entire dataset has been inserted.



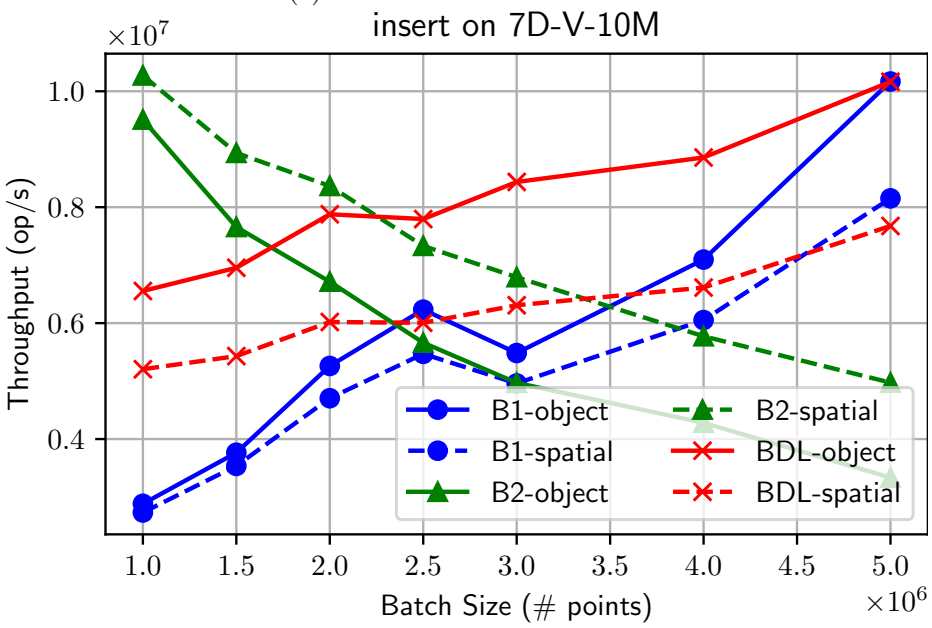
(a) Insertions on 2D-U-10M.



(b) Insertions on 2D-V-10M.



(c) Insertions on 7D-U-10M.



(d) Insertions on 7D-V-10M.

Figure 6-2: Plot of throughput (operations per second) of batch insertions vs. batch size for the 2D-U-10M, 2D-V-10M, 7D-U-10M, and 7D-V-10M datasets.

size increases. This is due to the fact that each insert has an associated overhead of recomputing spatial partitions, and so the larger the batch size, the fewer times this overhead is paid. For larger batch sizes, **BDL** has the best throughput among the three implementations for object median. This is again due to the fact that it amortizes the work across inserts, rather than having to recompute spatial splits over the entire dataset at each insert. Finally, note that in most cases, the spatial median heuristic has a better throughput than its object median counterpart, as it takes less work to compute.

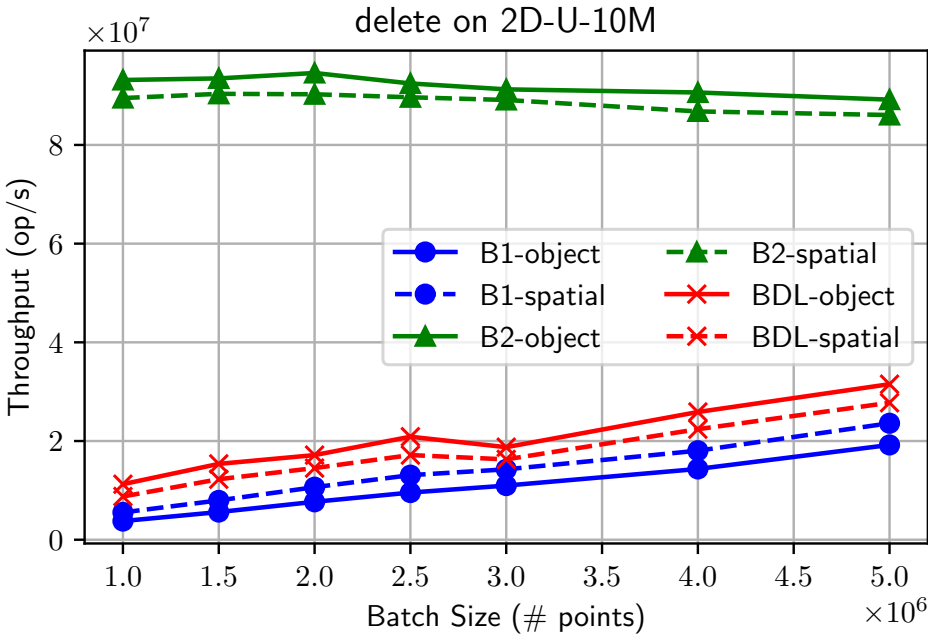
6.3 Deletion

In this benchmark, we measure the performance of our deletion implementation as compared to the baselines. Similar to the insertion experiment, we split this experiment into two separate benchmarks, one to measure the scalability and performance of our implementation, and one to measure the impact of varying batch size.

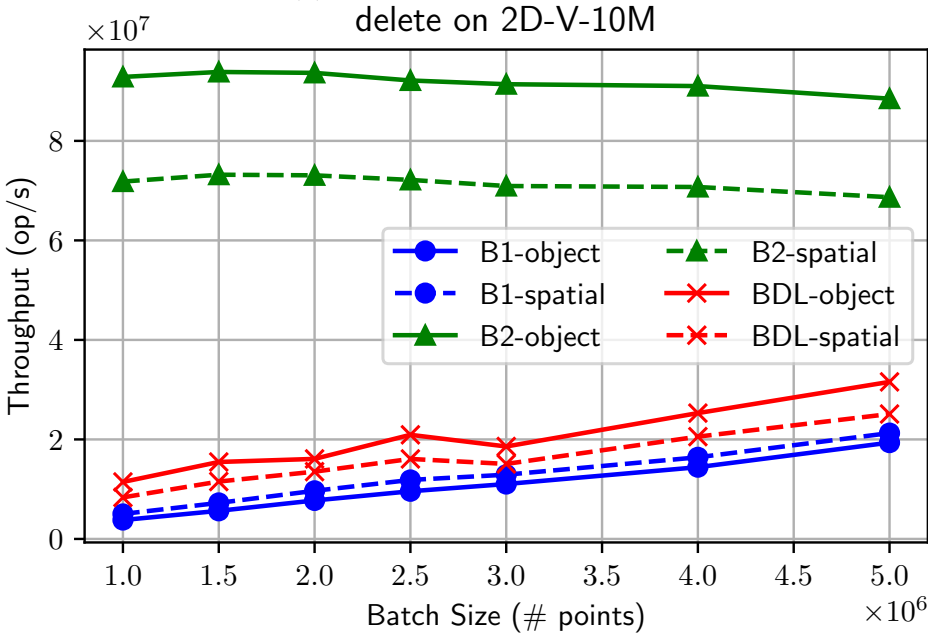
6.3.1 Scalability

In this benchmark, we measure the time required to delete 10 batches each containing 10% of the points in the dataset from an initially full tree for each of our two baselines as well as the BDL-tree. The results using an object median splitting heuristic are shown in Table 6.3a and the results using a spatial median splitting heuristic are shown in Table 6.3b. Figure 6-1c shows the scalability of the throughput on the 10M point 7D **Uniform** dataset.

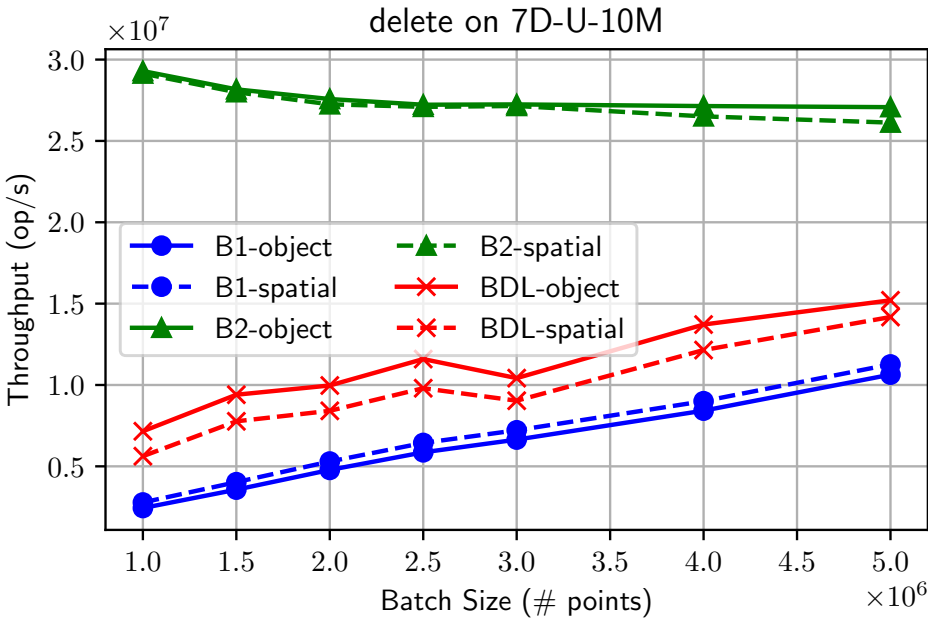
We observe that **B2** has vastly superior performance—it does almost no work other than tombstoning the deleted points so it is extremely efficient. Next, we see that **BDL** has the second-best performance, as it amortizes the rebuilding across the batches, rather than having to rebuild across the entire point set for every delete. Finally, **B1** has the worst performance as it rebuilds on every delete. With the object median splitting heuristic, **BDL** achieves parallel speedup of up to 33.1×, with an average speedup of 28.5×.



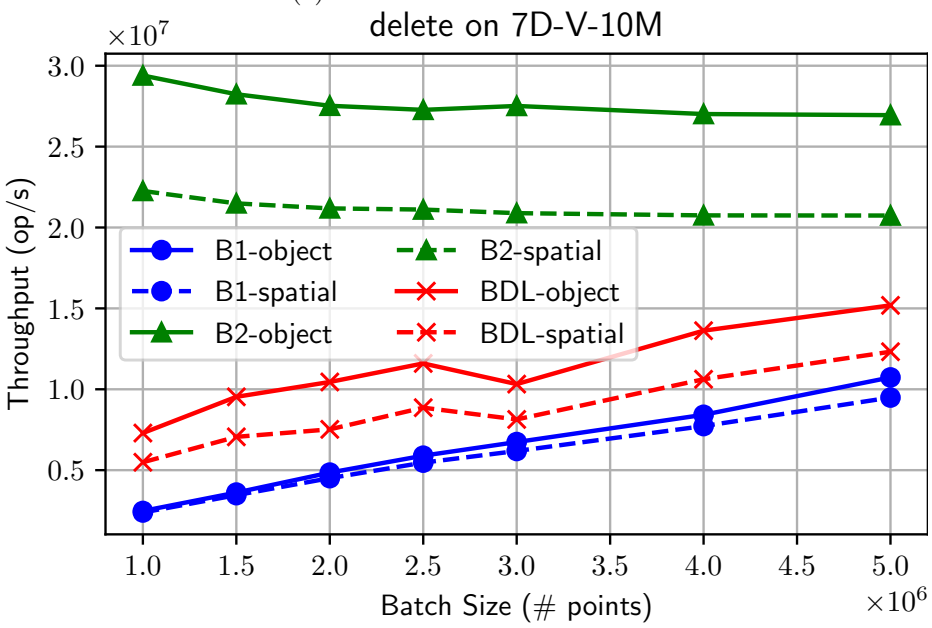
(a) Deletions on 2D-U-10M.



(b) Deletions on 2D-V-10M.



(c) Deletions on 7D-U-10M.



(d) Deletions on 7D-V-10M.

Figure 6-3: Plot of throughput (operations per second) of batch deletions vs. batch size for the 2D-U-10M, 2D-V-10M, 7D-U-10M, and 7D-V-10M datasets.

	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	114.0s	1.1s	29.2s	2.7s (43.0x)	0.1s (10.4x)	0.9s (33.0x)
2D-V-10M	114.2s	1.1s	29.0s	2.7s (43.0x)	0.1s (10.5x)	0.9s (33.1x)
5D-U-10M	130.2s	2.2s	40.1s	3.5s (37.2x)	0.2s (9.1x)	1.4s (28.1x)
5D-V-10M	127.0s	2.2s	39.3s	3.5s (36.8x)	0.2s (9.1x)	1.4s (27.8x)
7D-U-10M	146.7s	3.0s	37.3s	4.1s (36.0x)	0.3s (8.9x)	1.4s (27.0x)
7D-V-10M	144.4s	3.0s	36.9s	4.1s (35.6x)	0.3s (8.9x)	1.4s (27.2x)
10D-H-1M	24.7s	0.2s	21.9s	1.0s (24.7x)	0.0s (5.6x)	0.8s (28.5x)
16D-C-4M	74.8s	2.9s	33.1s	2.7s (27.4x)	0.3s (8.6x)	1.4s (23.6x)
3D-C-321M	–	–	–	–	–	17.3s

(a) Object median.

	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	70.2s	1.3s	30.2s	1.8s (38.8x)	0.1s (11.8x)	1.1s (26.3x)
2D-V-10M	71.7s	1.5s	30.7s	2.0s (36.0x)	0.1s (10.7x)	1.2s (25.4x)
5D-U-10M	82.9s	2.3s	29.2s	2.9s (28.9x)	0.2s (9.6x)	1.2s (23.9x)
5D-V-10M	85.9s	2.9s	29.6s	3.4s (25.4x)	0.3s (9.0x)	1.3s (22.2x)
7D-U-10M	97.1s	3.2s	38.2s	3.6s (27.1x)	0.3s (9.4x)	1.8s (21.4x)
7D-V-10M	100.8s	3.9s	38.3s	4.2s (24.0x)	0.4s (8.8x)	1.8s (20.9x)
10D-H-1M	27.6s	0.4s	22.1s	1.1s (25.1x)	0.1s (4.8x)	0.8s (26.2x)
16D-C-4M	60.5s	4.0s	32.5s	3.3s (18.3x)	0.5s (8.8x)	1.7s (19.5x)
3D-C-321M	–	–	–	–	–	–

(b) Spatial median.

Table 6.3: Batch deletion times (seconds) for a single thread (1) and 36 cores with hyper-threading (36h). The self-relative speedup for each implementation and dataset is shown in parentheses. We delete batches of 10% of each dataset, starting from a full tree until the entire dataset has been deleted.

6.3.2 Batch Size

In this benchmark, we measure the performance of our batch deletion implementation as the size of the batched update varies from 1M points to 5M points. We provide plots of the results for the 2D **Uniform**, 2D **VisualVar**, 7D **Uniform**, and 7D **VisualVar** datasets in Figures 6-3a 6-3b, 6-3c, 6-3d, respectively. We note again that **B2** consistently has the highest throughput, with **BDL** in second and **B1** with the lowest throughput. Furthermore, the throughput of **B1** and **BDL** increases as the batch size increases; this is true for the same reasons as with insertions. For **B2**, we observe consistent throughput across batch sizes.

6.4 Data-Parallel k-NN

In this benchmark, we measure the performance and scalability of our k-NN implementation as compared to the baselines. We split this into three separate experiments.

6.4.1 Scalability

In this experiment, we measure the scalability of the k-NN operation after constructing each data structure over the entire dataset (in a single batch). The results using an object median splitting heuristic are shown in Table 6.4a and the results using a spatial median splitting heuristic are shown in Table 6.4b. Figure 6-1d shows the scalability of the throughput on the 10M point 7D **Uniform** dataset. With the object median heuristic, **BDL** achieves a parallel speedup of up to 46.1×, with an average speedup of 40.0×.

The results show that **B1** and **B2** have similar performance (**B2** is slightly faster due to implementation differences). Furthermore, they are both faster than BDL-tree. This is to be expected, because the k-NN operation is performed directly over the tree after it is constructed over the entire dataset in a single batch. Thus, both baselines will consist of fully balanced trees and will be able to perform very efficient k-NN queries. On the other hand, **BDL** consists of a set of balanced trees, which

	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	34.9s	11.9s	64.5s	0.6s (57.2x)	0.3s (40.3x)	1.5s (43.1x)
2D-V-10M	37.2s	12.7s	67.0s	0.7s (57.0x)	0.3s (40.7x)	1.5s (43.7x)
5D-U-10M	302.3s	178.0s	339.4s	6.3s (47.6x)	3.5s (51.4x)	8.6s (39.4x)
5D-V-10M	109.5s	64.1s	145.8s	2.1s (51.1x)	1.2s (52.7x)	3.2s (46.1x)
7D-U-10M	1520.0s	1239.4s	1621.6s	44.3s (34.3x)	34.1s (36.4x)	52.9s (30.7x)
7D-V-10M	133.6s	86.3s	173.1s	2.8s (48.1x)	1.7s (50.7x)	4.0s (43.8x)
10D-H-1M	5.3s	5.5s	11.9s	0.1s (54.5x)	0.1s (50.5x)	0.3s (45.8x)
16D-C-4M	464.4s	612.2s	468.3s	16.5s (28.1x)	16.1s (38.1x)	17.3s (27.1x)
3D-C-321M	–	–	–	–	–	15.6s

(a) Object median.

	1			36h		
	B1	B2	BDL	B1	B2	BDL
2D-U-10M	33.6s	13.0s	69.9s	0.6s (55.6x)	0.3s (40.0x)	1.6s (44.9x)
2D-V-10M	35.7s	13.5s	72.7s	0.6s (56.8x)	0.3s (39.5x)	1.6s (44.9x)
5D-U-10M	348.8s	216.7s	503.1s	6.7s (52.2x)	4.0s (54.1x)	9.6s (52.2x)
5D-V-10M	103.1s	65.2s	174.2s	2.0s (52.8x)	1.4s (45.4x)	3.5s (49.5x)
7D-U-10M	2142.0s	1494.0s	2804.0s	45.5s (47.1x)	36.7s (40.7x)	57.3s (48.9x)
7D-V-10M	116.9s	81.3s	203.1s	2.3s (50.5x)	1.7s (48.0x)	4.1s (49.3x)
10D-H-1M	5.7s	4.7s	15.5s	0.1s (54.6x)	0.1s (45.2x)	0.3s (47.3x)
16D-C-4M	606.7s	557.0s	623.4s	19.9s (30.5x)	11.9s (47.0x)	20.4s (30.5x)
3D-C-321M	–	–	–	–	–	17.4s

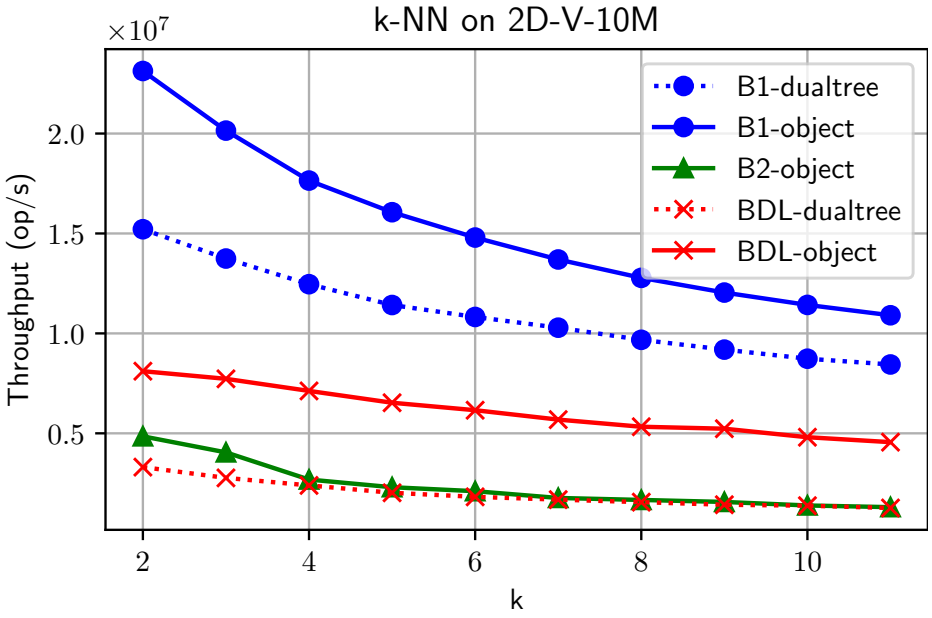
(b) Spatial median.

Table 6.4: k-NN times (seconds) for a single thread (1) and 36 cores with hyper-threading (36h). The self-relative speedup for each implementation and dataset is shown in parentheses. We use 100% of dataset except for 3D-C-321M, which was run with 10% of the dataset because the full k-NN results could not fit in memory.

adds overhead to the k-NN operation, as it must be performed separately on each of these individual trees. However, as the next two benchmarks show, **BDL** provides superior performance in the case of a mixed set of dynamic batch inserts and deletes interspersed with k-NN queries.

6.4.2 Effect of Varying k

In this experiment, we benchmark the throughput of the k-NN operation on 36 cores with hyper-threading as k varies from 2 to 11. For all three trees, we perform the

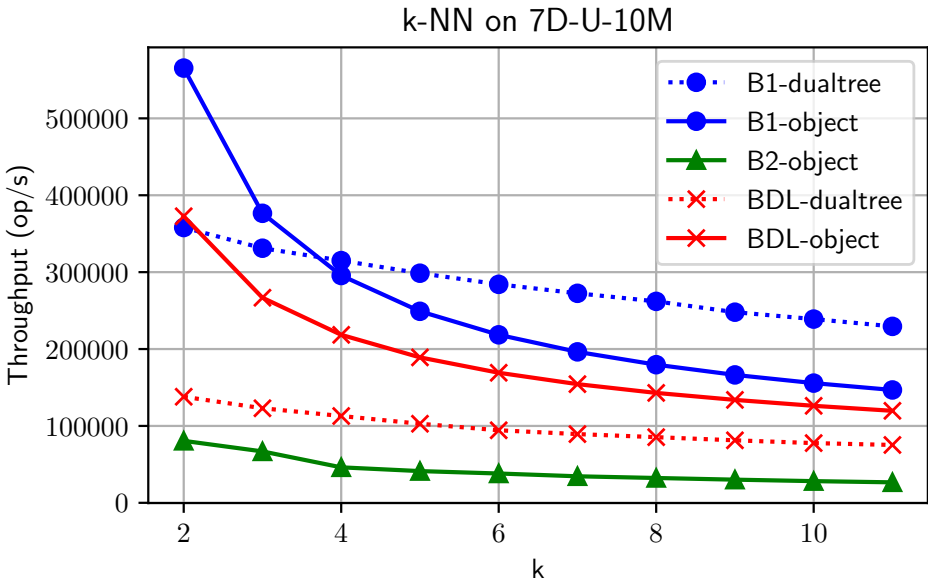


(a) k-NN on 2D-V-10M.

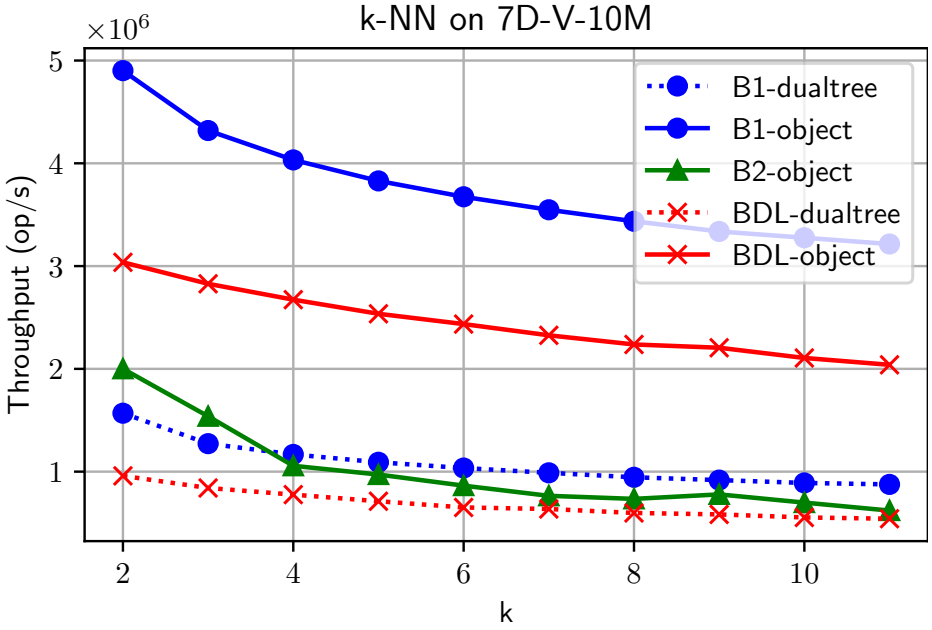
k-NN operation after building the tree from a set of batch insertions, with a batch size of 5% of the dataset, until the entire dataset is inserted. The results are shown for the 2D **VisualVar** dataset in Figure 6-4a, for the 7D **Uniform** dataset in Figure 6-4b, and for the 7D **VisualVar** dataset in Figure 6-4c. In all three cases, we see that **B1** has the best k-NN performance, followed closely by **BDL**. **B2** has significantly worse performance—this is because the construction of the tree was performed with a set of batch insertions, rather than a single construction over the entire dataset, the tree ends up imbalanced and the k-NN query performance suffers.

6.4.3 Mixed Operations

In this final experiment, we measure the k-NN and overall performance of the trees as a mixed set of batch insertions, batch deletions, and batch k-NN queries are performed. In particular, we perform a set of 20 batch insertions, each consisting of 5% of the dataset, into the tree. After every 5 batch insertions, we perform a k-NN query with $k = 5$ (over the entire dataset) to measure the current query performance. Then, we perform a set of 15 batch deletes, each consisting of a random 5% of the dataset (with no repeats). After every 5 batch deletes, we again perform a k-NN ($k = 5$)



(b) k-NN on 7D-U-10M.



(c) k-NN on 7D-V-10M.

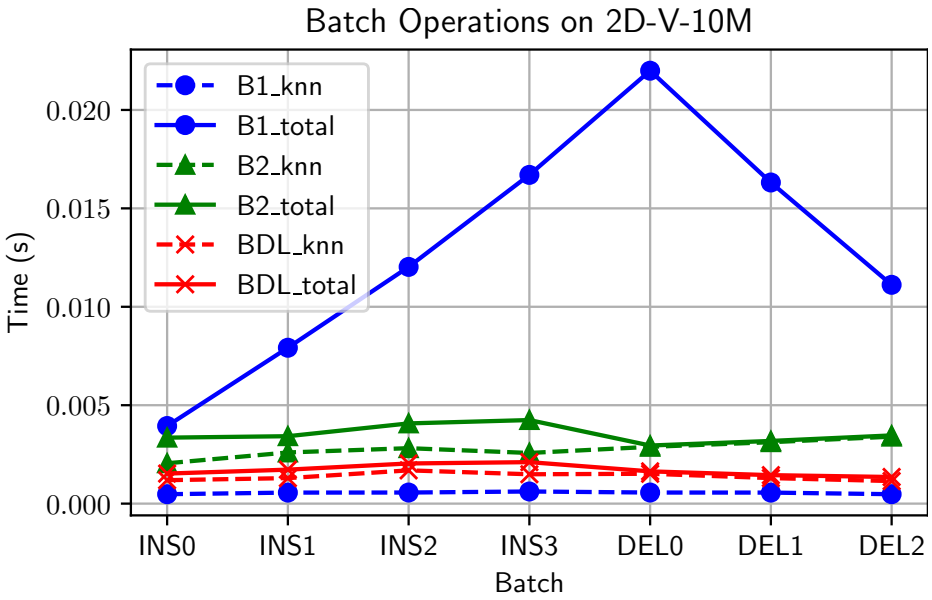
Figure 6-4: Plots of k-NN throughput (operations per second) vs. k using all 36h cores with hyperthreading, for the 2D-V-10M and 7D-U-10M, and 7D-V-10M datasets.

	1		36h	
	BDL-object	BDL-spatial	BDL-object	BDL-spatial
2D-U-10M	53.3s	47.0s	4.3s (12.3x)	2.7s (17.2x)
2D-V-10M	56.0s	53.3s	5.1s (11.1x)	2.5s (21.1x)
5D-U-10M	383.2s	346.2s	19.7s (19.4x)	17.3s (20.0x)
5D-V-10M	116.1s	100.9s	8.8s (13.1x)	8.8s (11.4x)
7D-U-10M	2365.8s	2172.9s	100.2s (23.6x)	96.4s (22.5x)
7D-V-10M	160.3s	132.8s	14.7s (10.9x)	17.1s (7.8x)
10D-H-1M	24.7s	19.2s	2.1s (12.0x)	1.7s (11.1x)
16D-C-4M	419.6s	467.0s	21.1s (19.9x)	28.5s (16.4x)

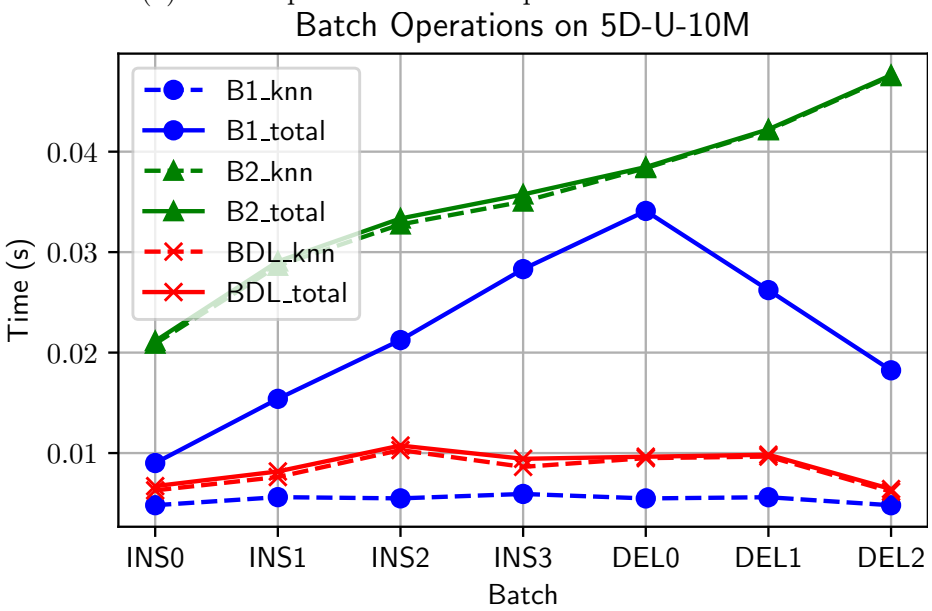
Table 6.5: Dual-Tree k-NN Scalability

query. Overall, there are 7 k-NN queries. We thus split the experiment into 7 distinct sections, demarcated by the k-NN queries. For each section (labeled as INS0, INS1, INS2, INS3, DEL0, DEL1, DEL2), we measure the k-NN runtime as well as the time for the 5 batched insertion/deletion operations. The results are shown for the 2D **VisualVar** dataset in Figure 6-5a and for the 5d **Uniform** dataset in Figure 6-5b (we observed similar results for other datasets). The x-axis shows the 7 sections, and the y-axis shows the time for each of these sections. There are two lines for each tree—a dashed line indicating just the k-NN times at each section, and a solid line indicating the total time for the batched update and k-NN.

In the **Uniform** case, we see that **B2** performs the worst overall, due almost entirely to its poor k-NN performance after batched updates. Note that the batched updates themselves contribute minimally to the total runtime of this baseline—they are very fast but cause significant imbalance in the tree structure, leading to degraded query performance. **B1** has the best raw k-NN query time, but its overall runtime is the second worst, as the batched updates are quite expensive (in order to maintain the balance that results in fast query times). Finally, **BDL** represents the best tradeoff between dynamic batch updates and k-NN performance. In particular, it has the best total runtime after every operation. The results are quite similar for the **VisualVar** case, except that we note that **B1** has the overall worst runtime, due to very expensive updates.



(a) Batch updates and k-NN queries on 2D-V-10M.



(b) Batch updates and k-NN queries on 7D-U-10M.

Figure 6-5: Plots of running times (seconds) of updates and queries vs. progressive batch updates on the tree using all 36 cores with hyper-threading, for the 2D-V-10M and 7D-U-10M datasets. "knn" represents the k-NN query performance after cumulative updates on the trees and "total" represents the combined running time of both updates and queries since the previous batch.

6.5 Dual-Tree k-NN

In addition to the data-parallel approach discussed above, we also implemented k-NN based on the dual-tree traversal proposed by March et al. [29]. While we found our dual-tree k-NN for **BDL** was faster on a single-thread due to more efficient pruning, it was not as scalable as other k-NN approaches and thus did not perform as well in parallel. This provides a promising direction to explore in the future, as we hope it could be possible to derive more parallelism with more work. Similarly to the other experiments, we ran both scalability experiments and tested the effect of varying k on this approach.

6.5.1 Scalability

We first tested the performance of the dual-tree k-NN in the single core and multicore settings. The results are shown in Table 6.5. Note that, when compared to the regular k-NN results, dual-tree k-NN is almost always faster in the single-core case, indicating that this approach does in fact yield benefits in terms of efficient pruning and traversal. This is a confirmation of the theoretical and experimental results from prior work on serial dual-tree k-NN traversals. However, it displays lower scalability at 36 cores and thus has worse multicore performance. This is due to worse pruning behavior, as the traversal of the tree in parallel is not able to share tightening radius bounds as efficiently and thus explores more of the tree than it does in the serial case.

6.5.2 Effect of Varying k

We also tested the effect of varying k with 36 hyperthreaded cores to observe the performance characteristics of the dual-tree k-NN implementation. The results for this experiment are seen in the "BDL-dualtree" line in the graphs in Figures 6-4a, 6-4b, 6-4c. We notice the same result as before — the dual-tree k-NN for **BDL** has a lower throughput than the data-parallel k-NN for **BDL** in the multicore setting. For this experiment, however, we also tested a dual-tree k-NN approach on **B1** — the results are seen in the same figures in the line labeled "B1-dualtree". We notice some

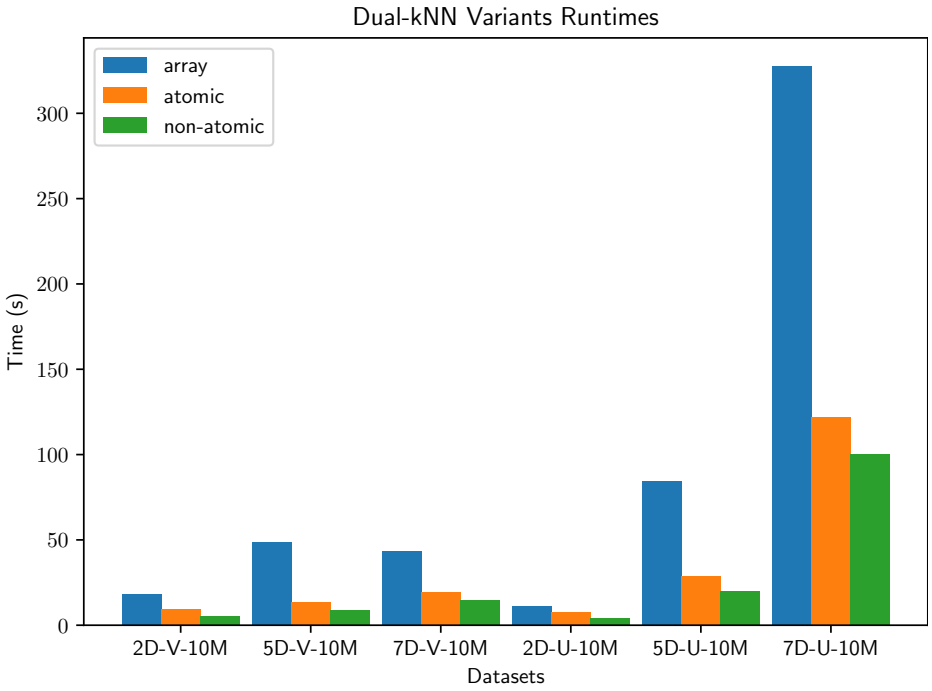


Figure 6-6: Plots of running times (seconds) of the three different dual-tree k-NN implementations on the BDL-tree on all 6 synthetic datasets using all 36 cores with hyper-threading.

promising results here – in particular, the “B1-dualtree" line in Figures 6-4a, 6-4b is very similar to the “B1-object" line, suggesting that the multicore performance of the dual-tree approach for a single tree is similar to the data-parallel k-NN performance. Future work could be focused on performance engineering this approach in order to achieve similar levels of parallelism to the data-parallel k-NN.

6.5.3 Different Implementations of Radius

As mentioned in Section 5.3, we tested several different implementations of the dual-tree k-NN when developing this approach. In order to compare these three variants, we ran a full k-NN ($k = 5$) with each of the implementations on each of the 6 synthetic datasets. The results are shown in Figure 6-6. It is clear from these results that the non-atomic radius implementation was consistently the best, as mentioned before. This is due to the poor cache performance induced by exploring multiple trees in parallel, the overheads of atomic updates, and the decrease in shared radius

estimates due to parallelism.

Chapter 7

Conclusion

We have presented the BDL-tree, a parallel batch-dynamic kd-tree which supports batched construction, insertions, deletions, and k-NN queries. We show that our data structure has strong theoretical work and depth bounds. Furthermore, our experiments show that the BDL-tree achieves good parallel speedup and presents a useful tradeoff between two baseline implementations. In particular, it delivers the best performance in a dynamic setting involving batched updates to the underlying dataset interspersed with k-NN queries. Future work includes exploring further splitting heuristics, vectorized operations and other performance engineering, and supporting more query types.

Bibliography

[1] Chem dataset. `https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures`.

[2] Ht dataset. `https://archive.ics.uci.edu/ml/datasets/Gas+sensors+for+home+activity+monitoring`.

[3] Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, page 237–245, 2003.

[4] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, page 429–440, 2016.

[5] Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. Cache-oblivious data structures. In *Handbook of Data Structures and Applications*, pages 545–565. Chapman and Hall/CRC, 2018.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 399, 2000.

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

[8] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[9] Jon Louis Bentley and James B Saxe. Decomposable searching problems I. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[10] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104, 2006.

[11] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, page 507–509, 2020.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[15] Ryan R. Curtin. Faster dual-tree traversal for nearest neighbor search. In *Proceedings of the 8th International Conference on Similarity Search and Applications*, page 77–89, 2015.

[16] Ryan R. Curtin, William B. March, Parikshit Ram, David V. Anderson, Alexander G. Gray, and Charles L. Isbell. Tree-independent dual-tree algorithms. In *Proceedings of the 30th International Conference on International Conference on Machine Learning*, page 1435–1443, 2013.

[17] Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

[18] Laxman Dhulipala, Quanquan C Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 129–143, 2021.

[19] Magdalen Dobson and Guy E. Blelloch. Parallel nearest neighbors in low dimensions with batch updates. *CoRR*, abs/2111.04182, 2021.

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

[21] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

[22] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[23] Junhao Gan and Yufei Tao. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 519–530, 2015.

[24] Alexander G. Gray and Andrew W. Moore. 'N-body' problems in statistical learning. In *Proceedings of the 13th International Conference on Neural Information Processing Systems*, page 500–506, 2000.

[25] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[26] Ramon Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai F Rulkov, and Irene Rodriguez-Lujan. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 157:169–176, 2016.

[27] Joseph JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[28] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*, pages 132–150, 2010.

[29] William B. March, Parikshit Ram, and Alexander G. Gray. Fast Euclidean minimum spanning tree: Algorithm, analysis, and applications. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 603–612, 2010.

[30] Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[31] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[32] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*, pages 46–65. Springer, 2003.

[33] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, 2007.

[34] Roberto Agostino Vitillo. Parallel log-structure merge tree for key-value stores. `https://github.com/vitillo/kvstore`.

[35] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. A parallel batch-dynamic data structure for the closest pair problem. In *37th International Symposium on Computational Geometry*, volume 189, pages 60:1–60:16, 2021.

[36] David Wehr and Rafael Radkowski. Parallel kd-tree construction on the GPU with an adaptive split and sort strategy. *Int. J. Parallel Program.*, 46(6):1139–1156, December 2018.

[37] Stefan Zellmann, Jürgen P. Schulze, and Ulrich Lang. Binned k-d tree construction for sparse volume data on multi-core and GPU systems. *IEEE Transactions on Visualization and Computer Graphics*, 27(3):1904–1915, 2021.