New Techniques for Parallelism and Concurrency in Nearest Neighbor Search

Magdalen Dobson Manohar

CMU-CS-25-100

May 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Phil Gibbons
Andy Pavlo
Matthijs Douze, Meta AI Research
Harsha Vardhan Simhadri, Microsoft Azure

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the National Science Foundation under grants CCF-1901381, CCF-1910030, CCF-1919223, and CCF-2119352, as well as the NSF Graduate Research Fellowship Program.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, any government or any other entity

Keywords: Nearest Neighbor Search, Information Retrieval, Range Search, Parallelism, Concurrency

To my husband, my parents, and my siblings.

Abstract

Nearest neighbor search in both high and low dimensions is an important problem in the field of computer science and beyond. In this thesis, we extend the capabilities of nearest neighbor search algorithms to meet modern demands, including support for billion-scale indices, parallelism and concurrency on machines with hundreds of cores, efficient updates, and extensions to related geometric problems such as range search. We progress towards this goal by introducing the zd-tree, a data structure for low-dimensional nearest neighbor search with provable guarantees on the work and span of search, build, and update, a scalable parallel build, and the ability to perform batch-dynamic updates in parallel. Building on the zd-tree, we also present the CLEANN-Tree (for Concurrent Linearizable Efficient Augmented Nearest Neighbor Search Tree), a generalization of the zd-tree which supports concurrent queries and updates utilizing versioned pointers and lock-free locks. In high dimensions, we introduce techniques to make existing nearest neighbor search algorithms lock-free, deterministic, and scalable to billion-size datasets. We apply these techniques to four existing graph-based nearest neighbor search algorithms in a library called ParlayANN. Building off our work in ParlayANN, we extend the search algorithms for graph-based nearest neighbor indices to the related but relatively under-studied problem of range search in high dimensions, making significant gains over a naive baseline.

Acknowledgments

First, I would like to thank my advisor Guy Blelloch, for countless hours of instruction and collaboration, and for teaching me almost everything I know about parallelism and about algorithms that are efficient in both theory and practice.

I would like to thank my thesis committee members: Phil Gibbons, Andy Pavlo, Matthjis Douze, and Harsha Vardhan Simhadri for all their support and advice in shaping this thesis. I would especially like to thank Harsha Vardhan Simhadri for the support he has given me in my career, and for introducing me to many of the problems I worked on both during my PhD and beyond.

I would also like to thank all the fantastic colleagues whom I collaborated with over the course of my PhD: Laxman Dhulipala, Yan Gu, Yihan Sun, Hao Wei, Ben Landrum, Zheqi Shen, and Taekseung Kim. I've gained so much as a researcher from all of your experience and insight.

A big thank you to my friends, both at CMU and elsewhere, for supporting me throughout my PhD. Thank you to Brian Zhang, Steven Jecmen, Dorian Chan, and Hugo Sadok for helping me stay sane through the pandemic years. Thanks to Jatin Arora for lots of fun outings and dinners. Thank you to Jason Zhang, Helen Jiang, Sudeep Dasari, and Varsha Venkat for lots of fun outings and afternoons spent watching football, even if I was never very good at fantasy. Thank you to Katherine Kosaian and Aditi Kabra for all the girl talk, especially the many, many hours on all things wedding planning. Thanks to Ahaan Rungta, Marisa Gaetz, and Chase Vogeli for our always entertaining weekly phone calls.

Thank you to my family: my parents Susan Cook and Ted Dobson, and my siblings Blue and Alcuin Dobson. Thank you for making every Christmas special, for weekly calls and Minecraft sessions, and for all your support over the past six years and all the time before. Finally, thank you to my husband Peter Manohar, who has been there for me every day through every up and down of this degree.

Contents

1 Introduction **1**
- 1.1 Low-Dimensional Nearest Neighbor Search . . . 2
  - 1.1.1 Contributions of This Thesis . . . 3
- 1.2 High-Dimensional Approximate Nearest Neighbor Search . . . 4
  - 1.2.1 Contributions of This Thesis . . . 7
  - 1.2.2 Outline . . . 8

2 Preliminaries **9**
- 2.1 Nearest Neighbor Search and Range Search . . . 9
- 2.2 Parallelism and Concurrency . . . 10

I Low-dimensional Nearest Neighbor Search **13**

3 Parallel Nearest Neighbors in Low Dimensions with Batch Updates **17**
- 3.1 Introduction . . . 17
  - 3.1.1 Preliminaries. . . . 19
  - 3.1.2 Related Work. . . . 20
- 3.2 Algorithm Design and Bounds . . . 22
  - 3.2.1 Theoretical Results. . . . 23
- 3.3 Implementation Details . . . 31
  - 3.3.1 Our Algorithms. . . . 31
  - 3.3.2 Other Implementations. . . . 32
- 3.4 Experiments . . . 33
  - 3.4.1 Experimental Setup. . . . 33
  - 3.4.2 Leaf vs. Root Based. . . . 34
  - 3.4.3 k-Nearest Neighbor Graphs. . . . 34
  - 3.4.4 Dynamic Queries . . . 34
  - 3.4.5 Dynamic Updates. . . . 35
  - 3.4.6 Code availability. . . . 35

4 CLEANN: Lock-Free Augmented Trees for Exact k-Nearest Neighbor Search **39**
- 4.1 Introduction . . . 39
- 4.2 Background and Related Work . . . 41

4.2.1 KNN structures . . . 41
4.2.2 Concurrent KNN Datastructures . . . 45
4.2.3 Optimistic Locking . . . 45
4.2.4 Lock-Freedom and Snapshotting with Verlib . . . 46
4.3 The CLEANN-tree . . . 47
4.3.1 Path-Copying . . . 49
4.3.2 Hand-Over-Hand Locking . . . 50
4.4 Experiments . . . 51
4.4.1 Implementation . . . 51
4.4.2 Experimental Setup . . . 51
4.4.3 Experimental Results . . . 53
4.5 Applications of Shallow Augmentation . . . 58
4.6 Conclusion . . . 59

II High-dimensional Nearest Neighbor Search 61

5 ParlayANN: Scalable and Deterministic Parallel Graph-Based ANNS Algorithms 65
5.1 Introduction . . . 65
5.2 General Techniques for Graph-Based ANNS Algorithms . . . 69
5.2.1 Incremental Algorithms . . . 70
5.2.2 Clustering-Based Algorithms . . . 72
5.3 ParlayANN Algorithms . . . 73
5.3.1 DiskANN . . . 73
5.3.2 HNSW . . . 74
5.3.3 HCNNG . . . 74
5.3.4 PyNNDescent . . . 75
5.3.5 Search and Layout Optimizations . . . 75
5.4 Experimental Evaluations . . . 76
5.4.1 Experimental Setup . . . 76
5.4.2 Partition-Based Competitors . . . 78
5.4.3 Comparison with ANN Benchmarks . . . 79
5.4.4 Parallelism and Scalability . . . 80
5.4.5 Full Billion-Scale and Hundred-Million Results . . . 81
5.4.6 Dataset Size Scaling . . . 82
5.4.7 Conclusions from Experiments . . . 83
5.5 Related Work . . . 83

6 Range Retrieval with Graph-Based Indices 87
6.1 Introduction . . . 87
6.1.1 Related Work . . . 89
6.2 Characteristics of Range Search Datasets . . . 90
6.2.1 Density of Matches by Dataset . . . 90
6.2.2 Frequency Distribution of Matches . . . 92

6.3 Algorithms for Range Search on Graph-Based ANNS Indices 93
6.3.1 Data Structure and Baseline Algorithm 93
6.3.2 Improving Range Search for Queries with Many Results 94
6.3.3 Improving Range Search for Queries with No Results 95
6.4 Experimental Results 98
6.5 Investigation of Range Search Metadata 105
6.6 Additional Data on Early Stopping Metrics 113

III Conclusion **117**

7 Conclusion **121**
7.0.1 Open Problems 122

Bibliography **125**

List of Figures

3.1 A figure showing the work (threads $\times$ time) performed by various nearest neighbor algorithms as the number of threads increases. The k-nearest neighbor graph was computed on 10 million points from a random dataset within a 3D cube. Ideally the line for a particular algorithm would be both low (small total work) and straight (indicating more threads does not change the total work). 19
3.2 Figures aiding with the proofs in Section 3.2.1. 26
3.3 A bar chart showing the performance of our three k-nearest neighbor algorithms on different datasets. All datasets are 10 million points, and times reported are for $k = 1$. . . . 31
3.4 Statistics related to non-dynamic queries. Unless otherwise stated, the size of the dataset is 10 million, the number of nearest neighbors $k = 1$, experiments were performed on 144 threads on a 72-core Dell R930, and data points are drawn randomly from a 3D cube. 36
3.5 Time required per point as the number of points in the batch increases. Updates performed on a dataset of 10 million random points inside a 2D cube. 37
3.6 Statistics related to dynamic queries. Unless otherwise stated, the size of the dataset is 10 million, 10 million dynamic queries were performed, the number of nearest neighbors $k = 1$, experiments were performed on 144 threads on a 72-core Dell R930, and data points are drawn randomly from a 3D cube. 37

4.1 An illustration of searching for the 3-nearest neighbors of the query point q, which is marked in red, with its nearest neighbors marked in blue. Points g_1, g_2, g_3 denote the initial guess for q's nearest neighbors. Boxes B'_1, B'_2 denote bounding boxes of tree nodes, while the boxes drawn with a dotted line show the exact bounding boxes of the point set in the node. Note that all three true nearest neighbors of q are within radius r of q, and that the ball of radius r overlaps with the two bounding boxes containing the remaining neighbors of q. 49
4.2 Visualizations of some of the datasets used in the experiments. From left to right: a visualization of the Plummer model [78], the Lucy statue from the Stanford 3D Scanning Repository, and the Thai statue from the Stanford 3D Scanning Repository [15]. 52
4.3 Figures measuring throughput and/or parallel speedup as the number of threads ranges from 1 to 144. Unless otherwise mentioned, the experiment is conducted using nearest neighbor search with a 50% update-to-query ratio. The size given in the caption refers to the steady-state size of the CLEANN-tree used. 54
4.4 Experiments measuring the effect of update rate on throughput. Both experiments run with 144 threads. 55

4.5 Experiments measuring the effect of dimension and data structure size on throughput. Figures run with 144 threads. 56
4.6 Experiments measuring throughput in oversubscription on the 3DinCube dataset with a small working set, and the 3DinCube dataset without modification. 56
4.7 Experiments measuring the effect of k, the number of nearest neighbors, on the overall throughput. Experiment conducted on 144 threads on the 3DinCube dataset with 10 million points and a 50% mix of updates and queries. 56

5.1 **Scalability of original and our new implementations of four ANNS algorithms on various number of threads. Within each subfigure, all numbers are *speedup numbers relative to the original implementation on one thread*. Higher is better.** Results were tested on a machine with 96 cores using dataset BIGANN-1M (10^6 points). "96h": 48 cores with hyperthreads. The two implementations in the same subfigure always use the same parameters and give similar query quality (recall-QPS curve). 68
5.2 An example of ANNS graph and a greedy search. The blue arrows represent directed edges in the proximity graph, which is a mix of long and short edges. Below is an example of NNS query on point q (red point). The algorithm starts with adding the starting point A as the only point in the beam $\mathcal{L}$, and then in every step, finds the closest unprocessed point in $\mathcal{L}$ (to q) and adds its out-neighbors. Once $|\mathcal{L}|$ goes beyond L, it is refined to keep only the L nearest points. A set $\mathcal{V}$ is maintained for all processed vertices. When all vertices in $\mathcal{L}$ are also in $\mathcal{V}$, the algorithm finishes. 69
5.3 Build time (hours), QPS, recall, and distance comparisons for all algorithms on billion-size datasets. 79
5.4 QPS-recall curves on all 100-million size datasets. The first row shows the overall QPS/recall curve, while the second row zooms into a higher-recall regime. The build times are given in Tab. 5.2 80
5.5 QPS on a single thread on BIGANN-1M. Shown to compare with ANN-benchmarks. 80
5.6 Figures showing the effect of dataset size on different metrics using the MSSPACEVdataset. 81

6.1 A preview of our experimental results on three datasets: from left to right, OpenAI-1M, BIGANN-10M, and SSNPP-100M. The solid line shows the beam search baseline, while the dotted and dashed lines show our two new algorithms. Datasets and algorithms are described in detail in Sections 6.2 and 6.3, respectively. 89
6.2 Plots of radius versus percent captured for each dataset: Euclidean datasets on the left and inner product datasets on the right. Radius is normalized to enable displaying multiple datasets on one plot. Note that inner product values can be less than zero, so negative ranges of radius are present on the right-hand figure. . 91
6.3 Histograms of early stopping metrics for selected metrics and datasets. All metric values were taken at step 20 of a beam search with beam 100. Queries are separated by color based on the number of range results. 97
6.4 Average precision vs QPS for eight datasets and three range search algorithms. For GIST-1M, the lines for doubling search and greedy search are very short due to even the smallest of initial beam sizes producing recall in the .999 range. . . . 99

6.5 Average precision vs QPS showing SSNPP, BIGANN, and Wikipedia at different scales. 100
6.6 Figures breaking down the cost in seconds (single threaded time) of each type of search for three datasets for selected average precision. The label at the top of each column indicates the beam width of the initial search. Each collection from left to right shows: the beam search baseline, beam search followed by greedy search, beam search with early stopping followed by greedy search, beam search followed by doubling beam search, and beam search with early stopping followed by doubling beam search. Note that in some cases, the time spent on the second phase of search is so short that it is not visible. GIST-1M has only one selected recall, as every search setting yields .999 recall within a .001 tolerance; the beam search baseline is not present for GIST, since it cannot achieve the desired recall (see Figure 6.4d). 101
6.7 Average precision vs QPS for all nine datasets using greedy search, comparing use of early stopping versus without early stopping. 102
6.8 Average precision vs QPS for all nine datasets using doubling search, comparing use of early stopping versus without early stopping. 103
6.9 Early stopping metrics for BIGANN-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius. . 114
6.10 Early stopping metrics for DEEP-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius. . 114
6.11 Early stopping metrics for Wikipedia-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius. . 115
6.12 Early stopping metrics for SSNPP-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius. . 115

List of Tables

4.1 Characteristics of the datasets used in our experiments. The first two are synthetic (uniform in 3D cube, and Plummer distribution [21] in 3D), and the size can be varied. The last two are from Stanford 3D Scanning Repository [15]. We show the logarithm of the maximum distance between two points divided by the minimum distance with respect to the 10 million size versions of the synthetic datasets. 52

5.2 Build times (hours) on hundred million scale datasets. 77

5.1 Parameters chosen for each dataset. For DiskANN, HCNNG, and pyNNDescent, α denotes the pruning parameter. For DiskANN, R denotes degree bound and L is the beam size. For HNSW, m denotes the degree bound and efc is the efConstruction. For HCNNG and pyNNDescent, Ls denotes leaf size and T denotes number of cluster trees. For HCNNG, s denotes MST degree. For pyNNDescent, K is the degree bound. For FAISS, the first string is the type of vector transform used for building the index, the second denotes the IVF index type, and the third indicates the PQ compression for the queries. For FALCONN, l is the number of hash tables, and rot is the number of rotations. 78

6.1 Descriptions of each dataset used in our experiments, along with the radius used for range search. Every dataset is publicly available [26, 156]. The SSNPP dataset is a range search benchmark, while the other datasets were originally published as top-k benchmarks. Unless otherwise indicated, the reference in the "Source" column covers all attributes of the dataset. 91

6.2 Table showing the distribution of result sizes for each dataset. Note that not all datasets have the same number of query points; see Table 6.1 for the number of queries and corresponding radius for each dataset. No data point had more than 10^5 results. 92

Chapter 1

Introduction

The use of machine learning and large language models has enjoyed a meteoric rise in the field of computer science over the past decade. Machine learning models convert objects such as text, audio clips, images, and videos to *metric embeddings*, hundred- to thousand-dimensional vectors where semantically similar objects are mapped to vectors close together in space. A common sub-step in machine learning assisted algorithms is thus to retrieve nearby vectors of a query vector; these applications include search and ads recommendations and information retrieval [169], as well as large language models (LLMs) such as ChatGPT [10], and other applications combining LLMs and vector search [9, 57, 160, 165]. This problem is known as *nearest neighbor search*, and solving it exactly often requires exhaustive scan of the database involved. Thus, applications instead use approximate nearest neighbor search, and new efficient, approximate algorithms for nearest neighbor search have proliferated in both academia and industrial use, although some key algorithmic questions remain unsolved. The largest publicly known indices contain around 1 trillion vectors, while many applications require indices in the billions of vectors, and even more smaller-scale applications use indices with millions of vectors.

Serving the demands of these large indices will necessarily require parallel processing for both construction and query; for example, a single CPU would take tens to hundreds of days to construct a billion-size index, which would likely be out-dated and unusable by the time its processing was complete. Luckily, the rise of so-called "big data" has been accompanied with a significant increase in availability parallel hardware; now, even personal laptops and smartphones have multiple CPU cores, and commodity bare-metal and virtual machines have up to hundreds of cores. Hence, theoretically and practically efficient multi-threaded algorithms are a necessity for constructing, querying, and updating large vector databases. Even data structures for vectors in smaller dimensions, which have applications such as ray tracing, collision detection, surface reconstruction, and density peaks clustering [66, 132, 139, 141] in fields from computer graphics [24, 82] to particle physics [151], must be able to run efficiently on machines with hundreds of cores and in fully concurrent settings.

In this thesis, we take steps to improve the performance and capabilities of the state-of-the-art algorithms in nearest neighbor search for today's large datasets and multicore machines, chiefly with respect to parallelism and concurrency.

In the low-dimensional data structures where d is a small constant in the range of 2-5, kd-trees and space-filling curves [32, 46, 55, 69] provide excellent sequential performance, but most

existing data structures only scale up to 8-16 threads, and very few of them support even single-threaded updates to the data structure. Furthermore, there was not a single linearizable, lock-free concurrent data structure that supported insertions, deletions, and k-nearest neighbor queries for arbitrary k [60, 102]. In this thesis, we introduce the *zd-tree* [46], a kd-tree supporting a parallel build and parallel batch-dynamic updates, with theoretical guarantees under reasonable assumptions, and the *CLEANN-tree* (Concurrent Linearizable Efficient Augmented Nearest Neighbor Tree), a kd-tree supporting fully linearizable and lock-free queries and updates.

In high dimensional nearest neighbor search, existing data structures, typically *graph-based* nearest neighbor data structures [84, 126, 161, 169], provide an excellent trade-off between accuracy and throughput for queries. However, they also tend to build and update in parallel with heavy use of locks, leading to contention and nondeterminism in the output. To solve this issue, we introduce ParlayANN [127], a library of scalable and deterministic nearest neighbor search algorithms. In ParlayANN, we develop new techniques in parallelism for nearest neighbor search that allow us to take four state-of-the-art approximate nearest neighbor search (ANNS) algorithms—DiskANN, HNSW, HCNNG, and pyNNDescent [126, 129, 135, 161]—and build their corresponding data structures in parallel without the use of locks, and thus with full determinism. ParlayANN also enables a fine-grained comparison between the four algorithms, as they are re-implemented using the same search routines and as much shared code as possible.

In addition to high-dimensional nearest neighbor search, *range search*—that is, finding all points in the ball of radius r around a query point—is an important subroutine in applications such as duplicate detection, plagiarism checking, and facial recognition [130], as a subroutine in search applications [163], and as a subroutine in tasks such as graph clustering [95, 122]. Despite these applications and the huge quantity of work on top-k search, range search is relatively understudied, with only a handful of papers even addressing the question and no work dedicated to improving range search on graph-based indices. In this thesis, we present a set of algorithms specialized for range search on ANNS graphs which are capable of dynamically adapting to the size of the query, extending the search for queries with thousands of results, and terminating early for queries with no results. Our algorithms generate significant speedup over a naive baseline and set the standard for future benchmarks.

The work in this thesis will support the following thesis statement:

> *With new techniques in parallelism and concurrency, data structures for nearest neighbor search can be made work-efficient, dynamic in both fully concurrent and parallel batch-dynamic settings, and effective at related geometric tasks such as range searching.*

1.1 Low-Dimensional Nearest Neighbor Search

Computing nearest neighbors in low dimensions in Euclidean space (where the dimension d is a small constant, usually 2 or 3), is an important and well-researched topic in computer science. Nearest neighbor data structures have been used for decades as a key component for solving problems such as ray tracing, image reconstruction, and collision detection [66, 132, 139, 141, 151]. They have been used to develop fast parallel algorithms for problems such as Euclidean minimum spanning tree and hierarchical clustering [172]. Furthermore, applications such as

motion planning [102] and density peaks clustering [101] require *dynamic* k-nearest neighbor (KNN) data structures, leading to work in developing KNN data structures with fast parallel batch updates [171, 180]. Beyond batch-dynamic updates, there is a limited amount of work on concurrent KNN data structures, which are capable of supporting insertions, deletions, and queries at all times [60, 102].

Data Structures. By far the most common method of computing nearest neighbors is via a kd-tree, a tree which keeps the entire bounding box of the point set at its root, and whose children represent progressively smaller bounding boxes. Kd-trees have many applications in point-based graphics, and have been the data structure of choice for many graphics practitioners [139], even though other methods have better worst-case guarantees. One of the best kd-tree implementations is Arya et al's [33], which has been used widely by researchers [66]. Another commonly used library of kd-trees is that of the Computational Geometry Algorithms Library [164]. Although in principle kd-trees should be able to support dynamic updates, we know of only one library which efficiently supports them [171], and few interesting theoretical bounds for the problem in low dimensions. When considering parallelism and updates together one should be interested in batches of updates that can be processed in parallel. In an alternative approach, a space-filling curve known as the Morton ordering, z-ordering, or Lebesgue ordering, is useful in computing nearest neighbors. Given a sequence of points sorted using their Morton ordering, it is possible to recursively recursively divide and search the input while searching for nearest neighbors. Two nearest neighbor algorithms that make use of Morton ordering are Connor and Kumar's k-nearest neighbor graph algorithm [70] and Chan's minimalist nearest neighbor algorithm [55].

Parallelism in nearest neighbor search. Given the ubiquity of parallel computing and the importance of computing nearest neighbors, there has been a surprisingly small amount of research on nearest neighbors in a parallel setting. Open-source geometric algorithms libraries such as CGAL [25] provide kd-trees with support for parallel queries and some further utilization of parallelism, such as building the kd-tree in parallel, but we were unable to find a commonly used geometric algorithms library that is also highly optimized for parallelism. One parallel library which focuses on computing nearest neighbors in parallel is Connor and Kumar's STANN library [70], which computes nearest neighbors in a thread-safe manner using the Morton ordering as well as doing parallel preprocessing.

Concurrent KNN Data Structures. The only previous work we know of on concurrent k-nearest neighbor structures has either supported only non-linearizable neighbor queries [102], or has been limited to finding the single nearest neighbor of a point [60].

1.1.1 Contributions of This Thesis

Zd-tree. We present a set of parallel algorithms for computing exact k-nearest neighbors in low dimensions. Many k-nearest neighbor algorithms use either a kd-tree or the Morton ordering of the point set; our algorithms combine these approaches using a data structure we call the *zd-tree*. We show that this combination is both theoretically efficient under common assumptions, and

fast in practice. For point sets of size n with bounded expansion constant and bounded ratio (two common assumptions defined in Chapter 3, Section 3.1.1), the zd-tree can be built in $O(n)$ work with $O(n^\epsilon)$ span for constant $\epsilon < 1$, and searching for the k-nearest neighbors of a point takes expected $O(k \log k)$ time. We benchmark our k-nearest neighbor algorithms against existing parallel k-nearest neighbor algorithms, showing that our implementations are generally faster than the state of the art as well as achieving 75x speedup on 144 hyperthreads. Furthermore, the zd-tree supports parallel batch-dynamic insertions and deletions; to our knowledge, it is the first k-nearest neighbor data structure to support such updates. On point sets with bounded expansion constant and bounded ratio, a batch-dynamic update of size k requires $O(k \log n/k)$ work with $O(k^\epsilon + \text{polylog}(n))$ span. This work appeared in ALENEX 2022 [46].

CLEANN-Tree. The CLEANN-Tree (for Concurrent Linearizable Augmented Efficient Nearest Neighbor Tree) is a linearizable lock-free data structure for k-nearest neighbor searching. In particular it maintains a set of points P in d dimensions under insertion and deletion, while supporting queries that given a point p and integer k return the k nearest points of p in P. It is the first such concurrent structure—two previous structures were respectively not linearizable or only supported $k = 1$ [60, 102]. Furthermore, the CLEANN-Tree supports augmentation on the internal nodes of their spatial decomposition tree, which significantly improves performance over the previous structures. The problem is challenging in the linearizable lock-free setting because queries can require looking at large parts of the structure, which might be changing, and updating the augmented values can require updating a non-constant number of nodes from the leaf to the root. We develop new approaches for the augmentation and leverage recent work on lock-free locks and snapshotting. We have implemented a library based on the approach and present preliminary experimental results. This work is currently under submission.

1.2 High-Dimensional Approximate Nearest Neighbor Search

The adoption of deep learning methods over the past decade have led to high-dimensional vector representations of objects a.k.a. embeddings becoming widely used. These representations are typically obtained by training deep neural networks. As a result, machine learning datasets usually contain billions of vectors representing embeddings of users, documents, search queries, images, among many other kinds of objects. These embeddings can span hundreds to thousands of dimensions. The algorithms producing these embeddings are trained so that similar objects have "close" embeddings (e.g., in L_2 distance). As a result, an important problem is to find the nearest and thus most similar set of k objects for a query point in the embedding space $\mathbb{R}^d$.

This high-dimensional nearest neighbor search problem is notoriously hard to solve exactly in high-dimensional spaces [44]. Since solutions for most real-world applications can tolerate small errors, most deployments focus on the *approximate nearest neighbor search (ANNS)* problem, which has been widely applied as a core subroutine for search recommendations, machine learning, and information retrieval [169], as well as large language models (LLMs) used in ChatGPT [10] and other applications combining LLMs and vector search [9, 57, 160, 165]. Considering that embeddings and similarity search are at the heart of these and many other modern AI applications, it is increasingly important to build scalable and efficient parallel ANNS

solutions that can scale to massive modern datasets.

Approximate Nearest Neighbor Search Algorithms. Data structures for ANNS fall roughly into two categories: graph-based indices and partition-based indices. A graph-based algorithm constructs a graph where the nodes represent points in the index and the edges represent proximity relationships, and where nearest neighbor queries are answered by applying a heuristic search on the graph. Prominent examples of graph-based algorithms include NSG [84], HNSW [126], DiskANN [161], but the academic literature includes many other graph-based approaches [1, 2, 4, 5, 7, 53, 62, 74, 85, 96, 105, 106, 124, 129, 135, 150, 182].

Partition-based algorithms, which truncate the search space by partitioning vectors into buckets and exhaustively searching only a small number of buckets during a query. They fall into three categories: inverted file indices, locality-sensitive hash (LSH) functions, and tree-based methods. Inverted file structures typically use a clustering algorithm to assign vectors to posting lists, with distance to a representative element used to determine which lists a query is mapped to. Some notable IVF-based algorithms include FAISS-IVF [75, 77, 109, 112], and many others [2, 5, 34, 63, 117, 178]. LSH algorithms use a *locality sensitive hash function*, which hashes similar points to the same bucket. Some prominent examples include PLSH [162], FALCONN [29], and many others [94, 144]. Trees such as kd-trees or cover trees are well-known data structures for computing nearest neighbors in metric space with low dimensionality (either actual or intrinsic) [32, 44, 91, 117], and are useful for many such applications [46, 69, 180]. Their search methods are subject to the curse of dimensionality, but there are some modified tree-based approaches adapted for high dimensional search [3, 6, 125, 134].

ANNS at a Billion Scale. While most ANNS algorithms validate their results on datasets in the range of one million to ten million points, there are a handful of papers which specifically focus on experiments on hundreds of millions to billions of points. Early work on ANN measured performance on datasets with up to a billion points using various forms of IVF [39, 110, 162, 178]. The results for FAISS [75], the best known of the algorithms in this class, have been reported for the BIGANN and DEEP billion scale datasets [111]. These works do not include comparisons to graph-based algorithms, and focus on recall (a measure of accuracy formally defined in Chapter 2) for the single nearest neighbor instead of the k nearest neighbors.

Other works use secondary storage-based algorithms to scale to billion-scale datasets. DiskANN [161], a graph-based algorithm, gives numbers for BIGANN and DEEP for a billion points. They present limited comparisons to the FAISS [75] and IVFOADC+G+P algorithms [39]. The SPANN system [63] uses an inverted index where the posting lists are stored in secondary memory.

Johnson, Douze, and Jégou [112] report billion scale numbers on a GPU-based implementation using an inverted-index-based approach. Here again, the recall rates are low and the implementation is only compared to another GPU-based system [177]. Recent work on BLISS [94] uses the same datasets as we do at a billion scale. They compare their approach to HNSW, but the numbers they report for HNSW are much worse than those we have found and that are reported here (over an order of magnitude). Several systems work on a billion or more points, but do not report numbers or comparisons to other systems [5, 84, 117, 126].

Benchmarking ANNS. There are three main efforts in benchmarking ANNS algorithms. The first is the ANN Benchmarks repository focusing on million-scale datasets [35]. This is a benchmark suite of ANNS algorithms where any contributor may submit an ANNS algorithm to be included in their public evaluations. Each algorithm is run by the authors on up to nine million-scale datasets. The Billion Scale ANNS Challenge, a competition hosted at NeurIPS 2021 [157], focused on billion-scale ANNS algorithms on three different hardware tracks and six different billion-size datasets, including one range query dataset and two datasets that exhibit out-of-distribution characteristics. The Practical Vector Search challenge, a competition hosted at NeurIPS 2023 [158], expanded the benchmark suite to four more specialized tasks in vector search.

Vector Databases. So-called *vector databases*, which support a vector retrieval data structure along with various guarantees on properties such as updates, determinism, crash recovery, and the ability to synchronize across replicas, have been a huge topic of interest in industry over the past few years. A few examples of commercial vector databases include Pinecone [13], Weaviate [16], Azure CosmosDB [166], QDrant [14], Milvus [12], ChromaDB [17], pgvector [18], VBase [183], Lucene [11], SingleStore [20], and Rockset [19]. Despite this interest, we do not know of any academic work on the design and performance of vector databases, or indeed on their core properties or definition.

Algorithmic Challenges in ANNS. In addition to making existing ANNS indices parallel and deterministic, there are still applications for which significant algorithmic development in ANNS indices is needed. One such area needing algorithmic development is high-dimensional range search. Range retrieval is useful for applications such as content moderation and duplicate detection [76, 77, 130], as well as some graph processing tasks, and some search tasks where the queries have large variance in the number of valid search results [163].

Another algorithmic challenge in ANNS is *filtered search*, in which a vector is augmented with some additional attributes or tags. Filtered search is crucial for applications such as online searching, where a user wishes to filter search results by attributes such as size or color. While filtered search is a core component of commercial vector databases such as Pinecone [13], Weaviate [16], QDrant [14], Milvus [12], and many more, there is little existing academic work on filtered nearest neighbor search. Some initial academic work on filtering includes Filtered-DiskANN [87] and AnalyticDB-V [173]. For the NeurIPS 2023 Practical Vector Search Challenge [8], the FAISS library [109] provided an IVF-based implementation that used bitmaps to identify elements satisfying the filter predicates in each bucket.

In addition to filtered search, a prominent algorithmic challenge in ANNS is presented by datasets that are "difficult" in some way or another. One example of datasets posing an algorithmic challenge is that of *out-of-distribution* queries, where the dataset the index is built on and the data queried are from different distributions and perhaps trained by different embedding models. Out-of-distribution queries tend to achieve significantly lower recall than in-distribution queries, as the indexing structure may "over-fit" to the dataset contained in the index. This is particularly apparent for the case of IVF indices such as FAISS, which are known to experience drops in recall of up to 70 percentage points on out-of-distribution datasets, but is also a problem

for graph-based ANNS algorithms. Some existing work on modifying DiskANN for the out-of-distribution setting makes progress towards closing the gap between in-distribution and out-of-distribution for graph-based ANNS algorithms [107]. Furthermore, at least one LSH partitioning scheme increases the recall on out-of-distribution datasets by taking the query distribution into account when assigning points to buckets [94]. However, there is much left unknown in terms of both closing the recall gap and gaining a better analytic understanding of the deficiencies of existing methods.

1.2.1 Contributions of This Thesis

ParlayANN. Approximate nearest-neighbor search (ANNS) algorithms are a key part of the modern deep learning stack due to enabling efficient similarity search over high-dimensional vector space representations (i.e., embeddings) of data. Among various ANNS algorithms, graph-based algorithms are known to achieve the best throughput-recall trade-offs. Despite the large scale of modern ANNS datasets, existing parallel graph-based implementations suffer from significant challenges to scale to large datasets due to heavy use of locks and other sequential bottlenecks, which 1) prevents them from efficiently scaling to a large number of processors, and 2) results in non-determinism that is undesirable in certain applications.

To combat this issue, we introduce ParlayANN, a library of deterministic and parallel graph-based approximate nearest neighbor search algorithms, along with a set of useful tools for developing such algorithms. In this library, we develop novel parallel implementations for four state-of-the-art graph-based ANNS algorithms that scale to billion-scale datasets. Our algorithms are deterministic and achieve high scalability across a diverse set of challenging datasets. In addition to the new algorithmic ideas, we also conduct a detailed experimental study of our new algorithms as well as two existing non-graph approaches. Our experimental results both validate the effectiveness of our new techniques, and lead to a comprehensive comparison among ANNS algorithms on large scale datasets with a list of interesting findings. This work appeared in PPoPP 2024 [127].

Range Search with Graph-Based Indices The approximate nearest neighbor search (ANNS) problem has been extensively studied in recent years. However, comparatively little attention has been paid to the related problem of finding all points within a given distance of a query, the *range retrieval* problem, despite its applications in areas such as duplicate detection, plagiarism checking, and facial recognition. In this thesis, we present a set of algorithms for range retrieval on graph-based vector indices, which are known to achieve excellent performance on ANNS queries. Since a range query may have anywhere from no matching results to thousands of matching results in the database, we introduce a set of range retrieval algorithms based on modifications of the standard graph search that adapt to terminate quickly on queries in the former group, and to put more resources into finding results for the latter group. Due to the lack of existing benchmarks for range retrieval, we also undertake a comprehensive study of range characteristics of existing embedding datasets, and select a suitable range retrieval radius for eight existing datasets with up to 100 million points in addition to the one existing benchmark. We test our algorithms on these datasets, and find up to 100x improvement in query throughput over a

naive baseline approach, with 5-10x improvement on average, and strong performance up to 100 million data points. This work is currently under submission.

1.2.2 Outline

In Chapter 2 we cover needed preliminaries. In Part I we cover contributions on low-dimensional nearest neighbor search. We present the zd-tree in Chapter 3, and the CLEANN-Tree in Chapter 4. Part II contains our contributions on high-dimensional nearest neighbor search, with ParlayANN presented in Chapter 5, and our work on range search in Chapter 6. The conclusion is presented in Part III, Chapter 7.

Chapter 2

Preliminaries

In this chapter we address some necessary preliminaries. In the first part, we define the problems of nearest neighbor search and range search, as well as the standards by which we judge an approximate solution. In the second part, we define the concepts in parallelism and concurrency used within this thesis.

2.1 Nearest Neighbor Search and Range Search

Given a dataset S of n elements, we use a d-dimensional point (vector) p to denote each $s \in S$, and denote all such vectors as set $\mathcal{P}$. For two points $p, q \in \mathcal{P}$, define the ***distance*** between p and q as $\|p, q\|$. Smaller distance between p and q indicates more similarity. In this work we typically use Euclidean distance (L_2 norm). We also use negative inner product as a distance function in order to use ANNS algorithms for maximum inner product search.

$$\|p, q\|_2 = \sqrt{\sum_{i=0}^{d-1} (p_i - q_i)^2} \qquad \text{Euclidean Distance}$$

$$\|p, q\|_\cdot = \sum_{i=0}^{d-1} -(p_i * q_i) \qquad \text{Inner Product Distance}$$

Definition 1. *(k-NNS) Given a set of points $\mathcal{P}$ in d-dimensions and a query point q, the k nearest neighbor search (k-NNS) problem finds a set $\mathcal{K} \subseteq \mathcal{P}$ with size $|\mathcal{K}| = k$, such that,*

$$\max_{p \in \mathcal{K}} \|p, q\| \leq \min_{p \in \mathcal{P} \setminus \mathcal{K}} \|p, q\|$$

We define k-ANNS as k-approximate NNS. With clear context, we omit k and call them NNS and ANNS. We now introduce the measure of accuracy most commonly used for ANNS, frequently referred to as *recall*.

Definition 2. *(k-recall at n) Let $\mathcal{P}$ be a set of points in d-dimensions and let q be a query point. Let $\mathcal{K}$ denote the true k-nearest neighbors of q in $\mathcal{P}$. Let $\mathcal{N} \subset \mathcal{P}$ be a an output of an ANNS algorithm of size n. Then the k-recall at n of q is:*

$$\frac{|\mathcal{K} \cap \mathcal{N}|}{|\mathcal{K}|}.$$

This is also denoted as $k@n$ recall.

The most common choice of recall is 10-recall at 10 (10@10). We use the term "recall" without a signifier to refer to 10@10 recall. Throughout this work we also refer to the recall of an entire query set. This refers to the average recall over all points in the query set.

A similar problem related to geometric processing is range search, which refers to returning all points within a certain radius as opposed to only the k closest.

Definition 3. *(Exact range search) Given a set of points $\mathcal{P}$ in d-dimensions, a query point q, and a radius $r > 0$, the exact range search problem finds a set $\mathcal{V}$ such that:*

$$\forall v \in \mathcal{V},\ ||v, q|| \leq r \ \text{ and } \ \forall p \in \mathcal{P} \setminus \mathcal{V},\ ||p, q|| > r.$$

In this work, we use "range search" to refer to approximate range searching unless otherwise stated. Since the number of results returned by a range query is not fixed, an alternate definition of accuracy must be provided. We use the following definition from [157]:

Definition 4. *(Average precision) Let $\mathcal{P}$ be a set of points in d-dimensions and let $\mathcal{Q}$ be a set of query points. For each $q_i \in \mathcal{Q}$, let $\mathcal{V}_i$ denote the true set of range results and let $\mathcal{L}_i$ denote the set of reported range results. Then the average precision of set $\mathcal{Q}$ is defined as*

$$\frac{1}{|\mathcal{Q}|}\left(\sum_{q_i \in \mathcal{Q} \mid \mathcal{V}_i \neq \emptyset} \frac{|\mathcal{V}_i \cap \mathcal{L}_i|}{|\mathcal{V}_i|}\right).$$

2.2 Parallelism and Concurrency

Computational Model. We use the ***binary fork-join model*** for parallelism [31, 50, 51]. We assume a set of threads that share the memory. Each thread acts like a sequential RAM plus a `fork` instruction that forks two child threads running in parallel. When both child threads finish, the parent thread continues. A parallel-for is simulated by `fork` for a logarithmic number of steps. The randomized work-stealing scheduler can execute such a computation efficiently [31, 51, 90].

Costs are measured in terms of the work (total number of instructions across all processes) and span or depth (longest dependence path among processes). Any algorithm in the binary forking model with W work and S span can be implemented on a CRCW PRAM with P processors in $O(W/P + S)$ time with high probability [31, 52], so the results here are also valid on the PRAM, maintaining work efficiency.

Concurrency We consider a concurrent execution of a series of steps in the asynchronous shared-memory model with n processes, or threads. An execution is an alternating sequence of configurations and steps. A configuration is a global view of the system at one point in time, and a step is a memory primitive, its arguments, return values, and the process which executes the operation. The steps taken by a thread during an execution implement operations, and a *data structure* is a set of operations. Each operation has a sequential specification, meaning the expected outcome of the operation if its steps are not interleaved with those of any other process. The operation's *execution window* is defined by its invocation and response, which specify its arguments and return values, respectively. The standard of correctness for a concurrent data structure is *linearizability*, which means that each operation appears to have taken place atomically in some sequential order [99]. Formally,

Definition 5. *(Linearizability, wording from [174]) An execution α is linearizable if, for every complete operation op in α (as well as for some of the uncompleted operations), we can assign it a linearization point within its execution interval, so that in the sequential execution defined by the linearization points, each operation has the same response as in α.*

Another desirable condition on concurrent data structures is *lock-freedom*, which means that across the processors, at least one thread will continue to make progress. A stronger condition than lock freedom, *wait freedom*, means that each thread is guaranteed to make progress on its execution. Formally,

Definition 6. *(Linearizability, wording from [174]) An implementation of a data structure D is lock-free if, in any infinite execution in which processes follow this implementation, infinitely many operations complete.*

Definition 7. *(Linearizability, wording from [174]) An implementation of a data structure D is lock-free if, in any infinite execution in which processes follow this implementation, each process executes an infinite number of steps.*

Part I

Low-dimensional Nearest Neighbor Search

Chapter 3

Parallel Nearest Neighbors in Low Dimensions with Batch Updates

3.1 Introduction

Computing nearest neighbors is one of the most fundamental problems in computer science, with applications in diverse areas ranging from graphics [66, 132, 139, 141] to AI [123] to as far afield as particle physics [151]. Research on nearest neighbors can be roughly divided into two areas: one area focuses on computing approximate nearest neighbors in high dimensions, primarily with clustering as an application. The second focuses on exact (or closer to exact) nearest neighbors in lower dimensions, with tasks such as surface reconstruction [24, 82] as a prominent application. This work focuses on the latter category.

The most common method of computing nearest neighbors in low dimensions is via a kd-tree [43], a tree which keeps the entire bounding box of the point set at its root, and whose children represent progressively smaller enclosed bounding boxes. Kd-trees have many applications in point-based graphics, and have been the data structure of choice for many graphics practitioners [139], even though other methods have better worst-case guarantees. One of the best kd-tree implementations is Arya et al's [33], which has been used widely by researchers [66, 132, 141]. Another commonly used library of kd-trees is that of the Computational Geometry Algorithms Library (CGAL) [164]. Recent work on kd-trees has focused on better theoretical guarantees [146], and with better performance in high dimensions [65].

Another approach for computing nearest neighbors uses space-filling curves known as the Morton ordering, z-ordering, or Lebesgue ordering (henceforth Morton ordering). Recursing by splitting the Morton ordering roughly splits space, making it possible to effectively search for nearest neighbors. Two nearest neighbor algorithms that make use of Morton ordering are Chan's minimalist nearest neighbor algorithm [55], and Connor and Kumar's k-nearest neighbor graph algorithm [70]. Other approaches to computing nearest neighbors include well-separated decompositions [54], and Delaunay triangulation [45].

Some important considerations when choosing a k-nearest neighbors algorithm are how it

performs (theoretically as well as practically), does it run efficiently in parallel (since todays machines only have multiple processors), what kind of point sets it handles, and whether it supports dynamic updates (since in many applications point sets change over time [159]). Vaidya [167] and Callahan and Kosaraju [54] give strong bounds for general point sets computing all nearest neighbors in $O(n \log n)$ time using variants of kd-trees. Chan improved this to $O(n)$ time if the the ratio of the largest distance to the smallest is polynomially bounded [56]. However these results are limited to static point sets and have not yet shown to be practical. Connor and Kumar give bounds under the assumption of bounded expansion constants [70] for a practical algorithm they implement. There has also been significant interest in parallel algorithms for the problem. This includes implementations based on MapReduce [23], for GPUs [100], the STANN library [70], and an implementation in CGAL [25]. Although in principle kd-trees should be able to support dynamic updates we know of no libraries that efficiently support them, and few interesting theoretical bounds for the problem in low dimensions. When considering parallelism and updates together one should be interested in batches of updates that can be processed in parallel.

In this chapter, we present a technique that combines the ideas of kd-trees and Morton ordering to achieve efficient algorithms for k-nearest neighbors in bounded dimension. Some guiding intuition for such a combination is that Morton-based algorithms tend to have quick preprocessing (since only a sort is required) and slower queries; on the other hand, tree-based algorithms can have slower building times but their additional structure leads to faster queries. Thus, combining these approaches may allow us to achieve the advantages of both. In particular we present a k-nearest neighbor algorithm that hybridizes the kd-tree and Morton order approaches by using a kd-tree whose splitting rule is based on the Morton ordering; we call this tree the **zd-tree**. We also present what is to our knowledge the first parallel batch-dynamic update algorithm for a k-nearest neighbor data structure. We prove the following theoretical results in the context of a point sets with bounded expansion constant and bounded ratio, two reasonable and broadly used assumptions when computing nearest neighbors [27, 28, 33, 44, 56, 70, 86, 115, 116, 153].

In Theorem 1, we show that the zd-tree can be build in $O(n)$ work and $O(n^\epsilon)$ span, where n is the size of the point set and $0 < \epsilon \leq 1$. In Theorem 2, we show that the expected work required to find the k-nearest neighbors of a point p is $O(k \log k)$. These two theorems together imply a linear-work algorithm for finding the k-nearest neighbors among a set of points (i.e. the k-nearest neighbor graph). They also imply that for a point $q \notin P$, finding the nearest neighbors requires $O(\log n + k \log k)$ work.

Finally, in Theorem 3, we show that under some distribution assumptions, a batch update of size k to a tree with n existing points be inserted in $O(k \log(n/k))$ work and $O(k^\epsilon + \text{polylog}(n))$ span.

In additional to the theoretical contributions, we implement both our nearest neighbor searching algorithm and the batch-dynamic updates described above, and we measure our nearest neighbor searching algorithm against a large number of competitors. Our algorithms are optimized for parallelism: in addition to presenting a thread-safe data structure so that queries can be conducted in parallel, we use parallelism when recursively building or updating our kd-tree.

A snapshot of our practical results can be found in Figure 3.1, which compares the work needed to pre-process and query a point set across our implementation and competitors.

Our experimental results show the following:

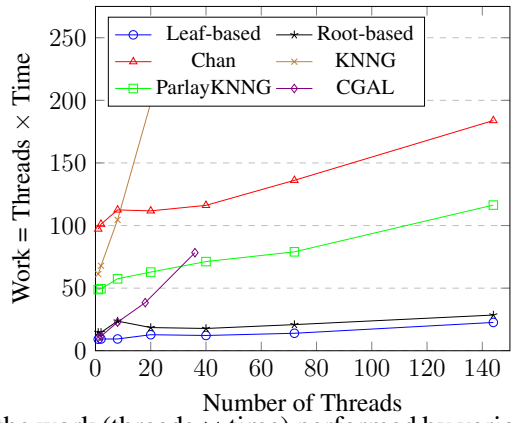


Figure 3.1: A figure showing the work (threads $\times$ time) performed by various nearest neighbor algorithms as the number of threads increases. The k-nearest neighbor graph was computed on 10 million points from a random dataset within a 3D cube. Ideally the line for a particular algorithm would be both low (small total work) and straight (indicating more threads does not change the total work).

1. Our k-nearest neighbor algorithms achieve **high parallelism.** Using our basic algorithm to query all nearest neighbors of a 3D dataset with 10 million points achieves 75-fold speedup on a 72-core Dell R930 with 144 hyper-threads.
2. Our algorithms are **fast**. Our algorithm's speed is robust across all the measures which we tested—adversarial datasets, varying k, varying the size of the dataset, and varying the number of threads. In most cases, it beats its competitors by close to an order of magnitude.
3. Our batch-dynamic updates **drastically decrease the cost per insertion**. An insertion of one point into a tree of $5,000,000$ points takes about 10^{-5} seconds, while an insertion of $100,000$ points takes 10^{-7} seconds per pointelement.

3.1.1 Preliminaries.

For the special case where we are given a point set P and wish to calculate the k-nearest neighbors of all points in P, we refer to the result as the **k-nearest neighbor graph** of P. A query of a point not in P is a **dynamic query**, and a query of a point in the tree is sometimes referred to as as a **non-dynamic query**. Similarly, adding a point to or deleting a point from the tree is referred to as a **dynamic update**, which is **batch-dynamic** if the updates are processed in batches rather than one at a time.

Kd-trees. Many nearest neighbor algorithms use a kd-tree as the data structure to query for nearest neighbors. Given a set of points P in a d-dimensional bounding box, a kd-tree splits the data into two smaller bounding boxes at every level of the tree. Designing a kd-tree requires making a choice of **splitting rule**—that is, how the bounding box will be divided. One common splitting rule is to divide the bounding box of points along its largest dimension; another is to divide the space such that equal numbers of points are on each side. A variant of the kd-tree is the $\{$*quad,oct*$\}$*-tree*, where each internal node of the tree has 2^d equal sized children in d-dimensional space.

Morton ordering. A common tool used in designing k-nearest neighbor algorithms is the Morton ordering. For a set of points whose coordinates are d-vectors of integers $(x_1, x_2, \ldots, x_d)$, the Morton ordering is calculated by taking each integer coordinate in binary form, interleaving the coordinates to create one integer per point, then sorting using the interleave integers. Nearest neighbor algorithms take advantage of the following property of the Morton ordering: given any two points p and q and the rectangle defined by those points as the corners, then all points in the rectangle must fall between p and q in the Morton ordering. This allows pruning regions of the ordering.

Bounded expansion constant. Several previous works for nearest neighbors in metric spaces have assumed a bounded expansion constant [27, 28, 44, 70, 86, 115, 116, 153], which roughly requires that the density of points in the metric space does not change rapidly. In the context of Euclidean space we use the following definition. Given a point p_i and a positive real r, let $\text{box}(p_i, r)$ denote the box centered at p_i with radius r (that is, half the side length).

Definition 8. *Given a point set P contained in a bounded Euclidean space X, P has **expansion constant** γ if for all $x \in X$ and all positive real r, if $|box(x, r)| = k$ for any $k > 1$ then*

$$|box(x, 2r)| \leq \gamma k.$$

*The expansion constant is referred to as **bounded** if $\gamma = O(1)$.*

Bounded Ratio. Another property that will be needed to prove some of our theorems is that of the point set having bounded ratio. This is a commonly used property in problems such as nearest neighbors and closest pair [33, 56].

Definition 9. *Given a point set P of size n, let $d_{\max}$ denote the maximum distance between any two points in the set, and let $d_{\min}$ denote the minimum distance between any two points in the set. Then P has **bounded ratio** if*

$$\frac{d_{\max}}{d_{\min}} = poly(n).$$

3.1.2 Related Work.

Arya and Mount's k-nearest neighbor implementation [33] is commonly referred to as the state-of-the-art sequential k-nearest neighbor algorithm for low dimensions. Their implementation uses a kd-tree known as a *balanced box decomposition (BBD) tree*, whose splitting rule attempts to get the best of both commonly used splitting rules—that is, splitting a bounding box into approximately equal areas which also have approximately equal numbers of points. They show that using a BBD tree, an approximate k-nearest neighbor query takes $O(\frac{k}{\epsilon} \log n)$ work. The BBD tree theoretically supports insertions, but their given implementation does not. In [70], Connor and Kumar parallelize Arya and Mount's all-nearest neighbors implementation and show that theirs produces faster results, so we compare against Connor and Kumar's instead of theirs.

One nearest neighbor algorithm which uses the Morton ordering instead of a kd-tree is Chan's "minimalist" nearest neighbors algorithm [55], which has a theoretical guarantee of $O(n \log n)$

expected preprocessing time and $O(\frac{1}{\epsilon}\log n)$ expected time per query for approximate nearest neighbors. The algorithm is notable for both its simple proof and strikingly minimalist implementation, whose sequential version requires fewer than 100 lines of code in C++. Chan's algorithm first randomly shifts the coordinate of each point, then sorts the points using the Morton ordering. The algorithm then uses an implicit tree, recursively dividing the sorted points and visiting every implicit vertex which is within some radius of the query point. An adversarial case for this algorithm is when some query point q is in the right half of the sorted data and its nearest neighbor p is in the left half, causing the algorithm to search a large number of vertices. The random shift helps avoid this case in expectation.

The k-nearest neighbor implementation that most closely matched ours—in that it is tailored for parallelism and for exact nearest neighbor searching in low dimensions—is that of Connor and Kumar in [70], where STANN stands for Simple Threaded Approximate Nearest Neighbor. Their algorithm makes several improvements on Chan's algorithm, especially for the case of computing the k-nearest neighbor graph. Their main improvement is to search from the leaf of the implicit tree rather than the root, which allows for the possibility of searching only $O(k)$ implicit nodes instead of at least $O(\log n)$ (and this would be a best case scenario where the tree is perfectly balanced). Indeed, they find that if the input point set has bounded expansion constant, their data structure uses $O(n \log n)$ work and their nearest neighbor queries use expected $O(k \log k)$ work. Their algorithm only works for static point sets and as our experiments show is not as fast as ours.

Another well-known tree used for computing nearest neighbors is Callahan and Kosaraju's well-separated pair decomposition [54]. For n points, they can build their tree (similar to a kd-tree) in $O(n \log n)$ work and polylogarithmic span (in parallel). Based on the tree, they can build the decomposition and find nearest neighbors in $O(n)$ work and polylogarithmic span. The approach is only described for the static case.

Another approach is to use the Delaunay triangulation of the set of points [45]. Although this seems to work reasonably well in two dimensions, in three and higher dimensions it can be very expensive. Beyond bounded expansion constant, another common geometric assumption used for finding nearest neighbors or the closest pair is a bound on the ratio of the furthest pair and the closest pair in a dataset [33, 56, 67, 68, 80].

The K-nearest neighbor (KNN) data structure problem is to maintain a set of points P in d dimensions while supporting queries that given a new point return the k nearest points in P (Euclidean distance). KNN data structures in modest dimensions have been used for decades as a key component for solving problems such as ray tracing, image reconstruction, motion planning, collision detection, and hierarchical clustering [66, 79, 132, 139, 141, 151], and many implementations have been developed [25, 33, 55, 70]. Most such structures are based on some form of tree structure, Since applications of the problem often have a large number of points, and processing many queries quickly is important, there has been a large interest in parallel algorithms and implementations [23, 46, 100, 100, 101, 102, 172, 184].

The majority of early work on parallel KNN data structures was for static point sets. Most recent work, however, has studied KNN structures for dynamically changing point sets [46, 60, 101, 102, 180]. Current parallel or concurrent solutions for the dynamic case, however, have limitations.

3.2 Algorithm Design and Bounds

Here we describe our algorithms for constructing a data structure for k-nearest neighbors, querying the structure, and batch updating it.

Data structure. The data structure we use for nearest neighbor searching is a kd-tree whose splitting rule uses the Morton ordering; this is what we refer to as the zd-tree. Since the Morton ordering is just the interleaving of the bits of each coordinate, the tree is built by letting the root represent the entire bounding box, and splitting the points into child nodes at level i based on whether the bit at place i is 0 or 1. In three dimensions, our tree is almost equivalent to an oct-tree in which every three levels of our tree corresponds to one level of the oct-tree; however, the leaves can be at different levels. Each internal node of the tree stores the two opposing corners defining its bounding box, its two children, and its parent. Each leaf node stores its two opposing corners, its parent, and the set of points it contains. We bound the number of points in a leaf by a constant, and a leaf can be empty. Note that every point covered by the root bounding box is included in exactly one leaf node.

Construction. Before the zd-tree can be built, we preprocess the input. Firstly, motivated by Chan [55], and necessary for our bounds (the proof of Theorem 2), we select a random shift for each coordinate, and shift all the coordinates by this amount. This shift is kept throughout. We then sort the points by the Morton order. This can use Chan's comparison function, which leads to an $O(n \log n)$ work sort, but as we describe in Section 3.2.1 can be reduced to a linear time radix sort with span $O(n^{\epsilon})$ [108] when assuming a bounded expansion constant. In this case the number of bits needed for the Morton order can bounded by $O(\log n)$.

After shifting and sorting we apply a divide-and-conquer algorithm (Algorithm 3.1) to build the zd-tree. The algorithm recurses at each level of the tree on the two sides of the cut for the given bit of the Morton ordering. Importantly, finding the cut in the routine `split_on_bit` only requires a binary search since the points are sorted by Morton order. This implies that when the tree is sufficiently shallow (guaranteed by bounded ratio) the work to build the tree is only linear, and the parallel depth is low. Even if completely imbalanced the work would be $O(n \log n)$.

Downward search algorithm. Our downward search algorithm is detailed in Algorithm 3.2. The algorithm maintains a current set of k nearest neighbors, which starts empty and is improved over time by inserting closer points. In our pseudocode, we use N to represent the nearest neighbor candidate set. The downward search works as follows: let r be the distance from p to the furthest element in N if N contains at least k elements, or infinity otherwise. Now search vertex v only if the bounding box for v intersects a ball of radius r around p. This is determined by the `within_epsilon_box` function. If the node is a leaf, iterate through the points it contains and update the set of nearest neighbors if necessary. If it is not a leaf, recurse on its children, searching first the child whose center is closest to the query point p.

Our **root-based** algorithm simply starts at the root of the zd-tree with an empty N and applies `search_rec`, but we also use `search_rec` in our upward algorithm.

Upward search algorithm. Our upward search algorithm is detailed in Algorithm 3.3. It always starts at the leaf in the tree containing the point p and works its way up the tree. The idea is

that in general, only a small part of the tree needs to be examined. It uses the downward search as a subroutine. As in the downward search, it maintains a priority queue N, initially empty, of the current estimate of the nearest k neighbors, which is improved over time helping to prune further search. The algorithm starts at the leaf by adding any points in the leaf to N. Then, as with the downard version, let r be the distance between p and k-th nearest neighbor in N, or infinity if there are not k neighbors in N yet. Now search the parent of the current node if and only if the ball of radius r around p extends outside the bounding box of the current node. Otherwise we know there are no points not included in the current node that could be closer than those in N. This can use the same `within_epsilon_box` as used in the downward algorithm, but with a negative r when searching the parent we search the parent's other child using the downward algorithm.

Finding the leaf in which a point p belongs, which is needed, depends on whether we are generating a k nearest neighbor graph or using the the structure for dynamic searches for points not in the set. In the first case we know the leaf since each point is in a leaf. Therefore to generate a k-nearest neighbor graph we need just build the tree and then run `search_from_leaf` on each point in each leaf. We refer to this as the **leaf-based** version of our algorithm. In the second case we have to search down the tree from the root to find the location of the leaf. This can use the bits of the Morton ordering to decide left or right. We refer to this as the **bit-based** version, and the downward search from the root as the **root-based** version.

Batch-dynamic updates. The tree data structure naturally lends itself to the possibility of dynamic insertions and deletions. Insertion of a new point q into a zd-tree T is conceptually simple: locate the leaf of T which q should be inserted into; then either add q to the sequence of points contained in the leaf, or if q would cause the number of points in the leaf to exceed some cutoff, split the leaf into two children. This concept can be refined to a parallel batch-dynamic algorithm, which takes a set of points and recurses in parallel down the right and left children of the root. One small subtlety is that to avoid cases where an insertion might require a rebuild of the entire tree, we require a bounding box that all data will be contained in to be specified before building the initial tree.

As with building the tree from scratch, a batch-dynamic insert starts by using the random shift to offset the points and sorting the points to be added based on their Morton ordering. We then apply the recursive algorithm shown in Algorithm 3.4. Deletions use an almost identical algorithm.

3.2.1 Theoretical Results.

In this section, we give theoretical results on the performance of our algorithms when assuming bounded expansion constant. The results in the rest of the section assume that the point set P has bounded expansion constant $\gamma \geq 2$ as well as bounded ratio, and they assume that the dimension $d = O(1)$. We also require that every coordinate of every point is unique. This is a fair assumption to make in the context of nearest neighbors, since every point set that does not have this property can be transformed into one that does and where each point retains the same nearest neighbors. The presented proofs in this section assume that X is a bounding hyperrectangle. Wherever not otherwise specified, we use B to denote the bounding box of the randomly shifted point set; note that the side lengths of B can be at most twice the side lengths of the smallest

```
1  node* build(vec<point> P){
2     vec<interleave_integer> Q = compute_interleave_integers(P);
3     sort(Q);
4     return build_rec(Q, sizeof(interleave_integer));
5     }

7  node* build_rec(vec<interleave_integer> Q, int b){
8    if(b==0 | Q.size() < cutoff){
9        return leaf(Q);
10    }else{
11       i = split_on_bit(Q, b);
12       do in parallel {
13          L = build_rec(Q[1:i], b-1);
14          R = build_rec(Q[i:n], b-1);
15       }
16       return node(L, R);
17    }
18 }
```

Algorithm 3.1: Building a zd-tree

```
1  priority_queue <point> search(point p, int k, node* T){
2     double r = infty;
3     priority_queue Q; // priority is distance from p
4     search_rec(T, p, k, r, Q);
5  }

7  void search_rec(node* T, point p, int k, double r, priority_queue Q){
8    if(within_epsilon_box(T, p, r)){
9     if(T->is_leaf()){
10      sequence<point> P = T->points;
11      for q in P{
12       if(dist(p,q) < r){
13         if(Q.size() == k) Q.pop();
14         Q.push(q);
15         r = dist(p, Q.top()) // kth closest element to p
16    }}} else if(search_heuristic(T) == T->Left()){
17      search_rec(T->Left(), p, k, r, Q);
18      search_rec(T->Right(), p, k, r, Q);
19    } else{
20      search_rec(T->Right(), p, k, r, Q);
21      search_rec(T->Left(), p, k, r, Q);
22 }}}
```

Algorithm 3.2: Searching for k-nearest neighbors on a zd-tree.

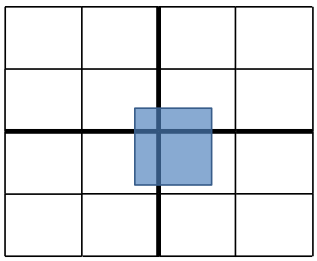
```
priority_queue<point> search_from_leaf(leaf* T, point p, int k){
   double r = infty;
   sequence<point> P = T -> points;
   priority_queue Q; // priority is distance from p
   for q in P{
      if(dist(p,q) < r){
         if(Q.size() == k) Q.pop();
         Q.push(q);
         r = dist(p, Q.top()) // kth closest element to p
 }}
   node* C = T -> parent();
   while !within_epsilon_box(C, p, -r) && C != null do{
      if C->Left() == C{
         search_rec(C->Right(), p, k, r, Q);
      } else{
         search_rec(C->Left(), p, k, r, Q);
      }
      C = C -> Parent();
      r = dist(p, Q.top());
   }
   return Q;
}
```

Algorithm 3.3: Searching for k-nearest neighbors on a zd-tree starting at the leaf.

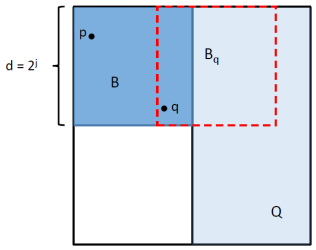
```
node* batch_insert(vec<point> P, zd_tree T){
   R = T->root();
   vec<interleave_integer> Q = compute_interleave_integers(P);
   sort(Q);
   return batch_insert_rec(Q, R);
   }

node* batch_insert_rec(vec<interleave_integer> P, node* T){
 if(T -> is_leaf()){
      if size(P) + size(T) < cutoff {
         for p in P{
            T->insert(p);
         }
      } else{
         L, R = insert_and_split(T, P);
      }
   } else{
      b = T -> bit;
      i = split_on_bit(P, b);
      do in parallel{
         batch_insert_rec(T->Left(), P[1:i]);
         batch_insert_rec(T->Right(), P[i:n]);
      }
   }
   update_bounding_box(T, P);
}
```

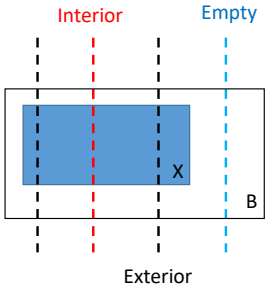
Algorithm 3.4: Inserting a batch of vectors into the zd-tree.



(a) A graphic illustrating a box situated inside the largest possible quad-tree box as referenced in Lemma 2.



(b) A figure illustrating the setup in Lemma 3. Here, the empty boxes represent empty splits, the light blue boxes represent unbalanced splits, and the dark blue box represents the box containing p.



(c) An illustration of the three types of cuts referenced in Lemma 4: interior in red, exterior in black, and empty in blue.

Figure 3.2: Figures aiding with the proofs in Section 3.2.1.

bounding box containing X.

Our first theorem concerns the height and build time of a zd-tree on a point set with bounded ratio.

Theorem 1. *For a point set P of size n with bounded ratio, the zd-tree can be built using $O(n)$ work with $O(n^\epsilon)$ span, and the resulting tree height $O(\log n)$.*

Proof. For the tree depth and work bound, we need to show that the longest path in the tree has length $O(\log n)$. The bounding cube of X has side length within a constant factor of $d_{\max}$, and it must be divided until the two points whose distance between them is $d_{\min}$ are in separate cubes. Since $d_{\max}/d_{\min} = \text{poly}(n)$, $d_{\max}$ must be halved $O(\log n)$ times to reach $d_{\min}$. Thus the tree has $O(\log n)$ depth.

For the sorting claim, we wish to show that a parallel radix sort can be used. For the radix sort to require $O(n)$ work, we need to guarantee that only $O(\log n)$ bits are required to sort the dataset. This follows directly from the fact that the tree has depth $O(\log n)$. □

Now, we work towards the theorem on the expected time required to query the k-nearest neighbors of a point p. Like the result from Connor and Kumar in [70], the $O(k \log k)$ bound only applies when searching for the nearest neighbors of a point already in the tree; for a dynamic query, finding the leaf to start from requires $O(\log n)$ time. The following proofs assume without loss of generality that a leaf of the zd-tree contains no more than k points. Furthermore, we assume that the tree is a true quad-tree, meaning that each internal node has 2^d children; thus we use the term "quad-tree box" to refer to the box belonging to a tree node. This will slightly simplify the analysis, and it can only be worse than the performance of the actual algorithm.

Theorem 2. *For a zd-tree representing a point set P of size n with bounded expansion, finding the k-nearest neighbors of a point $p \in P$ requires expected $O(k \log k)$ work.*

The proof of the theorem separates the work into two parts: the work of searching through the points in a leaf of the zd-tree, and the work of traversing the zd-tree to get to those leaves. The first lemma concerns the former.

Lemma 1. *When searching for the k-nearest neighbors of a point p, $O(k)$ candidate points will be considered, resulting in $O(k \log k)$ work to evaluate all the candidate points.*

Proof. The multiplicative factor of $\log k$ comes from the fact that a priority queue is used to store nearest neighbors, so an $O(\log k)$ cost is incurred each time the priority queue is updated.

Consider the leaf L of the tree that would contain p. The initial approximation is found by recursing up to L's ancestor until the ancestor has more than k descendants, then adding those descendants to the priority queue. The ancestor's bounding box B must contain $O(k)$ points by the fact of bounded expansion constant, since one of its children contains fewer than k points.

Now, let r be the side length of B. Our algorithm will search a leaf only if the box belonging to that leaf overlaps with $\text{box}(p, r)$. Those neighbors are contained within radius r of the quad-tree box containing our initial guess.

The search algorithm evaluates every point at radius r from p as a candidate point. If the radius r of B is expanded twice, all the candidate points will be contained in the resultant box; call this box of candidates Q. If $k' = O(k)$ points are in B, the expansion condition guarantees that at most $\gamma^2 k' = O(k)$ points are in Q. However, the search algorithm does not directly search each point in Q; rather, it searches every leaf whose bounding box overlaps with Q. This means that if a leaf L's bounding box were to fall partially inside Q and partially outside, all the points in L would be counted. Since B is a quad-tree box, this could only occur if the leaf L were to have a bounding box with radius larger than r. Since the radius is larger than r and $d = O(1)$, there can only be a constant number of such leaves. Since each leaf has at most k points, the original bound still holds. □

Now, we move on to considering the expected work required to traverse the zd-tree to access the leaves. The probability calculation given here can also be found in [70].

Lemma 2. *When traversing the zd-tree, the expected number of tree edges traversed to find all the nearest neighbors is $O(k)$.*

Proof. The worst case for the cost of the search algorithm is when the search space described in Lemma 1 is only contained in the bounding box containing all of P, as illustrated in Figure 3.2(a). First, we show that the length of the traversal is bounded by the longest path between two leaves in the search space; all other searches can be charged to the $O(k)$ points that are searched. We will use the box B to refer to the search space. The largest cut of B divides B into 2^d quad-tree boxes. Each of these boxes must be contained within a quad-tree box Q_i of at most twice the side length of B. Thus, traversing every leaf in Q_i incurs cost at most $O(k)$; since d is constant, the claim follows.

All that remains is compute the expectation of the length of the longest path in the zd-tree. Without loss of generality, assume that the search space is a box with side length 2^h. Then the probability that the search space is contained within a box of side length 2^{h+j} is $\left(\frac{2^j-1}{2^j}\right)$, since the box must have its upper left corner in one of $2^j - 1$ grid squares along each dimension. Thus, due to the random shift, the probability that the search space is NOT contained within a box of

side length 2^{h+j} is $1 - \left(1 - \frac{1}{2^j}\right)^d$. From the perspective of traversing the zd-tree, this is the event that the path between two leaves in the search space is length j. Thus its expectation can be upper bounded by the following summation, which charges a cost of one for each box the search space is not contained in:

$$\sum_{j=1}^{\infty} \left(1 - \left(1 - \frac{1}{2^j}\right)^d\right) = O(1)$$

and the result follows. □

Lemmas 1 and 2 together compose the proof of Theorem 2.

Now we move on to batch-dynamic updates. In a weight-balanced tree, the argument for the desired $O(k \log(n/k))$ bound would be as follows: when a batch of points is inserted into the tree, the work required to insert them into $\log k$ levels can be no more than the work that would be required to build them into their own tree. Thus insertion into the first $O(\log k)$ levels of the tree uses $O(k)$ work, and the work bound on an insertion is $O(k(\log n - \log k)) = O(\log(n/k))$.

Hence the goal of Theorem 3 is to show that in addition to the traditional notion of balance, the zd-tree also obeys some notion of weight balance—that is, that each split of a point set must produce two halves where each contains a constant fraction of the points. Unfortunately, this is not strictly true, since a set of points with bounded expansion may, for example, have only one element with a 1 at the largest bit if that element is very close to the rest of the elements. However, we will be able to show a slightly weaker notion than weight balance: that enough nodes in the tree are weight-balanced that the same work bound still applies.

One more piece of terminology is needed before the theorem statement. When building the zd-tree by successively splitting the bounding box of the input, a split may have no points in X on one side. During the tree building phase, we face a choice regarding empty cuts: when the algorithm makes an empty cut, we could either fork off two child nodes where one is a leaf containing no points, or we could simply not fork off an empty node, and divide the other node using the next bit. Since it is strictly cheaper, we choose the latter. Thus the following analysis will not deal with empty cuts; in particular, Lemmas 4 and 5 do not consider empty cuts in their analysis, since empty cuts do not affect the length of paths in the tree.

Theorem 3. *Let T be a pruned zd-tree representing point set P, and let Q be a point set of size k, such that $|P|+|Q| = n$. Then if $P \cup Q$ and Q both have bounded expansion and bounded ratio in the same hypercube X, Q can be inserted into T in $O(k \log(n/k))$ work and $O(k^\epsilon + \text{polylog}(n))$ span.*

When X in its bounding box B is being recursively divided using Algorithm 3.1, it will be useful to separate the divisions or cuts into several categories. A cut along dimension d divides some sub-cube S of B in two with cutting plane ℓ. The cut is either **empty**, meaning that ℓ does not touch any points in X; or it is **exterior**, meaning it touches the boundary of X; or it is **interior**, meaning that within S, ℓ does not touch any points on the boundary of X. See Figure 3.2(c) for an illustration.

The first step towards Theorem 3 is to show that interior cuts are weight balanced.

Lemma 3. *One out of every d interior cuts must be weight balanced; that is, it must split its bounding box into sets of size αn and $(\alpha - 1)n$ for constant $\alpha \in (0, 1)$.*

Proof. Consider a point $p \in P$. As the zd-tree is built, the point set P is split into smaller pieces along each dimension. Call a split **unbalanced** if it splits P into pieces of size n_1, n_2 such that one of $n_1, n_2 < \frac{1}{2(1+\gamma)}n$. Refer to a split as "involving" p if it splits a box containing p. We will show that after $d-1$ unbalanced splits involving p, the next split must be balanced.

Assume for contradiction that there are d consecutive unbalanced splits involving p. Let B be the quad-tree box containing p after those splits, and let the length of B's longest side be 2^j. By the assumption that d consecutive unbalanced splits have already happened, there must be some side of B where the most recent split on that side was of a region with maximum side length 2^{j+1}; call the hyperrectangle resulting from that split Q. Let $q \in B$ be the unique closest point to Q, as shown in Figure 3.2(b). Then, consider any $x \in Q$ such that $B_q = \text{box}(x, 2^j)$ contains q and no other point in B, and overlaps only B and Q. Due to its proximity to B, $\text{box}(x, 2^{j+1})$ must completely contain B. By our assumption, B_q contains fewer than $\frac{1}{2(\gamma+1)}n+1$ points, since it contains one point from B and otherwise only points from Q, which has at most $\frac{1}{2(\gamma+1)n}+1$ points by our assumption. The box B contains at least $\left(1-\frac{1}{2(1+\gamma)}\right)^d n$ points, so since $d \geq 2$, the expansion constant is violated and we reach a contradiction. □

While interior cuts are easily shown to be balanced, the same argument does not hold for a sequence of exterior cuts, since Lemma 3 relies on being able to choose a certain point x as the center of a box, and this point might not be included in X if some of the d unbalanced cuts were exterior. Since an arbitrarily long sequence of nodes formed from exterior cuts might not be weight-balanced, we take a different approach: showing that even if we have to pay the maximum possible cost for each unbalanced path, the number of points on such an unbalanced path is small enough that the overall bound is unchanged.

The first step towards this goal is to quantify, for a given point $p \in P$, how many exterior cuts involving p must be made before the first interior cut involving p.

Lemma 4. *Normalize the length of B to n. Then for every point $p \in P$, let $f(p)$ denote the minimum distance from p to the boundary of X, perpendicular to some side of the bounding box B. Then $O\left(\log \frac{n}{f(p)}\right)$ cuts will be made before the next cut containing p is interior.*

Proof. A cut involving p is guaranteed to be interior when the quad-tree box containing p has radius less than $f(p)$. Since the radius of the quad-tree box is halved every d cuts and side length of B is n, the aforementioned condition is met after $d \cdot \log(n/f(p))$ cuts. □

Lemma 4 gives us a way to bound the number of exterior cuts along the path to p. The next step is to bound the *number* of points that can be a given distance or closer to the boundary of one of the faces, which is done in the following lemma:

Lemma 5. *Normalize the side of B to n. Let S be a subset of B formed by cutting B parallel to one of its faces at distance r away from the face. Then at most a $\left(\frac{\gamma^2}{\gamma^2+1}\right)^{\log n/r}$ fraction of the total points in B are contained in S.*

Proof. One way of upper bounding the number of points in S is as follows: S can be formed by dividing the bounding box in half along one dimension $\log(n/r)$ times. On each division, at most how many points can be in the resultant rectangle? Consider the first cut which divides the

bounding cube in half; the goal is to maximize the number of points in one half without violating the expansion constant. The "sparse" half can be separated into 2^{d-1} hypercubes of radius $n/2$, each of which is expanded twice before the whole space X is encompassed. Note that it is optimal for all the "sparse" sub-cubes to contain the same amount of points since each must be able to expand a box around its center. The following equation solves for the largest fraction f of the total points that can be in any one sub-cube without violating the expansion constant.

$$\begin{aligned}\gamma^2 \frac{1-f}{2^{d-1}-1} &\geq x + (2^{d-1}-2)\frac{1-f}{2^{d-1}-1} \\ \implies f &\leq \frac{\gamma^2 - 2^{d-1} + 2}{\gamma^2 + 1}.\end{aligned}$$

The following cuts need to be upper bounded in a slightly different way, since they are cutting a hyperrectangle with $d-1$ sides of length n and one side of length s. Thus the area can be decomposed into $2n/s$ boxes of side length $s/2$. Half the boxes will receive the maximum number of points possible; both halves will evenly distribute their points across the boxes. Each "sparse" hypercube of side length $s/2$ must be expanded twice before it contains a cube C of side length s that is completely contained within the region. For that cube C, the same equations as before can be written to bound the number of points in the dense half of C to a $\frac{\gamma^2 - 2_2^{d-1}}{\gamma^2+1}$ fraction of the total points in C. The total number of points in C is upper bounded by $\left(\frac{\gamma^2 - 2_2^{d-1}}{\gamma^2+1}\right)^k$ where k is the total number of cuts that have occurred, assuming that the maximum possible fraction is on one side each time. The overall bound follows. □

Now, we can put all these pieces together to prove the theorem.

Proof of Theorem 3. The sorting cost bound follows directly from Theorem 1.

For the update bound, we know that over all weight-balanced paths in the tree, the cost of insertion the k points down those paths is $O(k \log(n/k))$. Thus our task is to account for the paths in the tree that are not weight-balanced. In the worst case, for every non-weight-balanced path of length ℓ, we incur an $O(\ell)$ cost for each point that traverses it. We will show that the number of points in Q that are distributed among the unbalanced parts of the tree is small enough that the overall bound is unchanged. Consider one face S of X. For a point p at a given length r away from the boundary of x perpendicular to S, the path traveled from the root to p can encounter $O(\log(n/r))$ unbalanced nodes. Consider all points at a distance r or closer to S. Lemma 5 shows that there are at most $k \cdot \left(\frac{\gamma^2}{\gamma^2+1}\right)^{\log(n/r)}$ such points. The cost incurred for each depth is also $\log(n/r)$. Thus, the maximum cost for all the points at depth $f(n)$ or smaller is $k\left(\frac{\gamma^2}{\gamma^2+1}\right)^{\log(n/r)} \log(n/r)$. Renaming $\log(n/r)$ as the variable x and integrating over values of x

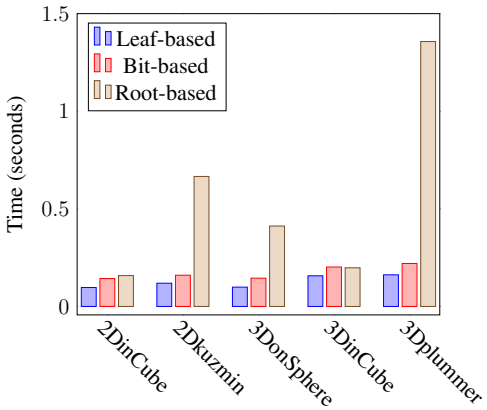


Figure 3.3: A bar chart showing the performance of our three k-nearest neighbor algorithms on different datasets. All datasets are 10 million points, and times reported are for $k = 1$.

from 0 to $n/2$ gives:

$$\begin{aligned} & k \int_0^{n/2} \left(\frac{\gamma^2}{\gamma^2 + 1} \right)^x x \, dx \\ < & \, k \int_0^{\infty} \left(\frac{\gamma^2}{\gamma^2 + 1} \right)^x x \, dx \\ & = O(k). \end{aligned}$$

Since the number of faces is constant, the overall bound follows. This shows that even in the worst case where every exterior cut is unbalanced and the maximum number of points are distributed in nodes formed from exterior cuts, the extra work incurred does not change the overall bound. □

We conclude the section with a note on the tradeoff between work and span. The versions of the theorems given in this section have a span that is greater than polylogarithmic due to the radix sort. If the algorithms used a comparison sort, they could run in polylogarithmic span at the cost of needing $O(n \log n)$ work for the sort.

3.3 Implementation Details

In this section we give more details on the practical implementation of both our algorithms and the other algorithms we use to benchmark our code.

3.3.1 Our Algorithms.

We implemented our algorithms in C++ using the parallel primitives from ParlayLib [49]. Our search implementation closely matches the algorithms shown in Section 3.2, so here we focus mostly on implementation details and other optimizations.

Numerical details. We work with double-precision floats, which we round to 64-bit integers for building the tree and computing the Morton ordering.

Miscellaneous optimizations. We mention a few optimizations that made significant differences in our runtime. Whenever possible, we used squared distances instead of Euclidian distances in our computations, which made our code about 10% faster. To store the current k-nearest neighbors when traversing the tree we use a vector for small k and the C++ STL priority queue for larger k. The overhead of the vector was significantly less for small k, but the linear instead of logarithmic cost dominates for $k > 40$ or so. A third optimization was to sort the sequence of queries using their Morton ordering so that nearby queries in this order access nearby nodes in the tree, thus reducing cache misses. The savings from reducing cache misses more than compensates for the cost of the sort, in some cases decreasing runtime by a factor of two. This is useful even when querying in parallel since the parallel scheduler processes chunks of the iteration space on the same core.

3.3.2 Other Implementations.

For the purpose of comparison we use three existing implementations of nearest neighbor search: CGAL [25], STANN [70], and Chan [55]. Here we describe some performance issues with their code, and some modifications we made to improve the performance of their code to ensure a fair comparison.

Chan. Chan's code was fully sequential so we needed to parallelize it. Conceptually this is relatively straightforward since the algorithm just requires using a parallel sort instead of a sequential one, and then running the queries in parallel. Chan's code only searches from the root of his implicit tree. We note that the root-based implementation of our code is significantly faster than Chan's.

STANN. STANN includes both a k-nearest neighbor graph (KNNG) function and a k-nearest neighbor (KNN) function. The first finds the k nearest neighbors among a set of points, and the second supports a function to build a tree and a separate function to query a point for its k nearest neighbors. They supply a parallel version of KNNG, that was parallelized with OpenMP, and only a sequential version of KNN. Their algorithm did not scale well beyond 16 threads, since it left some components sequential. We therefore updated their code to use the parallel primitives and built-in functions from ParlayLib [49]; this drastically improved their performance.

CGAL. CGAL implements a parallel version of their k-nearest neighbor code using the threading building blocks (TBB) [104]. We use their code directly with no modifications. We note that their code does not scale well past 16 or so threads. Furthermore, although the code appears to be thread safe, there seems to be contention when there are many threads, thereby slowing them all down. Due to the particularly bad performance beyond 36 threads (which all are on one chip), we only report numbers up to 36 threads. Furthermore, since we observed wildly varying times with higher k, we only included times for $k < 10$ in our experiments.

3.4 Experiments

In this section, we provide experimental results which show that 1) our algorithms perform well under many types of scaling and across different architectures, and 2) our algorithms outperform every implementation we test against.

3.4.1 Experimental Setup.

Machines. We ran most all of our experiments on a 72-core Dell R930 with 4x Intel(®) Xeon(®) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. With hyperthreading the total number of threads is 144. To check whether the results were robust across machines, we also ran one set of experiments on a 4-socket AMD machine with 32 physical cores in total, each running at 2.4 GHz, with 2-way hyperthreading, a 6MB L3 cache per socket, and 200 GB of main memory.

Test data. After testing our algorithms on some real-world image data, we discovered, similarly to Connor and Kumar [70], that uniformly random points perform very similarly to real-world datasets. To facilitate testing at various sizes we therefore use a few distributions of random points in 2 and 3 dimensions, on sizes up to 100 million points. The point sets we used are listed in Figure 3.3. The 2D and 3DinCube datasets are points picked uniformly at random in a square and cube, respectively. The points in the 3DonSphere dataset are selected on the 2D surface of a sphere in 3 space. This is meant to represent various graphics applications where the point sets are on a 2D surface embedded in 3D. The 3Dplummer distribution uses the Plummer model [21], which is based on the study of galaxies, and highly dense in at the center, becoming very sparse on the outside. The 2Dkuzmin distribution is a similarly skewed distribution in two dimensions.

As can be seen from the various statistics, the Plummer and Kuzmin distributions are significantly more skewed than the others. Indeed. these distributions do not have bounded expansion constant. The performance is slower due to the fact that a few small points are extremely far away from most of the points, which are in a dense cluster at the center. This causes the tree to be unbalanced and the searches for the nearest neighbors of these far points to be expensive. However the overall time is hardly affected by the skewed distribution for our leaf and bit-based implementations (Figure 3.3) showing the algorithms are robust under quite skewed distributions. As expected, the depth of the trees for the uniform distributions in a cube are just the logarithm of the number of points.

Algorithms tested. We ran three classes of experiments: (1) generating a k-nearest neighbor graph on a set of n points, (2) building a k-nearest neighbor query structure on n points followed by dynamic queries on a different set of n points, and (3) batch insertions for a total of n points (after the insertion).

Altogether we tested 9 variants of the algorithms: our parallelized version of Chan's algorithm, the CGAL algorithm, four variants of STANN (KNN, KNNG, parlayKNN and parlayKNNG), and three variants of our algorithm (leaf-based, root-based, and bit-based). The parlayKNN and parlayKNNG are our modified versions of Connor and Kumar's algorithms. Since

our modified versions are always significantly faster, we only report numbers for our versions.

For all implementations, we sort by Morton order before querying. This is to ensure that all algorithms are getting the same benefit of locality in the tree when querying. The experiments on batch insertion only use our algorithm since the others do not support dynamic updates.

3.4.2 Leaf vs. Root Based.

In Figure 3.3, we show the performance of our three search algorithms for finding the k-nearest neighbor graph on varying datasets and $k = 1$. Figure 3.3 shows the same result using a different measurement: the average and maximum number of nodes visited during a query. One takeaway from the figures is that even though the leaf-based method takes $O(n)$ work, and the bit-based method takes $O(n \log n)$ work, the prior is only slightly faster. This is because the constant in the $O(k \log k)$ term for a search from the leaf is much larger than the constant in the $O(\log n)$ search from the root. Another takeaway is that for the Kuzmin and Plummer distributions, when starting from the leaf using the root-based algorithm (Algorithm 3.3) makes an enormous difference.

3.4.3 k-Nearest Neighbor Graphs.

The results of our experiments for generating the k-nearest neighbor graph can be found in Figure 3.4.

Varying dataset size. (Figure 3.4(a) and (b).) We measured the total time per point (that is, to build the tree and perform the query) by dividing the total time to build and search by the number of points. As discussed in Section 3.3, we took measures to limit the number of cache misses where possible.

Work efficiency. (Figure 3.4(c) and (d).) Experiments showed that our algorithms performed significantly less work than our competitors as the number of threads increased to 144. To show that we maintain work efficiency on different architectures, we also ran the same experiments on a 32-core AMD machine.

Varying k. (Figure 3.4(e).) Our results on varying numbers of neighbors show that our algorithms remain fast and scalable.

Tree building. (Figure 3.4(f).) To illustrate that the tree-building step itself is efficient (except in the case of CGAL as explained in Section 3.3.2), we show time required to build the data structure for the 3DinCube and 3Dplummer distributions.

3.4.4 Dynamic Queries

The results from our experiments with dynamic queries are presented graphically in Figure 3.4.

Varying k, work efficiency. (Figure 3.5(a) and (b).) All the algorithms we tested had similar performance for work efficiency and scalability of the number of nearest neighbors as they did in the k-nearest neighbor graph building case.

Scaling size of dataset. (Figure 3.5(c).) Our algorithms are particularly robust compared to others when scaling the size of the dataset. One thing to notice in the relevant panel (c) of Figure 3.6 is that with the exception of CGAL, all algorithms experience a local minima when the size of the dataset reaches 10^5. While Chan, ParlayKNN, and CGAL's performances begin to slowly increase after this point, our algorithms' do not. The dataset of size 10^5 is approximately where we expect cache misses to start affecting the performance; as explained in Section 3.3, pre-sorting the data for dynamic queries helps alleviate this problem.

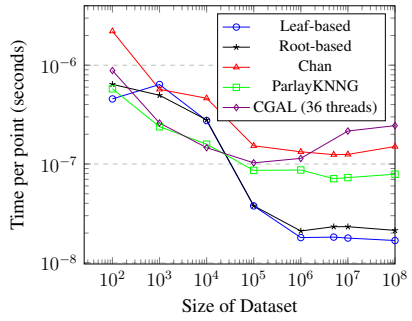
Other datasets. Since the bit-based and root-based algorithms performed very similarly on the random distribution for dynamic queries, we refer to Figure 3.4(f) for evidence that the bit-based algorithm outperforms the root-based for some distributions.

3.4.5 Dynamic Updates.

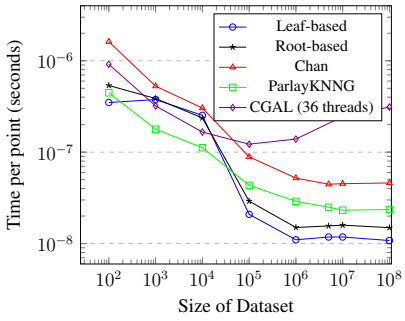
We test the efficiency of our batch-dynamic updates by measuring the time required per update as the number of updates in the batch increases. Figure 3.5 shows the time taken per point as the size of the batch increases, with both insertions and deletions shown. The figure shows a drastic change in time, spanning almost four orders of magnitude from 10^{-4} seconds for a single update to 10^{-8} seconds per update for a batch of 5 million. The first period of decrease as the batch size increases to 10^4 or 10^5 can be explained by parallelism—this is the point at which the parallel sort and the parallel recursion down the tree begin to save significant time. The fact that time continues to decrease even after the size grows large enough to see the full effects of parallelism can be attributed to the work efficiency of the batch-dynamic updates, as shown in Theorem 3.

3.4.6 Code availability.

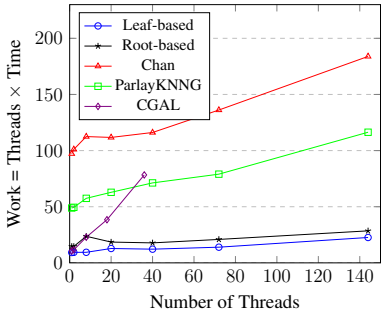
Our implementation is part of the publicly available Problem-Based Benchmark Suite [155].



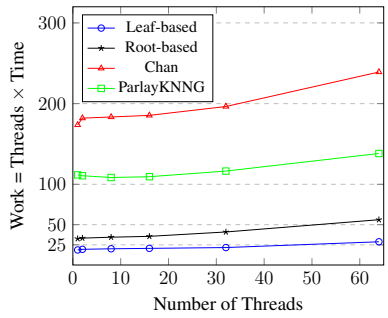
(a) Time required to calculate nearest neighbors as the size of the dataset increases. Calculated by dividing the total time by the number of points queried.



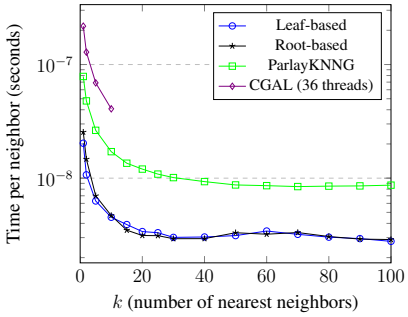
(b) The same as (a) but with a 2D dataset drawn randomly from a square instead of a 3D dataset.



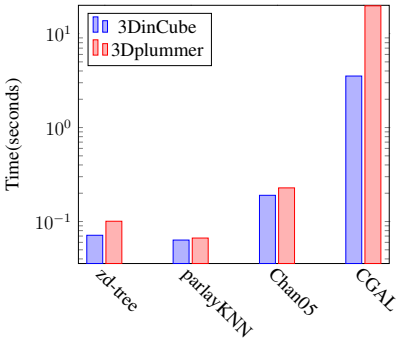
(c) Total work (threads $\times$ time) required to build a tree of 10 million points, then build the nearest neighbor graph of the point set. Shown as the number of threads vary.



(d) The same measurements as (c), but on the 32-core AMD machine.



(e) Time required to calculate a neighbor as the number of neighbors k increases. Calculated by dividing the total time by k times the number of queries.



(f) The bars represent the total time each algorithm takes to build the data structure, for points drawn randomly from a 3D cube, and points drawn from a Plummer distribution.

Figure 3.4: Statistics related to non-dynamic queries. Unless otherwise stated, the size of the dataset is 10 million, the number of nearest neighbors $k = 1$, experiments were performed on 144 threads on a 72-core Dell R930, and data points are drawn randomly from a 3D cube.

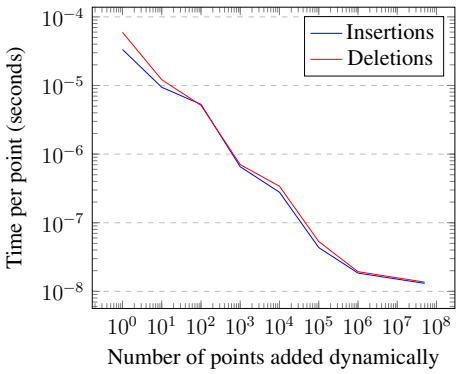
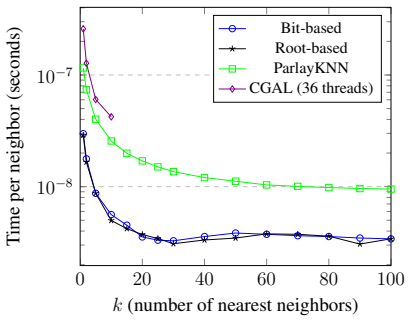
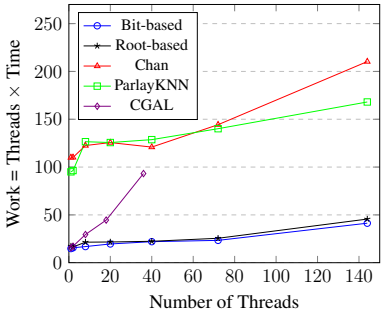


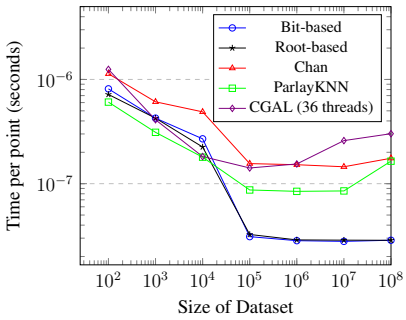
Figure 3.5: Time required per point as the number of points in the batch increases. Updates performed on a dataset of 10 million random points inside a 2D cube.



(a) Time required to calculate a neighbor as k increases. Calculated by dividing the total time by k times the number of queries.



(b) Total work (threads $\times$ time) required to build a tree of 10 million points, then dynamically query the same number of points, on varying numbers of threads.



(c) Time required to calculate nearest neighbors as the size of the dataset increases. Calculated by dividing the total time by the number of points queried.

Figure 3.6: Statistics related to dynamic queries. Unless otherwise stated, the size of the dataset is 10 million, 10 million dynamic queries were performed, the number of nearest neighbors $k = 1$, experiments were performed on 144 threads on a 72-core Dell R930, and data points are drawn randomly from a 3D cube.

Chapter 4

CLEANN: Lock-Free Augmented Trees for Exact k-Nearest Neighbor Search

4.1 Introduction

Concurrent data structures are crucial in environments in which updates and queries are arriving online and asynchronously from a variety of sources. In such a structure we typically expect the operations to be linearizable [99], i.e., that each appears as they happen atomically in some linear order. Designing correct concurrent data structures can be very difficult. Lock-free structures, ones that guarantee progress even if a process stalls or fails during its operation, can be particularly difficult.

Most concurrent structures only support single "point" updates and queries—that is, those that update or report on the property of a single point, such as finding a single key in a set. Supporting linearizable queries that consider multiple points has gained significant recent attention, especially for the purpose of range queries—i.e., returning all the keys in a sorted set within some range of key values [22, 30, 37, 40, 59, 118, 137, 175]. Such queries are more complicated than single point queries since they effectively have to get an atomic view of many points, even as some of those points are being updated. Many of the more interesting data structures, however, inherently require looking at multiple points.

Concurrent KNN structures. Chapter 3 addresses building static k-nearest neighbor data structures for the parallel setting, as well as extending those data structures to the parallel batch-dynamic setting. In addition to the zd-tree discussed in the previous chapter, most other work on parallel and dynamic k-nearest neighbor data structures uses some kind of tree structure, such as a kd-tree [43], and queries need to explore some potentially large part of the tree. Existing parallel or concurrent solutions for the dynamic case, including the zd-tree, have significant limitations. Some work is based on the batch parallel model [46, 180]. This requires applying updates in batches, during which time the data structure is not available for queries. Furthermore all updates need to be gathered together in quite large batches (10K+) to make them efficient. Other work supports arbitrarily interleaved concurrent updates and queries, but supports only

non-linearizable neighbor queries [102], or has been limited to finding the single nearest neighbor of a point [60]. In this second case, the approach is inherently limited to finding the single nearest neighbor.

Implementing concurrent KNN data structures is difficult for two reasons. Firstly the queries can require looking at many points in the data set, which might be changing. Secondly, for efficiency, interior nodes of the kd-trees often store augmented information (i.e. bounding boxes), which need to be propagated up the tree when making updates. Indeed neither of the concurrent approaches mentioned above use augmentation on the interior tree nodes.

The CLEANN-tree. In this chapter, we introduce the CLEANN-tree (CLEANN, for concurrent, linearizable, efficient, augmented nearest-neighbors), a concurrent linearizable lock-free dynamic KNN data structure. It supports insertions, deletions, k nearest neighbor queries, and range queries (i.e., report all points within a given radius). The CLEANN-tree is built on top of the zd-tree, as introduced in Chapter 3, which supports parallel batches of insertions or deletions but is not safe for concurrent operations. Its sequential behavior is identical to that of the zd-tree. From the zd-tree, it inherits the property of history independence (i.e. the tree shape depends only on its contents) and is therefore not susceptible to bad insertion orders, nor does it need internal rotations, and has various provable beneficial properties. It maintains bounding boxes for interior nodes—i.e., the smallest box that contain all points in the subtree.

To make the zd-tree safe for concurrent operations, the CLEANN-tree leverages recent work on lock-free locks [176] and snapshotting [47, 175], and develops new ideas on maintaining augmented bounding-box information. In our data structure, queries are wait-free (take bounded time), and updates (insertions and deletions) are lock-free.

Augmentation. We study two approaches for maintaining the bounding boxes on interior nodes of the CLEANN-tree that are safe for concurrency. The first approach precisely maintains exact boxes, but requires taking simultaneous "locks" on the path up from the leaf to the topmost interior node whose augmented value needs to modified. Since the locks are "lock-free", the approach is still lock-free, but might cause contention at higher nodes and affect the performance. We refer to this approach as *path-copying*. The second approach ensures a lower bound on the box but only requires holding two locks at a time by using hand-over-hand locking. The approach ensures the bounding box is at least as big as the true linearizable bounding box, which does not affect correctness of the search. Furthermore it can only be larger than the true box if there is an ongoing update within the subtree of the box. It works by expanding boxes before an insert linearizes, and contracting boxes after a delete linearizes.

We refer to augmentation that typically only needs to propagate a few layers up a tree as *shallow-effect augmentation*. We believe that both approaches could be of more general utility to other applications where shallow-effect augmentation is sufficient, with path-copying being more generally applicable as it is more precise. We briefly describe some such applications in Section 4.5.

Experiments. We implemented both the path-copying and hand-over-hand variants of the CLEANN-tree, and experimented with a variety of datasets and scenarios. We use both real-world datasets and synthetic datasets, including highly skewed synthetic datasets, with dimensionality from two to five and with sizes from 10 thousand to 1 billion. We test scalability on up to 144 hardware threads, and also with oversubscription where the number of software threads is larger than the number of hardware threads. We test with $k = 1$ and $k = 10$, as well as range queries. We

test with varying update rates from all searches to all updates. We test both the lock-free and blocking versions of our implementations. We test with random insertion order as well as a highly contended order. We compare to both a baseline sequential implementation (the original zd-tree), and to the concurrent LFkD-tree, which only supports $k = 1$.

The experiments show that the CLEANN-tree achieves near-linear speedup up to 72 cores compared to the sequential zd-tree algorithm, and scales well up to 1 billion points, and across the different data sets and scenarios (update rates, different dimensions, different k, and different insertion orders). The CLEANN-tree scales significantly better than the existing state-of-the-art for concurrent KNN data structures, the LFkD-tree, with respect to both number of cores and dataset size (even though LFkD only supports $k = 1$). With regards to our two variants we find that in most scenarios the path copying variant slightly outperforms the hand-over-hand variant, except under the skewed insertion order where the hand-over-hand noticeably outperforms the path copying variant. This indicates that hand-over-hand does help alleviate contention, but perhaps at a cost for other scenarios.

Contributions. The contributions of this chapter are:

- We describe the first lock-free concurrent KNN data structure. In addition queries are wait-free. We don't know of any non-trivial [1] prior work for general k and for linearizable operations even when not lock-free.
- We introduce two general approaches for maintaining augmented information on trees: hand-over-hand locking and path copying. Both can be applied to augmented kd-trees as well as more general tree data structures.
- We experimentally compare our two approaches both to each other and to prior work. The experiments show that the CLEANN-tree scales well to many cores, to large sizes, across different distributions, and in oversubscription when there are more software threads than hardware threads.

4.2 Background and Related Work

In this section, we cover some background on KNN searching and on the concurrency techniques we will leverage. Throughout this chapter, **point** refers to a d-vector of real numbers $(x_1, x_2, \ldots, x_d)$. The dimension d is assumed to be a small constant. All distances are Euclidean.

We assume the standard concurrent shared memory model with P asynchronous processes [98]. Algorithms in this model are required to be correct for an adversarial scheduler that can make a process arbitrarily slow or even crash (never take a step again). The correctness condition we consider is called linearizability [99], which means that each operation appears to take effect atomically at some point during its execution interval.

4.2.1 KNN structures

Almost all efficient data structures for KNN search in low dimensions (at or below ≈ 5), are based on hierarchical decompositions of space. Various structures differ in how the space is

[1] Excluding taking a global lock for every operation.

partitioned. Some use explicit trees, e.g. *kd-trees* or *oct-trees* and some use space-filling curves, such as the *morton-ordering*. Among the tree structures there is again variance on whether the points are kept balanced or the space is kept balanced, and on what the branching factor is. Furthermore, some maintain a tight bounding box around the points in each node of the tree, which can be significantly smaller than the partition itself. Searching for the k nearest neighbors of a point p in such structures consist of first searching for a leaf with a nearby point (traversing from the root in an explicit tree or using binary search on a space filling curve), and then moving up the tree keeping track of the current k nearest points and pruning any spatial partitions that cannot contain a point closer than the current k-th nearest point. Recent work [46] has shown that maintaining a tight bounding box significantly improves performance since it more accurately prunes the search.

Our approach should work on most tree-based decompositions, although we use it on a zd-tree [46], described next. We use the zd-tree both because it seems to be the state-of-the-art in performance for sequential and parallel trees but also because the partition is based on splitting space so no rebalancing is required when inserting and deleting points as long as the bounding box for the root is kept fixed.

The zd-tree. The zd-tree is a hybrid of a space filling curve and a binary tree. The splitting rule uses the *Morton ordering* [70], or *z-ordering*, of the point set. For a set of points whose coordinates are d-vectors of integers $(x_1, x_2, \ldots, x_d)$, the Morton ordering is calculated by taking each integer coordinate in binary form, interleaving the coordinates to create one integer per point, then sorting using the interleave integers. The zd-tree is constructed by computing the Morton ordering of a point set by interleaving the bits of each coordinate, then building a binary kd-tree at each level by recursively splitting the point set on its most significant bit. The information contained in a zd-tree node is shown in Algorithm 4.1; the parts marked in red are the modifications necessary to make the zd-tree support concurrent queries and updates, which will be explained in detail in Section 4.3.

The zd-tree uses the standard search algorithm on a kd-tree: first, it finds a leaf in the tree using a heuristic (shown in Algorithm 4.2 as `search_heuristic()`). It then uses the k nearest points within the leaf as an initial approximation to the k-nearest points to the search point q. This initial approximation defines a ball B of radius r centered at q, where r is the distance between q and its furthest such neighbor.[2] Then, it traverses back up the recursion stack and searches the siblings and their descendants whose bounding box intersects with B (shown as the function `within_epsilon_box()` on Line 10). Each time a point is found within the ball, the approximate ball is updated by replacing the furthest point with the one that is found, and hence shrinking the radius.

A critical observation about Algorithm 4.2 is that it correctly finds the k-nearest neighbors of q *regardless of the choice of search heuristic* since it has examined all points within the final ball, and the result is the closest of them. Another observation is that pruning nodes that do not intersect the current ball is very effective, both in practice and in theory. In theory it can be shown, under certain assumptions of the point distribution, that the number of nodes that are examined after the initial leaf is found is constant in expectation [46].

[2]If there are fewer than k points in the leaf it includes them all, and uses a radius of infinity.

```
struct node : verlib::versioned {
  int bit;
  flck::atomic<bool> removed;
  flck::lock lck;
  sequence<point> points; // empty if internal node
  verlib::versioned_pointer<node> L; // null if leaf
  verlib::versioned_pointer<node> R; // node* if zd-tree
  pair<point, point> bbox;
  point center; };
```

Algorithm 4.1: Zd-Tree node.

```
void search(point p, int k, node* T){
  verlib::with_snapshot([&], {
    double r = infty;
    priority_queue Q; // priority is distance from p
    search_rec(T, p, k, r, Q);
    });}

void search_rec(node* T, point p, int k,
double r, priority_queue Q){
  if(within_epsilon_box(T, p, r)){
    if(T->is_leaf()){
      sequence<point> P = T->points;
      for(point q : P){
        if(dist(p,q) < r){
          if(Q.size() == k) Q.pop();
          Q.push(q);
          r = dist(p,q);
    }}} else if(search_heuristic(T) == T->Left()){
      search_rec(T->Left(), p, k, r, Q);
      search_rec(T->Right(), p, k, r, Q);
    } else{
      search_rec(T->Right(), p, k, r, Q);
      search_rec(T->Left(), p, k, r, Q);
}}}
```

Algorithm 4.2: Searching for k-nearest neighbors on a zd-tree; the red lines are the additional code required for the CLEANN-tree.

```
1   void insert(point p, node* T){
2     flck::with_epoch([&], {
3       size_t il = compute_interleave_integer(p);
4       while(true){
5         if(insert_rec(p, il, T, nullptr, false)) break;
6     }});}

8   bool insert_rec(point p, size_t il, node* T, node* parent, bool locking_mode){
9     if((T->is_leaf() || !within_box(p, T->bbox)) &&
10    !locking_mode){
11      return parent->lck.try_lock([&], {
12        if(parent->removed) return false;
13        if(parent->child != T) return false;
14        else return insert_rec(p, il, T, parent, true)
15        });}
16    if(T->is_leaf()){ // insert into leaf
17      sequence<point> points = T->points;
18      points.push_back(p);
19      node* N;
20      if(points.size() > nodeSizeCutoff){
21        sort(points); \\ sort by interleave ints
22        left_pts, right_pts = split(points);
23        node* L = flck::New<node>(left_pts, T->bit+1);
24        node* R = flck::New<node>(right_pts, T->bit+1);
25        N = flck::New<node>(L, R, T->bit);
26      } else N = flck::New<node>(points, T->bit);
27      parent->set_child(N);
28      flck::Retire(T); // free in zd-tree
29      return true;
30    } else{ // insert into internal node
31      if(!within_box(p, T->bbox)){
32        return T->lck.with_lock([=] {
33          // need to correct T's bounding box
34          box new_b = construct_box(p, T->bbox);
35          node* N = flck::New<node>(T->Left(), T->Right(), T->bit, new_b);
36          return N->lck.with_lock ( [=] {
37            parent->set_child(N);
38            T->removed = true;
39            parent->lck.unlock(); T->lck.unlock();
40            flck::Retire(T); // free in zd-tree
41            if(il[N->bit]==0) return insert_rec(p, il,
42            N->Left(), parent, locking_mode);
43            else return insert_rec(p, il,
44            N->Right(), parent, locking_mode);
45            });
46          });
47      } else{
48        if(il[T->bit]==0) return insert_rec(p, il,
49        T->Left(), parent, locking_mode);
50        else return insert_rec(p, il, T->Right(), parent, locking_mode);
51  }}}
```

Algorithm 4.3: Inserting a point into a zd-tree using hand-over-hand locking, with lines in red showing modifications made in the CLEANN-tree. Some edge cases, such as parents and children with nonconsecutive bit values and insertion of duplicate points, are excluded from this pseudocode for ease of exposition.

Inserting a point into the zd-tree follows a similar routine to its building algorithm. Since the splits of the tree are on the most significant bit, there is no need to rebalance the tree; thus the insert procedure can simply recurse to the leaf it would be contained in and insert the point into the leaf, splitting the leaf to form an internal node with two children if the leaf size cutoff is exceeded. Furthermore, inserting a point may require enlarging bounding boxes of internal nodes, which is performed during the search for the leaf. Pseudocode for the algorithm can be found in Algorithm 4.3.

4.2.2 Concurrent KNN Datastructures

As far as we know, there are only two existing data structures for concurrent nearest neighbor search. The concurrent kd-tree of Ichnowski and Alterovitz [102] supports linearizable insertions and wait-free (but non-linearizable) k-nearest neighbor queries. It does not support deletions.

The LFkD-tree [60] is a concurrent kd-tree which supports linearizable insertions, deletions, and single nearest neighbor queries, but not k-nearest neighbor queries for arbitrary k. The sequential data structure the authors adapt is a kd-tree with single-point leaves and internal nodes representing splitting hyperplanes. The splitting hyperplanes are calculated ad-hoc; i.e. when the tree is updated sequentially, a hyperplane is chosen that separates the first two points, then points are inserted into the tree by following a root-to-leaf path based on splitting hyperplanes, and splitting the leaf node they encounter. A single point insertion or deletion will thus create changes only to an existing leaf node and its parent, so the algorithm traverses the tree to this leaf and performs lock-free and linearizable updates using compare-and-swap operations in a similar manner to the Natarajan-Mittal tree [136]. This means that certain insertion orders will make the tree very unbalanced—indeed inserting in z-order will effectively make it a linked list.

To linearize queries and updates, the LFkD-tree maintains a single global linked list of all currently active queries and their current nearest neighbor in the data structure. When a node is inserted, it scans all existing queries to see if it is the closest neighbor of any of them, and if so, reports itself as such, making its linearization point before that of the query. Note that this allows for reporting a single nearest neighbor but cannot generalize to even two neighbors since there is no place to put the linearization point to make two points consistent. Also, as our (and their) experiments show, it does not scale well due to contention on the list of active queries.

4.2.3 Optimistic Locking

We next describe the locking techniques used in the design of the CLEANN-tree. For many search tree data structures, it has been observed that traversing from the root to a leaf can be done optimistically without taking any locks [93, 97, 119]. To update the tree, one simply needs to lock a small neighborhood around the nodes that need to be changed. It is possible that these nodes could have changed or been removed by a concurrent process between the traversal and the locking phase, so a validation step is performed after all the locks are taken to ensure this is not the case. If the validation fails, the update operation restarts, and otherwise, it makes the desired changes to those nodes and then unlocks the locks it acquired. This approach is supplemented by using two different locking techniques: a *try-lock*, which tries once to acquire the lock and fails if

```
1   // a versioned pointer to object of type T
2   struct versioned_ptr<T> {
3    versioned_ptr(T v); // constructor with value v
4    T load();         // read the value
5    void store(T v);}; // store a new value

7   // inherited in objects pointed to by versioned pointers
8   struct versioned;

10  // function f applied on an atomic snapshot
11  // where R is the return type of f
12  R with_snapshot(F f);

14  T* flck::New<T>(args); // idempotent memory allocation
15  void flck::Retire<T>(T*);
16  T flck::with_epoch(F f);

18  struct flck::atomic<T>;

20  struct flck::lock {
21   bool try_lock<F> (F f);
22   T with_lock<F> (F f);
23   void unlock(); }
```

Algorithm 4.4: The VERLIB interface. C++ template declarations for `F` and `T` left out.

it cannot acquire it, and a *with-lock*, which keeps trying to acquire the lock until it succeeds. Try-locks are used to acquire a lock on the internal nodes of the tree during an optimistic traversal, with the idea being that if the traversal tries to take a lock that is already held, it is very likely that it would fail validation after acquiring the lock. With-lock is used at the leaf after a try-lock has succeeded on one of its ancestors, and since the operation can no longer fail.

This design pattern is sometimes referred to as *optimistic locking*, and it has been observed to give very good performance for a wide range of lock-based data structures [42, 93]. We follow this design pattern in the CLEANN-tree. In Algorithm 4.3 the validation steps can be seen in lines 12 and 13, where after trying a lock, we check that the parent has not been removed and that the child of the parent has not changed.

4.2.4 Lock-Freedom and Snapshotting with Verlib

In this section we introduce the VERLIB library [47], a collection of techniques for snapshotting, lock-freedom, and memory management that we use to make the CLEANN-tree lock-free and linearizable. An overview of the supported operations can be found in Algorithm 4.4.

Snapshotting. To ensure that nearest neighbor queries on our CLEANN-tree are linearizable, we use an approach for snapshotting concurrent data structures called *versioned pointers* [47, 175], which is implemented using version lists [140, 148]. At a high level, VERLIB's versioned pointers ensure that a nearest neighbor query starting at time t is linearizable by keeping a snapshot of each node at time t accessible until all such queries have completed. When a node is updated at time $t+i$, the query reads the version from time t, thus ensuring linearizability. The approach can be applied to any concurrent data structure as long as all mutable variables are pointers. To support snapshots, the user replaces all pointers they want to include in a snapshot

with versioned pointers (`verlib::versioned_ptr`). Versioned pointers support standard reads (`load`) and writes (`stores`). In addition VERLIB supplies a `with_snapshot` operation that can be wrapped around code. The code within the wrapper is then all read as if at an atomic point in time. The time for loads and stores are constant, if not within a `with_snapshot`. Therefore the asymptotic time of a concurrent data structure is not affected by using a versioned pointer, and taking a snapshot is cheap. The time for each loads within a `with_snapshot` is at most proportional to the number of update operations on the pointer since the start of the snapshot (i.e., the depth of the desired version in the version list). Normally version lists require a level-of-indirection for every pointer reference but the approach used in VERLIB avoids this in the common case.

The query algorithm uses `with_snapshot` to ensure linearizable queries on Line 2 of Algorithm 4.2. In Lines 1, 6, and 7 of the Zd-Tree node description in Algorithm 4.1 show the use of versioned pointers.

Lock-free Locks. To make the CLEANN-tree lock-free, we use the concept of *lock-free locks* (which can be used alongside versioned pointers). While this sounds contradictory, we note that lock-freedom is a technical term meaning that a stalled thread will not prevent other threads from making progress, while a "lock" is a programming construct indicating the expectation of mutual exclusion of critical code among threads that acquire the same lock. The high-level idea behind lock-free locks is that a thread can help other threads complete their critical code if they have a lock that it needs, and hence release the lock. The approach, described by Ben-David, Blelloch and Wei (BBW) [41] and implemented in a library called flock, works on nested locks (locks taken while holding other locks) as long as there are no lock cycles—i.e., no deadlocks using standard locks. The approach does not require any changes to the user code beyond using the flock version of shared variables (`atomic`) the flock version of locks (`try_lock` and `with_lock`) and flock memory allocation (`alloc`). The use of the atomic is shown in Algorithm 4.1, line 3. Locks are used in Algorithm 4.3 lines 11, 32 and 36.

Memory Reclamation. The flock library also implements an epoch based reclamation scheme [83] for safe memory reclamation. It supports a `with_epoch` function that takes as input a block of code and protects all memory locations that could be potentially accessed by that block of code from being freed. It also supports a `retire` function that safely frees a memory location once it is no longer protected by any `with_epoch`.

4.3 The CLEANN-tree

The CLEANN-tree is a concurrent version of the zd-tree introduced in Section 4.2.1. It supports concurrent inserts, deletes, and KNN queries in a linearizable and lock-free manner. One of the main challenges of this work is in correctly and efficiently maintaining the augmented values in internal nodes, specifically the bounding boxes. We introduce two solutions for this problem in Sections 4.3.1 and 4.3.2 which lead to two variations of our algorithm. The first solution borrows ideas from functional data structures and path copying and the second uses hand-over-

hand locking and exploits the property that bounding boxes can temporarily be too big without affecting the correctness of searches.

Code for the CLEANN-tree is shown in Algorithms 4.1–4.3 with the lines highlighted in red representing the changes we made for concurrency. We explain these changes in this section, starting with ones that are common to both the path copying and hand-over-hand locking versions of the data structure. Both versions share the same node definition (Algorithm 4.1) and KNN search function (Algorithm 4.2), but differ in their insert and delete functions. Algorithm 4.3 shows how to insert into the hand-over-hand version.

We begin by noting that, even though the CLEANN-tree uses locks, the data structure is still lock-free through the use of the FLOCK library introduced in Section 4.2.4. To minimize the number of locks taken, the traversal phase of each insert and delete operation first proceeds optimistically without taking any locks and then switches into locking mode when it encounters the first node that needs to be updated. This could either be because the node's bounding box needs to be updated or because a point needs to be inserted or removed at the node. In Algorithm 4.3, the insert function switches into locking mode when the checks on line 9 passes. After switching into locking mode, both insert and delete lock the parent of the node they wish to change and perform some validation checks (Lines 12-13). At each point in time, we define the *search path* of a traversal as the path the traversal would have followed if it started from the root and ran in isolation until it reached a leaf. These validation checks ensure that, after the locks are taken, the parent and the current node are on the search path of the traversal. This property helps ensure linearizability of update operations. If the validation checks fail, then the traversal restarts from the root and returns back to optimistic mode. This can only happen if another insert or delete operation linearized during the traversal. Since an operation can only restart if another made progress, lock-freedom is still guaranteed.

After the validation checks pass, the traversal locks the remainder of the path to the leaf. The nodes along this path are guaranteed to be on the search path and no further validation checks are needed. At this point, the path-copying and hand-over-hand version of the data structure diverge and we describe them in the next two subsections. One thing that both have in common is that they linearize insert and delete operations at the pointer swing that adds or removes the desired point from the search path.

Linearizability for k-NN searches is achieved by running the zd-tree search algorithm on a snapshot of the data structure state. To support snapshotting, we use the VERLIB library described in Section 4.2.4 by wrapping the left and right child pointers of each node in `versioned_pointer` objects (Lines 6-7 in Algorithm 4.1). To avoid having to use multi-versioning for bounding boxes and other routing information inside each node, we make those fields immutable and have update operations create new copies of nodes. This way, taking a snapshot of all the `versioned_pointer`s also gives a snapshot of the bounding boxes and routing information in the data structure. The update operations we will present in Sections 4.3.1 and 4.3.2 maintain the property that each snapshot is always a valid zd-tree with bounding boxes at least as big as the exact bounding box containing the points in each subtree. We prove in Proposition 1 that this is sufficient to ensure correctness of the zd-tree search algorithm

Proposition 1. *Let T' be a zd-tree built on point set P. Then, for each node N_i of T', let B_i be the exact bounding box containing all of N_i's descendants, and let B'_i be the bounding box stored at*

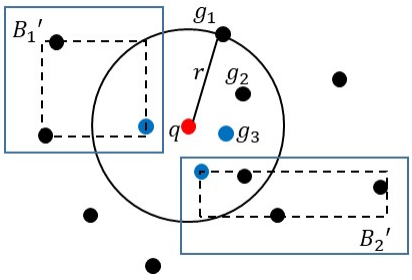


Figure 4.1: An illustration of searching for the 3-nearest neighbors of the query point q, which is marked in red, with its nearest neighbors marked in blue. Points g_1, g_2, g_3 denote the initial guess for q's nearest neighbors. Boxes B_1', B_2' denote bounding boxes of tree nodes, while the boxes drawn with a dotted line show the exact bounding boxes of the point set in the node. Note that all three true nearest neighbors of q are within radius r of q, and that the ball of radius r overlaps with the two bounding boxes containing the remaining neighbors of q.

node N_i. Then if for all N_i, $B_i \subseteq B_i'$, a k-nearest neighbor search on T following Algorithm 4.2 will return the true k-nearest neighbors.

Proof. Let T be the zd-tree built on point set P with exact bounding boxes, and T' be the zd-tree on point set P with boxes that strictly contain the exact bounding boxes. We will show that, if the set of leaf nodes containing the k-nearest neighbors of point q are visited when searching on T, they must also be visited when searching on T'. This is a sufficient condition for correctness, since (a) the search algorithm is known to be correct on a tree with exact bounding boxes [46], and (b) if a leaf node containing a true k-nearest neighbor of q is visited, then that neighbor is guaranteed to be returned since it will never be removed from the queue.

Now we will show that, if a leaf node L containing one of q's k-nearest neighbors is visited when searching T, it must also be visited when searching on T'. Consider the stage of searching on T' such that L has not yet been visited. Then the ball surrounding the current best k points must have a radius r' such that r' is greater than the true distance between q and it's kth-nearest neighbor. This implies that each bounding box on the root-to-leaf path to L on T must intersect with $B_{r'}(q)$, the ball of radius r' around q. Thus, no matter where the `search()` algorithm is in the recursion stack, it must eventually encounter an ancestor A of L and follow the path from A to L. Now, consider the case where the bounding boxes on the root-to-leaf path all strictly contain the exact bounding boxes: these boxes would all still intersect with $B_{r'}(q)$ since the box strictly contained in them also intersects $B_{r'}(q)$. Thus L would also be visited when searching on T'. See Figure 4.1 for an illustration. □

4.3.1 Path-Copying

We can ensure that the bounding boxes in each snapshot are exact by borrowing a technique from functional data structures called path copying [114, 138]. After an insert or delete operation identifies the first node whose bounding box needs to change and enters locking mode, it locks the entire path starting from the parent p of this node to the leaf. This prevents any concurrent

operation from changing these nodes. Then it makes a new copy of this path with the updated bounding boxes and a new leaf node, and installs the new path atomically by changing p's child pointer to point to it. The insert or delete operation is linearized when it changes p's child pointer. It then marks the nodes along the old path as removed and retires them for memory reclamation before finally releasing all the locks.

One observation is that, when an insert operation needs to expand a bounding box, it also needs to expand the bounding boxes of all the nodes below it on the way to the leaf. Therefore, the insert operation locks precisely the nodes that it needs to change. For a delete operation, we start locking when the point being deleted lies on the edge of the bounding box. This does not necessarily mean that the bounding box needs shrink—there could be another point along the same edge. However, there is no way to distinguish between these two cases as the delete is traversing down the tree so we need to conservatively start locking from this point.

We find that the path that needs to be updated on each insert or delete is usually short because the chance of lying outside a bounding box or on the edge of a bounding box shrinks drastically for nodes closer to the root. This circumvents one of the drawbacks of traditional path-copying data structures which require copying the entire path to the root and thus introduce significant contention, as every update must then sequentialize over the root. The limited path copying we perform near the leaf of the tree allows for a high degree of concurrency among update operations.

4.3.2 Hand-Over-Hand Locking

A disadvantage of the path copying approach from Section 4.3.1 is that the locks along a copied path need to be held simultaneously. These locks need to be held until the path is copied, so if the path is long, they could be held for a long time, leading to reduced concurrency. Hand-over-hand locking alleviates this problem by ensuring that an updating process only holds onto two locks at once. However, using this approach requires updating bounding boxes one at a time, so bounding boxes cannot be kept consistent with each other. We have shown in Proposition 1 that the correctness of KNN queries is maintained as long as bounding boxes are at least as large as the true exact bounding box—that is, bounding boxes can be allowed to temporarily be too large. We can maintain this property as long as we preemptively expand bounding boxes on inserts as we traverse down the tree and post-operatively shrink bounding boxes on deletes in the opposite direction (from leaf to root). The code for implementing this is shown in Algorithm 4.3. It is similar to the path-copying version except with nodes being copied one at a time and with locks being released in a sliding window manner as an update traverses down the tree (Line 39).

The importance of maintaining exact bounding boxes is that it reduces the runtime of K-NN queries by allowing them to prune more aggressively. For efficiency, the hand-over-hand version of our algorithm ensures that all the bounding boxes are exact if there are no updates in progress and that bounding boxes can only be temporarily out of date along the path of an insert or a delete.

We see in our experimental evaluation that the early unlocking from this hand-over-hand approach has significant advantages in skewed workloads where update paths of concurrent operations are likely to overlap. However, in workloads where this does not happen, the added complexity and cost of copying nodes one at a time lead to a slight performance decrease relative

to the path copying version of the algorithm. We believe that both versions can be generalized to maintain augmented data beyond just bounding boxes. It should be applicable to any tree structure where the augmented data typically only changes close to the leaves and rarely propagates to the root. The hand-over-hand solution also requires that there is a partial order over the augmented data and that queries are still correct even if the augmented values are larger than they should be in this partial order.

4.4 Experiments

In this section we introduce our implementation and experimental setup, and provide experimental results on the performance of the CLEANN-tree.

4.4.1 Implementation

In this section we provide details on our implementation as well as our experimental setup. Our algorithms are implemented in C++ using parallel primitives and other parallel tools from ParlayLib [49] and VERLIB. The implementation closely matches the pseudocode shown in Section 4.2. We work with double-precision floats, which we round to 64-bit integers for building the CLEANN-tree and computing the Morton ordering.

Machine. Our experiments ran on a 72-core Dell R930 with 4x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1 TB main memory running Ubuntu 20.04 LTS. Each core is 2-way hyper-threaded giving 144 hyperthreads. The machine's interconnection layout is fully connected so all four sockets are equidistant. We interleave memory across sockets using `numactl -i all`.

Datasets. We use a mix of synthetic and real-world datasets in our experiments. For synthetic datasets, we use a 3D dataset generated by picking points uniformly at random from a cube (3DinCube), as well as the 3Dplummer dataset, which is generated from the Plummer model of galaxies [21], and is highly dense at the center with many sparse outliers. For real-world datasets, we use the Lucy and Thai Statue datasets from the Stanford 3D Scanning Repository [15]. Lucy consists of 14 million points from a 3D scan of an angel sculpture, while the Thai statue consists of 5 million points from a 3D scan of a Thai statue. Characteristics of the datasets are reported in Figure 4.1, and visualizations of some of the datasets are shown in Figure 4.2.

Code Availability. After the process of anonymous review is completed, we will release our code publicly.

4.4.2 Experimental Setup

In this section we present details on the variants of the CLEANN-tree used in our experiments, and provide details on the other implementations we consider testing against.

Dataset	Size (Millions)	Dim	$\log \frac{d_{\max}}{d_{\min}}$
3DinCube	NA	3	15.93
3DPlummer	NA	3	27.96
Lucy	14	3	25.90
Thai Statue	5	3	20.24

Table 4.1: Characteristics of the datasets used in our experiments. The first two are synthetic (uniform in 3D cube, and Plummer distribution [21] in 3D), and the size can be varied. The last two are from Stanford 3D Scanning Repository [15]. We show the logarithm of the maximum distance between two points divided by the minimum distance with respect to the 10 million size versions of the synthetic datasets.

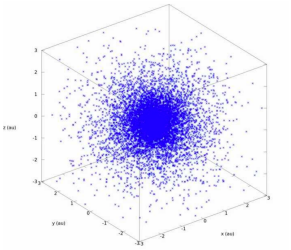
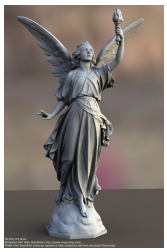




Figure 4.2: Visualizations of some of the datasets used in the experiments. From left to right: a visualization of the Plummer model [78], the Lucy statue from the Stanford 3D Scanning Repository, and the Thai statue from the Stanford 3D Scanning Repository [15].

CLEANN-Tree Variants. As described in Section 4.3, we test two main algorithmic variants of the insertion and deletion procedures: hand-over-hand locking and path copying. In addition to these variants on the update algorithm, we also test these variants using both traditional locks and lock-free locks [41]. The reason for including both traditional locks and lock-free locks is that lock-free locks tend to provide superior performance in conditions of oversubscription (that is, more software threads than hardware threads), but they incur a small overhead in other cases. Hence, we use the lock-free versions in experiments that include oversubscription of software threads, and the traditional locking variants otherwise. The total set of variants used in our experiments is as follows: **CLEANN-Tree-HOH**: hand-over-hand locking with traditional locks; **CLEANN-Tree-HOH-LF**: hand-over-hand with lock-free locks; **CLEANN-Tree-PC**: path copying with traditional locks; **CLEANN-Tree-PC-LF**: path copying with lock-free locks.

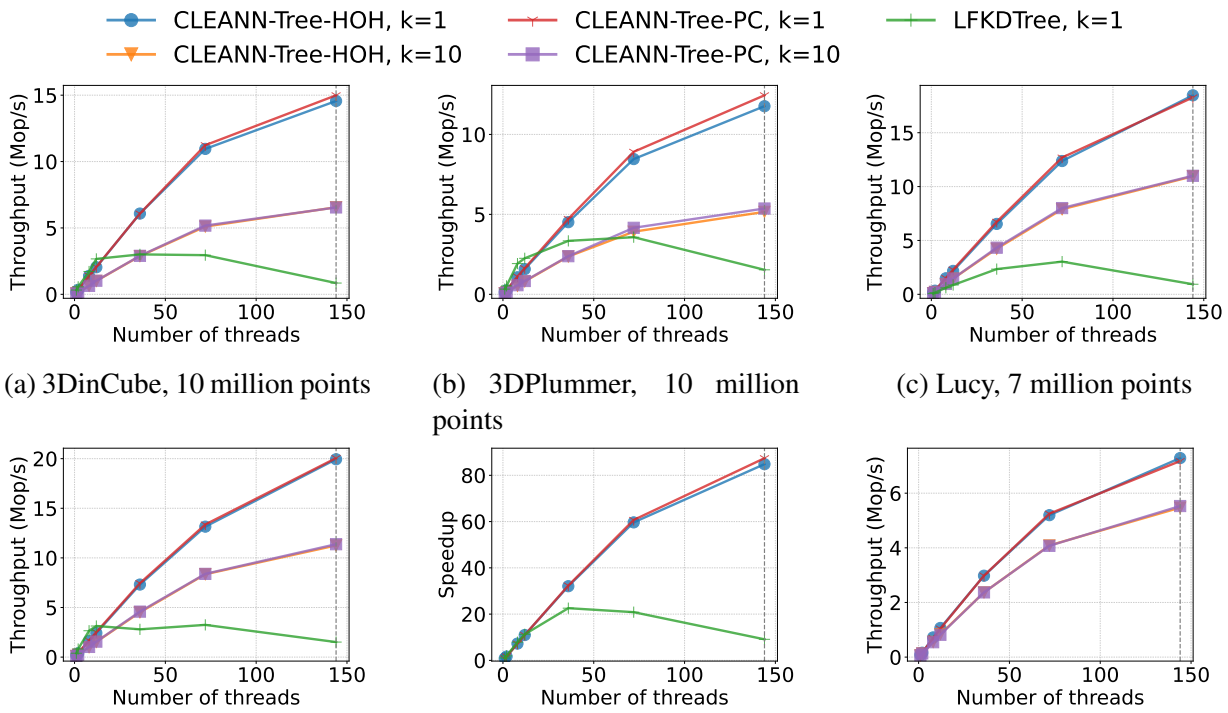
Other Concurrent KNN Structures. As mentioned in Section 4.1, to the best of our knowledge there are only two existing lock-free data structures for nearest neighbor search. The LockFree-kD-Tree (LFkD-tree) [60] is a lock-free and linearizable kd-tree which supports insertions, deletions, and single nearest neighbor queries. The LFkD-Tree does not support linearizable k-nearest neighbor queries for arbitrary k, and this limitation appears to be inherent to the authors' techniques. The authors provide a public implementation of the LFkD-tree written in Java [58], which we use in our experiments.

The concurrent kd-tree implementation of Ichnowski and Alterovitz [102] supports linearizable insertions and wait-free (but non-linearizable) k-nearest neighbor queries. It does not support deletions, making it impossible to run in our framework, which requires the data structure to maintain a steady size throughout experiments. Furthermore, non-linearizable k-nearest neighbor queries are significantly easier than linearizable ones, making any comparison somewhat unfair. Due to these significant differences from the CLEANN-tree, we do not include it in our evaluation.

4.4.3 Experimental Results

We now present experiments on throughput, parallel speedup, and other measures of scaling on various datasets and workloads. The experimental results support the following conclusions:

- The CLEANN-tree achieves good throughput on a variety of datasets, workload patterns, and numbers of cores.
- The CLEANN-tree is scalable to a large number of cores with near-linear speedup.
- The CLEANN-tree is capable of scaling to datasets up to one billion data points.
- The CLEANN-tree scales significantly better than the existing state-of-the-art for concurrent KNN data structures, the LFkD-tree, with respect to both number of cores and dataset size.
- The path copying and hand-over-hand locking approaches give similar results, but path copying performs slightly better on high-update workloads, especially on skewed datasets, and hand-over-hand performs slightly better in when threads are working on a localized subset of the tree.



(a) 3DinCube, 10 million points

(b) 3DPlummer, 10 million points

(c) Lucy, 7 million points

(d) Thai statue, 2.5 million points

(e) 3DinCube, 10 million points. Speedup is with respect to the best known sequential implementation, the zd-tree [46].

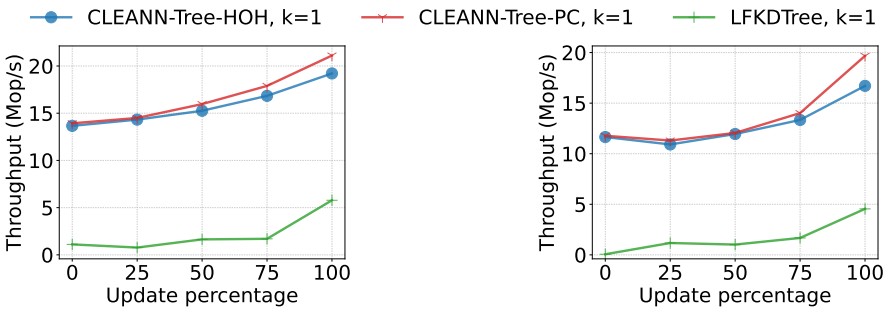
(f) Range search, 3DPlummer dataset, 10% updates. Here k denotes the average number of range candidates returned.

Figure 4.3: Figures measuring throughput and/or parallel speedup as the number of threads ranges from 1 to 144. Unless otherwise mentioned, the experiment is conducted using nearest neighbor search with a 50% update-to-query ratio. The size given in the caption refers to the steady-state size of the CLEANN-tree used.

- Lock-free locks perform better under oversubscription, and the effect is stronger when the workload has locality.

Setup. In all our experiments, we first build an initial tree with a specified number of points, with the points inserted in random order. Then, we use a workload with a mix of queries, insertions, and deletions and measure throughput over an interval of 10 seconds. To keep the size of the tree constant throughout the experiment, the percentages of insertions and deletions are always kept equal, and thus we refer to a singular update percentage throughout the section. The focus of the experiments is on nearest neighbor queries, but we also include one experiment on range queries to demonstrate the CLEANN-tree's capability to answer them efficiently.

Throughput on Various Datasets. We first test the throughput of several variants of the CLEANN-tree on each of the four datasets discussed in Section 4.4.1. These experiments are shown in Figure 4.3. We test both the hand-over-hand unlocking and the path copying versions under a



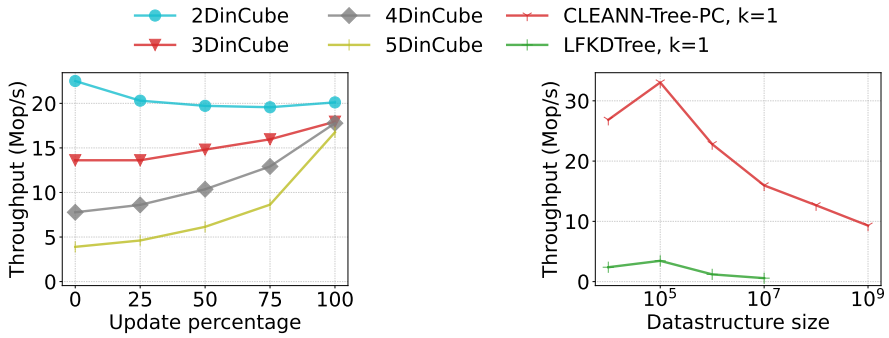
(a) 3DinCube (b) 3DPlummer

Figure 4.4: Experiments measuring the effect of update rate on throughput. Both experiments run with 144 threads.

workload of 50% updates and 50% queries, and we test queries for both $k = 1$ and $k = 10$. Since the LFkD-tree does not support queries for $k > 1$, the LFkD-tree is only shown for $k = 1$. Since the CLEANN-tree also supports range searching for all candidates fixed radius, in Figure 4.3f we report throughput numbers for range searching. Since each query or update is executed right away on its assigned thread without any batching, average latency is the number of processors divided by the throughput.

In these experiments, we find that overall the CLEANN-tree supports search and updates at high throughput, achieving anywhere from 10 to 20 million operations per second on 144 threads depending on the dataset and workload. It also achieves near-linear speedup (compared to its single-threaded throughput) on 72 threads and around 90x speedup at 144 threads (using hyperthreading). On the other hand, the LFkD-tree shows good performance up to 12 threads but experiences rapid performance decline at 36 threads and higher. There are at least two significant reasons for the performance deficits of the LFkD-tree; the first is that queries are highly contended, as their mechanism for ensuring that queries are linearizable requires adding queries to the tail of a global linked list. Secondly, the LFkD-tree stores a single point in each leaf while the CLEANN-tree stores a batch of points, meaning that more of our data structure fits into the L3 cache of the machine. In some cases, the LFkD-tree outperforms the CLEANN-tree on smaller thread counts; however, most of these experiments were conducted on a random ordering of the dataset and thus the LFkD-tree picked close to best-case splitting hyperplanes. In the case where the input is ordered, such as in the Lucy dataset shown in Figure 4.3c, the performance of the LFkD-tree declines below that of the CLEANN-tree on all thread counts.

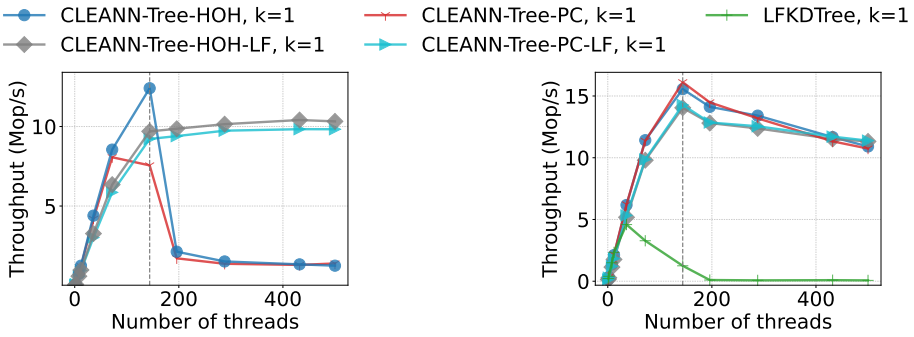
Parallel Speedup. To measure the parallel speedup of the CLEANN-tree and the LFkD-Tree, we compare both data structures to the best known sequential algorithm for computing nearest neighbors with insertions and deletions, the zd-tree, by measuring their speedup with respect to the single-threaded throughput of the zd-tree. We find that the CLEANN-tree achieves 60x speedup compared to the zd-tree on 72 threads, and up to 85x speedup on 144 threads (with hyperthreading), while the LFkD-tree's speedup bottoms out at about 20 threads.



(a) Dimension scaling on data drawn from a hypercube of 2, 3, 4, and 5 dimensions.

(b) Effects of size scaling on throughput using the synthetic 3DinCube dataset

Figure 4.5: Experiments measuring the effect of dimension and data structure size on throughput. Figures run with 144 threads.



(a) Small working set

(b) 3DinCube

Figure 4.6: Experiments measuring throughput in oversubscription on the 3DinCube dataset with a small working set, and the 3DinCube dataset without modification.

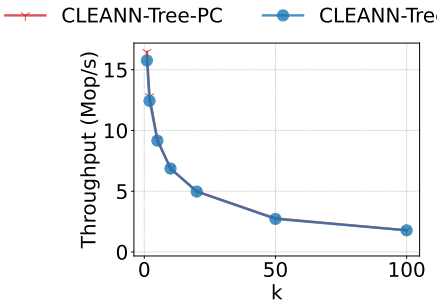


Figure 4.7: Experiments measuring the effect of k, the number of nearest neighbors, on the overall throughput. Experiment conducted on 144 threads on the 3DinCube dataset with 10 million points and a 50% mix of updates and queries.

Performance on Different Workloads. Next, we investigate the performance of the CLEANN-tree when varying the proportion of updates to queries. For these experiments, we build a tree of 10 million points on both the 3DinCube and 3DPlummer datasets and vary the update percentage from 0 to 100; the results are shown in Figures 4.4a and 4.4b, respectively. The CLEANN-tree maintains high throughput under all workloads. The path-copying version outperforms the hand-over-hand version as the update percentage increases; this is likely due to the efficiency of swapping in the modified path with a single pointer swing as opposed to updating nodes in a path one by one. Indeed, we find that on the skewed 3DPlummer dataset, which contains longer and more variable length paths, the path-copying version has a more pronounced advantage.

Complexity of queries and updates. The fact that updates are less expensive than queries CLEANN-tree may seem somewhat unintuitive. However, it follows because in the CLEANN-tree, each update traverses only a single root-to-leaf path and does not perform any distance comparisons. On the other hand, a k-nearest neighbor query must traverse a root-to-leaf path, compute the distance from the query to each point in the leaf, and potentially repeat this process with more leaves until all nearest neighbors have been found. This has the effect of making queries more expensive than updates.

Size scaling. To evaluate the CLEANN-tree on various dataset sizes, we used the synthetic 3DinCube dataset and generated datasets with sizes ranging from 10^4 to 10^9. We measure the throughput of the CLEANN-tree and the LFkD-tree under a workload of 50% queries and 50% updates on 144 threads as the steady-state size of the tree increases from 10^4 to 10^9. The results are found in Figure 4.5a. We found that the CLEANN-tree was able to support adequate throughput up to a tree size of 1 billion datapoints (the limit of our machine's memory). We notice a drop in performance at a tree with 10^4 points, likely due to increased contention, as the tree contains only a few thousand nodes, meaning that many updates might fail or stall behind a lock when all 144 threads are active. We were unable to run the LFkD-Tree at sizes greater than 10^7 due to memory constraints.

Dimensionality scaling. Next we investigate the performance of the CLEANN-tree under datasets of varying dimensionality using a synthetic dataset drawn from a hypercube of dimension 2, 3, 4, or 5, with results shown in Figure 4.5b. We find that the CLEANN-tree maintains good performance up to 5 dimensions, suffering the most on query-only workloads due to the "curse of dimensionality" where exponentially more bounding boxes lie adjacent to each bounding box. On update-only workloads, the performance of each dimension is quite similar.

Experiments under a small working set. To test the performance of the CLEANN-tree under a different type of workload, we take the 3DinCube dataset on 10 million points and generate an additional set of test points that are restricted to a very small geometric subset ("working set") of the original dataset's bounding box. We then perform an experiment where insertions, deletions, and queries are drawn only from that additional subset, thus mimicking the real-world situation where only a small area of a large data structure receives most of the traffic. In this scenario, we test only our data structure as the interface of the LFkD-tree did not support a

separate test and build dataset. In Figure 4.6a, we show the throughput of both variants of the CLEANN-tree. In this experiment, we found that the hand-over-hand version outperformed the path-copying version; this is likely because the effects of early unlocking were extremely beneficial in a scenario where all threads are trying to modify a very small subset of the tree nodes.

Effects of lock-free locks. We test the effects of using lock-free locks instead of traditional locking in two scenarios: the first is the "working set" example discussed above where the workload is isolated to a small subset of the tree, and the second is the 3DinCube dataset under a uniform workload. Since lock-free locks are primarily helpful in reducing contention under oversubscription, in this experiment we extend the number of hyperthreads up to 496 on our 72-core machine. In the working set scenario (Figure 4.6a), we find significant benefit from using lock-free locks after 72 threads, as well as a large slowdown using traditional locks past 144 threads. The slowdown is likely occurring because the workload is extremely contended, and thus the scenario where a thread holds a lock and then temporarily ceases operations becomes both increasingly likely to happen and increasingly likely to hold up the progress of other threads. Thus, the helping enabled by lock-free locks becomes crucial in obtaining a high throughput. On the other hand, in the uniform workload (Figure 4.6b), the operations are much less contended and the beneficial effects of lock-free locks do not appear until over 400 threads.

Effects of scaling k. We investigate the effects of scaling k in Figure 4.7, which tracks the throughput of operations on the same dataset as k varies from 1 to 100. Throughput declines rapidly as k decreases from 1 to 20, and then begins to flatten out past 20. Since each leaf holds up to 32 points, the average number of leaf accesses required to complete a search scales sub-linearly.

4.5 Applications of Shallow Augmentation

Here we describe some other applications that use augmented values, but for which the values rarely in practice need to propagate high up in the tree. We expect our techniques apply to these. Firstly we consider the classical interval stabbing problem. Here a data structure needs to maintain a set of intervals, each with a start point and an end point (in one dimension). The intervals could be, for examples, the time at which someone is logged into a computer. We assume intervals can be inserted and deleted. A query on a point returns whether any intervals cover that point, and if so returns one such interval. A standard solution is to use an augmented BST ordered by the left endpoints of the intervals. The right endpoints are stored as values, and nodes are augmented with the maximum of any right endpoint in the subtree. A query can then be answered by traversing the tree to the point and checking if any trees that are to the left of the path have a right endpoint larger than the point (i.e. cross the point). We note that since the augmentation is maximum, it most typical cases an update will not propagate very high up the tree.

Another application is for tournament trees. Here we simply want to keep a winner among a set of members. Assuming most updates do not propagate far up the tree, this again would only

need shallow updates in most cases. Another application is for approximate counters, and other approximate statistics. It is well known how to maintain approximate counts by only updating counts once in a while [73, 81, 113, 131, 133]. When applied on a tree, this means that counts will only once in a while need to be propagated to the next level. This idea has already been applied in the context of stand-alone concurrent counters [73], but can also be applied as an augmented value.

4.6 Conclusion

We present two general approaches for the design of lock-free and linearizable augmented trees: hand-over-hand locking and path copying. We apply these techniques to develop the CLEANN-tree, a lock-free and linearizable data structure supporting k-nearest neighbor queries and range queries that is built based on an augmented kd-tree. We perform experiments supporting the claim that the CLEANN-tree is scalable to high thread count, and which investigate scenarios in which path-copy and hand-over-hand locking differ in performance. Some avenues of future work include applying these ideas to other augmented kd-trees and more general concurrent augmented data structures.

Part II

High-dimensional Nearest Neighbor Search

Chapter 5

ParlayANN: Scalable and Deterministic Parallel Graph-Based ANNS Algorithms

5.1 Introduction

The adoption of deep learning methods over the past decade have led to high-dimensional vector representations of objects a.k.a. embeddings becoming widely used. These representations are typically obtained by training deep neural networks. As a result, machine learning datasets usually contain billions of vectors representing embeddings of users, documents, search queries, images, among many other kinds of objects. These embeddings can span hundreds to thousands of dimensions. The algorithms producing these embeddings are trained so that similar objects have "close" embeddings (e.g., in L_2 distance). As a result, an important problem is to find the nearest and thus most similar set of k objects for a query point in the embedding space $\mathbb{R}^d$.

This problem is known as *k-nearest neighbor search*, and is notoriously hard to solve exactly in high-dimensional spaces [44]. Since solutions for most real-world applications can tolerate small errors, most deployments focus on the *approximate nearest neighbor search (ANNS)* problem, which has been widely applied as a core subroutine for search recommendations, machine learning, and information retrieval [169], as well as large language models (LLMs) used in ChatGPT [10] and other applications combining LLMs and vector search [9, 57, 160, 165]. Considering that embeddings and similarity search are at the heart of these and many other modern AI applications, it is increasingly important to build scalable and efficient parallel ANNS solutions that can scale to massive modern datasets.

Some of the best-performing ANNS algorithms today are *graph-based* ANNS algorithms, which are able to achieve high recall (i.e., fraction of the true k-NNs returned by the query) while obtaining high throughput (queries per second, or QPS). Graph-based ANNS algorithms construct a *proximity graph* over the points that connects each point with closeby points. ANNS queries search for the k-nearest neighbors of a query point by traversing the proximity graph from a seed point, greedily exploring points that are closer to the query until the search converges. Among various types of ANNS algorithms, graph-based algorithms in general achieve superior

recall and QPS, as shown in many recent benchmarking papers [84, 126, 129, 135, 161].

Despite the focus on efficiency and benchmarking in the ANNS literature, *there is very little work (algorithmic ideas or benchmarking) that systematically studies how parallel graph-based ANNS algorithms perform as we scale the input size and the number of processors.* On the algorithmic side, some graph-based algorithms do have parallel implementations, but rely on per-vertex locks to enable parallelism which raises two major issues affecting both performance and "correctness". First, due to the use of locks, most existing implementations *tend to only scale well to tens of threads*. Fig. 5.1 demonstrates parallel scalability curves for four state-of-the-art (SOTA) implementations of graph-based algorithms (grey lines), on a well-known ANNS benchmark [35] with 1M points. None of them achieve significant speedup beyond 50 threads. Furthermore, using locks results in *non-deterministic* outputs, i.e., multiple runs of the algorithm may yield different proximity graphs due to lock acquisition order. Non-determinism can be a serious issue for applications that require persistence, crash recovery, or replication, e.g., for vector databases such as Pinecone, Weaviate, and Lucene [11, 13, 16].

On the benchmarking side, existing results [35, 169] focus on relatively small input sizes (usually million-scale), and evaluate algorithms based on their sequential performance. Therefore, techniques that perform well on existing ANNS benchmarks may not be suitable (or are unclear to be suitable) for a significantly larger dataset or more cores. Due to the lack of benchmarking studies focusing on parallelism, we also find that some of the scalability issues for existing parallel implementations are from some sequential bottlenecks that do not appear until a large number of cores or sockets are used, or until they are run on much larger datasets. Therefore, understanding how different ANNS algorithms scale from million to billion-scale as a function of the number of cores, and across a diverse set of datasets is an important open problem.

To address this problem, in this chapter we develop ParlayANN, a parallel ANNS library that scales to *billion-scale* datasets, scales to more than a hundred threads, and is deterministic. To achieve these goals, we exploit multi-threading (specifically, using fork-join parallelism) as much as possible to reduce the build time, which can be weeks on a single thread at such a scale. We provide new general techniques for building ANNS graphs in parallel, such as prefix doubling and batch updates. We then apply our general techniques to four SOTA graph-based algorithms: DiskANN [161], HNSW [126], HCNNG [135] and PyNNDescent [129]. In addition to new general techniques, we also developed several algorithmic optimizations to remove scalability bottlenecks for each specific algorithm, such as very large per-thread hash-tables (in HNSW, see Sec. 5.3.2), and certain data structures overflowing the L3 cache (in HCNNG, see Sec. 5.3.3). Our implementations, ParlayDiskANN, ParlayHNSW, ParlayHCNNG and ParlayPyNNDescent, are *deterministic, and achieve much better scalability than the best existing parallel implementations for each of them.*

Many of the tools in our library are of general use; to give an idea of the generality and practicality of ParlayANN, ParlayANN contains about 5000 lines of code, of which around 2000 are specific to one algorithm and the remaining 3000 are shared.

In Figure 5.1, we present the scalability of our implementations relative to existing implementations of graph-based ANNS algorithms on 1M points (all numbers are relative to the one-thread running time of the original implementation in each kind). Our implementations scale well up to all 96 cores on the machine we use, with further performance improvements from hyperthreading.

We carefully benchmarked our new implementations along with two existing SOTA non-graph algorithms (FAISS and FALCONN [29, 109]) on diverse real-world datasets with a billion points, including one dataset for out-of-distribution (OOD) queries (see more details in Sec. 5.4.1). Three of our implementations (ParlayDiskANN, ParlayHNSW and ParlayHCNNG) scale to billion-size datasets with reasonable preprocessing time for index building (around 10h) with high-quality query results (up to .99 recall with about 10^4 QPS). Our graph-based implementations achieve the best tradeoffs between recall and QPS across the recall spectrum, while the non-graph approaches failed to achieve a recall higher than 95% on billion-size datasets, even with very low QPS. **We believe this is the first work that scales deterministic parallel ANNS algorithms to billions of points with high recall.**

By supporting these algorithms in a unified framework (e.g., same parallel framework, primitives, and work-stealing scheduler) and applying similar optimization effort across all of them, our results also provide a *fair comparison of the algorithmic ideas* among the existing graph-based approaches, both for index quality and their potential for parallelism. Benchmarking these algorithms at a billion-scale required significant programmer and computational effort; for example, building all of the ANNS indexes shown in Sec. 5.4.1 (six algorithms each with three datasets) took more than 14 days of computation time on a machine with 192 threads. Our efforts led to a variety of interesting new findings about how ANNS algorithms perform as dataset sizes are scaled. **We believe this work is also the first to depict an accurate picture of performance comparison among ANNS algorithms on billion-scale datasets.**

In summary, our results include both algorithmic contributions and new experimental findings about the performance of ANNS algorithms at very large scales, listed as follows. Our code is available at `https://github.com/cmuparlay/ParlayANN/`.

1. A variety of general and specific techniques to parallelize existing graph-based ANNS algorithms to scale to billions of points (Sec. 5.2).
2. High-performance parallel implementation ParlayANN, which contains four graph-based ANNS algorithms.
3. In-depth experimental study of existing and our algorithms on a variety of billion-scale datasets, including a special dataset for out-of-distribution queries (Sec. 5.4).
4. A list of interesting findings about parallel ANNS algorithms on large scale datasets (Sec. 5.4).

Findings on modern ANNS applications. We test all algorithms on four billion-scale datasets. In particular, we include one dataset that is *out-of-distribution*, meaning that the query distribution is from an inherently different source from the base distribution (e.g., text queries on images-embedded datasets). It is often significantly harder to achieve good recall on such datasets, and most algorithms need to be adapted by training with the query data [94, 107]. We find that **graph-based methods significantly outperform IVF and LSH-based methods on billion-scale datasets**, which are especially to adapt to out-of-distribution data. Furthermore, graph-based algorithms were the **only** algorithms capable of achieving recall higher than 95% on billion-size datasets.

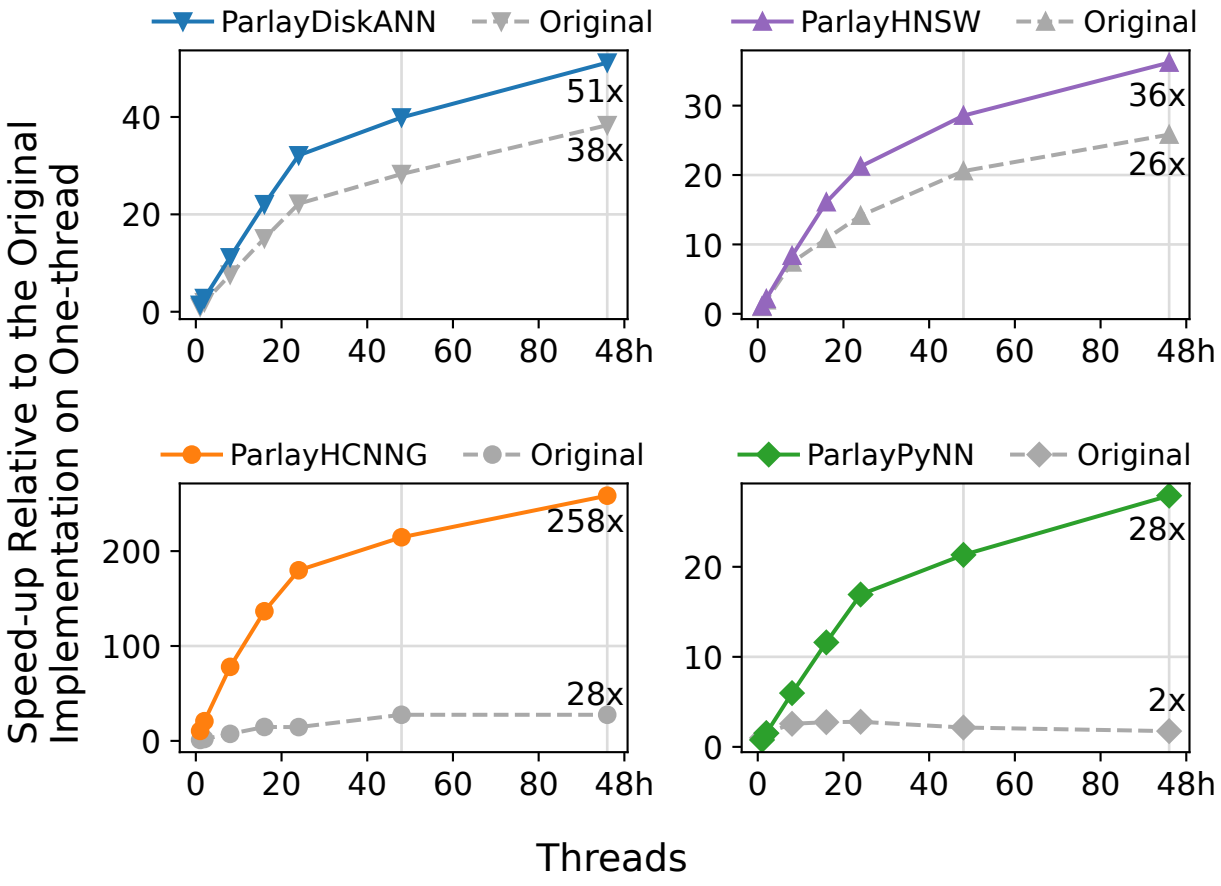


Figure 5.1: **Scalability of original and our new implementations of four ANNS algorithms on various number of threads. Within each subfigure, all numbers are *speedup numbers relative to the original implementation on one thread*. Higher is better.** Results were tested on a machine with 96 cores using dataset BIGANN-1M (10^6 points). "96h": 48 cores with hyperthreads. The two implementations in the same subfigure always use the same parameters and give similar query quality (recall-QPS curve).

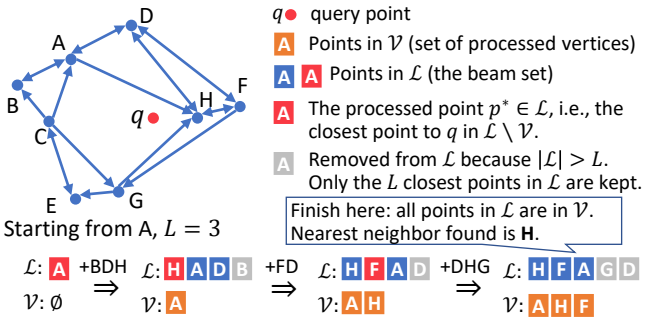


Figure 5.2: An example of ANNS graph and a greedy search. The blue arrows represent directed edges in the proximity graph, which is a mix of long and short edges. Below is an example of NNS query on point q (red point). The algorithm starts with adding the starting point A as the only point in the beam $\mathcal{L}$, and then in every step, finds the closest unprocessed point in $\mathcal{L}$ (to q) and adds its out-neighbors. Once $|\mathcal{L}|$ goes beyond L, it is refined to keep only the L nearest points. A set $\mathcal{V}$ is maintained for all processed vertices. When all vertices in $\mathcal{L}$ are also in $\mathcal{V}$, the algorithm finishes.

5.2 General Techniques for Graph-Based ANNS Algorithms

In this section, we describe our new techniques for parallel graph-based algorithms. We first present the high-level idea underpinning graph-based ANNS algorithms. We then introduce two major existing approaches: *incremental algorithms* and *clustering trees*, as well as our new general techniques to make them parallel and deterministic. In the next section, we show how these general techniques can be applied to four graph-based ANNS algorithms.

High-Level Approach Given point set $\mathcal{P}$, an ***ANNS graph*** $G_{\mathcal{P}}$ refers to a directed graph with vertices representing points in $\mathcal{P}$. For a point $p \in \mathcal{P}$, we define $N_{\text{out}}(p)$, or the out-neighbors of p. We illustrate an example of an ANNS graph on them in Fig. 5.2. The neighborhood of a point in the graph roughly corresponds to other nearby points, while some "long edges" are also needed (see details below).

Greedy (Beam) Search Almost all ANNS graph algorithms use a variant of ***greedy (beam) search*** to answer NNS queries (see Fig. 5.2 and Alg. 1). Such a search for a query q maintains a *beam* $\mathcal{L}$ with size at most L as a set of nearest neighbor candidates of q. We call L the *width* of the beam. The beam starts with a single starting point s. In each step, the algorithm pops the closest vertex to q from $\mathcal{L}$, and *processes* it by adding all its out-neighbors to the beam. We use a *visited set* $\mathcal{V}$ to maintain all points that have been processed (i.e., the neighborhood of the point has been traversed and added to the beam). If $|\mathcal{L}|$ exceeds L, the L closest points are kept.

Intuitively, for greedy search to converge quickly and produce accurate answers, the ANNS graph should contain a mix of long edges (connecting with neighbors that are far away) and short edges (connecting with neighbors that are close). Long edges enable fast navigation from the starting point towards the region close to a query point, and short edges enable the search to quickly converge once it reaches this region of the graph.

Algorithm 1: greedySearch(p, s, L, k).

Input: Point q, starting point s, beam width L, integer k.
Output: Set $\mathcal{V}$ of visited points and set $\mathcal{K}$ of k-nearest neighbors to point q.

$\mathcal{V} \leftarrow \emptyset$
$\mathcal{L} \leftarrow \{s\}$
while $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
  $p^* \leftarrow \arg\min_{(p \in \mathcal{L} \setminus \mathcal{V})} \|p, q\|$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
  $\mathcal{L} \leftarrow \mathcal{L} \cup N_{\text{out}}(p^*)$
  if $|\mathcal{L}| > L$ **then** retain only L closest points to q in $\mathcal{L}$
$\mathcal{K} \leftarrow k$ closest points to q in $\mathcal{V}$
return $\mathcal{V}, \mathcal{K}$

5.2.1 Incremental Algorithms

One class of graph-based ANNS algorithms is *incremental* algorithms, which work by inserting all points into the graph in some order; when inserting p, the algorithm adds new edges between p and the existing points in the graph so that p can be discovered by queries. Among the algorithms we study, DiskANN and HNSW are incremental algorithms.

Most incremental graph algorithms, such as DiskANN, HNSW, and NSG [84, 126, 161] use a greedy search procedure as a substep during insertion. Alg. 2 presents the high-level idea of this `insert` routine. Inserting a point p (Alg. 2) first does a greedy search on the existing graph, and then chooses the out-neighbors of p from the visited set $\mathcal{V}$ of the search by performing a `prune` routine. The `prune`$(p, \mathcal{V}, R)$ routine selects a subset from a candidate set $\mathcal{V}$ as the neighbors of p, which ideally should cover a diverse range of edge lengths and directions. Pruning also ensures that the size of $N_{\text{out}}(p)$ has at most a given ***degree bound*** R; smaller R typically results in fast but less accurate searches compared to a larger R. In addition to selecting out-neighbors of p, the `insert` algorithm must add p as the out-neighbors of other points so p is reachable during a search. This is done by adding p to each of p's out-neighbor q, and calling `prune` on q to ensure the degree bound R. The pruning strategies are specific to each graph-based algorithms, and we describe them in Sec. 5.3.

Challenges for Incremental Algorithms To parallelize incremental ANN algorithms, many existing implementations (e.g., DiskANN) insert all points in a single parallel loop over *all* the points, with per-point locks to ensure that the points are accessed safely. This can cause performance issues and cause non-determinism.

New Technique in ANNS: Prefix Doubling We now present our first technique to avoid using locks in incremental graph-based algorithms. Note that the main reason of using locks in the existing implementations is that the points being inserted in parallel all start from an *empty* index (graph), and therefore need a way to "see" each other and to "bootstrap". Using locks effectively sequentializes all conflicts and achieve a result close to the sequential algorithm, but introduces performance and non-determinism issues.

Algorithm 2: insert(p, s, R, L).

Input: Point p, starting point s, beam width L, degree bound R.

Output: Point p is inserted into the nearest neighbor graph.

1 $\mathcal{V}, \mathcal{K} \leftarrow$ greedySearch$(p, s, L, 1)$
2 $N_{\text{out}}(p) \leftarrow$ prune$(p, \mathcal{V}, R)$
3 **for** $q \in N_{out}(p)$ **do**
4 $\quad N_{\text{out}}(q) \leftarrow N_{\text{out}}(q) \cup \{p\}$
5 $\quad$ **if** $|N_{out}(q)| > R$ **then** $N_{\text{out}}(q) \leftarrow$ prune$(q, N_{\text{out}}(q), R)$

To address this, we use *prefix-doubling* [48, 72, 91, 92, 154]. The high-level idea is to insert points in batches of exponentially increasing size (but upper bounded by a parameter θ, see details below), as shown in the while-loop in Alg. 3. Each point will add itself based on the *snapshot at the end of the last batch*, and therefore points do not conflict with each other. Initially, the batches are relatively small, which more closely resembles the sequential version, allowing for a more accurate index initially. When the index becomes reasonably large, larger batches are allowed, which also enables high parallelism. Compared to the sequential version where point i is inserted based on the index of $i-1$ points, this approach allows point i to deterministically see an index with $O(i)$ points (roughly $i/2$), while extracting significant parallelism. For potential conflicts when adding multiple points to the neighborhood of an existing point, we carefully merge them together using a deterministic semisort. Prefix-doubling provides balance between *parallelism* (most of the batches are sufficiently large to utilize a large number of threads), *progress* (no contention or race within each batch), and *accuracy* (each point see a reasonably accurate snapshot of the index).

New Technique in ANNS: Batch Insertion and Pruning A basic building block in our incremental algorithms is ***batch insertion***, which adds a batch of points to the current index. In Alg. 3, inserting each batch involves two steps: (1) building the neighborhood for the newly-inserted points (Lines 7–9), and (2) adding the reversed edges to the existing points (Lines 11–14). Step 1 deals with each point in the batch in parallel, which uses a greedy search on the immutable snapshot index to find a candidate set, followed by pruning the candidates. In this step, all points in the batch construct their own neighborhood independently on an immutable snapshot, and thus does not affect each other. Therefore, this step is parallel and deterministic, and no locks are needed.

In the next step, the edges are reversed and any vertices whose neighborhood exceeds the degree threshold are pruned. To do this in deterministic manner without using locks, we collect all edges to be added in $\mathcal{B}$ in the format (u, v), where u is a newly-added point in this batch, and v is an existing point in the graph. We then run a ***parallel semisort*** [89]—that is, given a sequence A of entries, each associated with a *key*, reorder A such that all entries with the same key are consecutive—on $\mathcal{B}$ by the key of v, such that all edges incident the same existing point v are consecutive, and thus can be added together without locks.

Optimization: Batch Size Truncation While allowing each point to see an index that is roughly half the size it sees in the sequential setting, prefix-doubling may still lose significant

information in the last few rounds when the batches are very large. To avoid this, we upper bound the batch size by θ, which we empirically set to $0.02n$. This relaxation does not affect parallelism or scalability in practice; for large datasets, 2% of the input is more than enough to utilize all threads on modern multi-core computers. With this optimization, our prefix-doubling index achieved similar quality as the sequential version: ParlayDiskANN with $R = 64, L = 128$ on a benchmark dataset BIGANN-1M differs within 1% of the QPS from the sequentially-built index, at the same level of recall.

5.2.2 Clustering-Based Algorithms

Another approach for building an ANN graph is to use ***clustering trees***. At a high-level, the algorithm splits the input into two pieces, and keeps recursively splitting until the number of points drops below a given threshold, reaching a *leaf cluster*. The structure of splitting points form a tree-like structure, called a *cluster tree*. The splitting step usually involves randomization, e.g., we can generate a random hyperplane and split points based on which side of the plane they fall. Within each of the leaf clusters, a local ANN graph with stronger conditions (e.g., connecting each point with some exact nearest neighbors) is built.

Using different random seeds to generate different cluster trees, we can generate multiple (overlapping) local ANN graphs. The overall algorithm will obtain an ANN graph as the union of all local ANN graphs, and obtains the final ANN graph by performing some postprocessing. These algorithms differ in the methodology in generating the clustering tree, building the local ANN graphs, and/or postprocessing. Among the algorithms in this chapter, HCNNG and PyNNDescent use the clustering trees.

Challenges for Clustering-Based Algorithms There are several challenges to efficiently construct ANN graphs in parallel using this approach. Firstly, some existing systems achieve parallelism simply by parallelizing the construction of the T trees (each tree is constructed sequentially). Since empirically the best value of T is tens of trees (e.g., about 30 for HCNNG) [135], the algorithm naturally cannot scale to more than T threads in the tree construction step, which is also the main reason that the original HCNNG implementation in Fig. 5.1 does not improve beyond 30 threads. Secondly, existing parallel implementations also take per-point locks when merging the edges from all the local ANN graphs, which causes contention and non-determinism if pruning is used. Lastly, some subroutines, such as the local ANN graph construction, can generate costly (in terms of time or space) local structures, which can become a performance bottleneck when the data size or the number of threads is large.

Next, we present our general ideas to achieve better parallelism for clustering trees. In Sec. 5.3.3 and 5.3.4, we further discuss our new ideas to address the scalability issue in HCNNG and PyNNDescent.

Parallelizing Clustering-Based Algorithms To parallelize the clustering-based algorithms, we apply two general ideas. First, we parallelize the construction of *each clustering tree*. We then use parallel divide-and-conquer to always deal with both branches in parallel, and use a parallel partitioning primitive [49, 108] to assign points to different branches in parallel. This

Algorithm 3: batchBuild($\mathcal{P}, s, R, L$).

Input: Point set $\mathcal{P}$, starting point s, beam width L, degree bound R.
Output: An ANN graph consisting of all points in $\mathcal{P}$.

$start \leftarrow 1$
while $start \leq |\mathcal{P}|$ **do** `// Prefix-doubling`
  $end \leftarrow \min(start \times 2, start + \theta, |\mathcal{P}|)$ `// θ: batch size upper bound`
  BatchInsert($\mathcal{P}[start..end]$)
  $start \leftarrow end + 1$
Function BatchInsert($\mathcal{P}'$) `// Insert a batch P' to the current index`
  parallel for $p \in \mathcal{P}'$ **do**
    $\mathcal{V}, \mathcal{K} \leftarrow$ greedySearch$(p, s, L, 1)$
    $N_{\text{out}}(p) \leftarrow$ prune$(p, \mathcal{V}, R)$
  $\mathcal{B} \leftarrow \bigcup_{p \in \mathcal{P}'} N_{\text{out}}(p)$ `// All (existing) affected points`
  parallel for $b \in \mathcal{B}$ **do**
    `// N: all points in P' that added b as their neighbors`
    $\mathcal{N} \leftarrow \{p \mid p \in \mathcal{P}' \wedge b \in N_{\text{out}}(p)\}$
    $N_{\text{out}}(b) \leftarrow N_{\text{out}}(b) \cup \mathcal{N}$
    if $|N_{out}(b)| > R$ **then** $N_{\text{out}}(b) \leftarrow$ prune$(b, N_{\text{out}}(b), R)$

approach offers abundant parallelism across *all leaves*, instead of just over the trees. Although this is a natural idea, exposing more parallelism causes some challenges, e.g., for HCNNG, more threads running in parallel causes some space issues which we explain more in Sec. 5.3.3.

The second general technique is to avoid per-point lock when combining edges in all local ANN graphs. Instead of adding all edges concurrently, our idea is to collect all edges in an array and run a ***semisort*** on it, such that the edges incident the same point are consecutive. The graph can be built accordingly.

5.3 ParlayANN Algorithms

In this section, we further describe four graph-based ANNS algorithms that benefit from our techniques proposed in Sec. 5.2. In addition to the general techniques, we also employ specific optimizations for each individual algorithm to improve their scalability, which will be introduced below.

5.3.1 DiskANN

DiskANN [161] is a system consisting of an incremental in-memory ANNS graph algorithm as well as a system for storing the graph on an SSD. We focus on only the incremental ANNS graph algorithm as our work focus on the in-memory ANNS system. The in-memory DiskANN algorithm is almost completely described by Alg. 2, with the exception of the pruning step. In the paper on the navigating spreading-out graph (NSG) [84], Fu et al. proposed a pruning method on the visited list $\mathcal{V}$: roughly, they repeatedly select the point p^* closest to p in $\mathcal{V}$, then filter out

points p' that are *(α times) closer to p^* than to p* (i.e., remove all p' s.t. $\alpha\|p^*, p'\| \leq \|p, p'\|$). This can be thought of as streamlining navigation by pruning out long edges of triangles. As this technique is general, we also apply the α parameter to other algorithms in this paper to reduce their degrees (and thus make the ANN graph sparser) when possible, in order to make a more fair comparison.

To adapt DiskANN for machines to be scalable to hundreds of cores in the in-memory setting, we used the prefix-doubling approach as described in the previous section.

5.3.2 HNSW

The hierarchical navigable small world (HNSW) algorithm [126] is an incremental algorithm that constructs a hierarchical structure (intuitively the structure is similar to a skip list); each layer of the hierarchy is a navigable small world (NSW) graph [145]. In a NSW graph, nodes tend to be connected to their near neighbors, while ensuring that the overall graph is *navigable*, i.e., a search can reach any node in a small number of hops.

HNSW builds multiple layers of NSW graphs so that the lower layers are supersets of the upper layers. The number of vertices in each layer increases geometrically from top to bottom, and the bottom layer contains all the input points (conceptually this is similar to a skip list). Insertion in an NSW graph is also similar to Alg. 2. The `prune` scheme in HNSW is similar to DiskANN in that it prunes out long edges of triangles, but also includes some additional heuristics.

For search, HNSW traverses through the layers one at a time. It starts at the top layer, looks for the 1-nearest neighbor p of the query point using Alg. 1 with beam size 1, and shifts down to the next layer at p to repeat the procedure until reaches the last layer. Then, taking the current result as the entry point, it runs Alg. 1 to obtain the k-nearest neighbors at the bottom layer.

In our implementation (ParlayHNSW), we utilize parallel prefix-doubling. To adapt prefix-doubling to the multi-level hierarchical structure, we simply use batch insertion for each layer. We also carefully remove locks in all internal data structures in HNSW.

5.3.3 HCNNG

The hierarchical clustering-based nearest neighbor graph (HCNNG) [135] uses the clustering-based approach. The clustering works by randomly selecting two points p_1 and p_2, and partitioning the input by deciding whether a point is closer to p_1 or p_2. Leaf clusters are obtained when the number of points is below a given threshold. Within a leaf, the local ANN graph is a degree-bounded minimum spanning tree (MST), i.e., an MST where each point has degree at most K. Pruning is then applied to remove redundant edges.

Reducing Work and Space using Edge-Restricted MSTs We parallelized HCNNG without locks by constructing the clustering trees and merging edges in parallel as mentioned in Sec. 5.2.2. However, extra challenges emerge when a large number of threads can run in parallel. In particular, the MST is of the *complete graph* containing all pairwise distances of points in a leaf. When hundreds of threads perform this process on different leaves in parallel, the temporary memory usage can be very high. In our experience, storing all pairwise edges exceeds the

L3 cache on our machines, and severely limited speedup. To remedy this, instead of building the MST over all potential edges, we build an *edge-restricted MST*: instead of generating all pairwise edges, the MST is based on a graph where each point is connected with its l-nearest neighbors for some small l (we use 10). This optimization significantly saved space and in turn improved parallelism with no drop in QPS for a given recall. Our ParlayHCNNG is up to $12\times$ faster than the original HCNNG implementation (see Fig. 5.1), and achieves good self-relative speedup.

5.3.4 PyNNDescent

The PyNNDescent [129] algorithm uses a combination of a clustering-based approach to find an initial set of out edges along with iterative refinement to improve the set. The clustering initially used to construct the graph is based on choosing random hyperplanes. The local ANN graphs connects each point to the exact K nearest neighbors within each leaf. In addition to the clustering-based approach, PyNNDescent also includes a special postprocessing called *nearest neighbor descent*, which runs in an iterative way. Each round begins by undirecting the graph, i.e., adding the opposite edge of each directed edge. Then, each point p computes its two-hop neighborhood $\mathcal{Q}$ and retains the K closest candidates among the points $q \in \mathcal{Q}$. The algorithm terminates once only a small fraction of edges change on each round (i.e., converges). We then use a pruning algorithm to prune out the long edges of all triangles.

Optimizing Parallelism and Random Edge Sampling We had to significantly modify the PyNNDescent algorithm to scale to large datasets, and indeed as shown in Sec. 5.4, despite our optimization efforts we were not able to scale PyNNDescent to datasets with billions of points. However, our techniques still make it achieve reasonable QPS and recall on inputs with $\sim$100 million points.

The fundamental challenge is that calculating the neighbors of neighbors of a vertex requires work (and space) proportional to the square of the degree. We used two ideas to address this challenge. First, note that undirecting the graph edges can significantly increase the degree of a vertex. Thus, in edge undirecting, we limit each vertex's degree to be at most 2000 by randomly sampling edges, which makes the quadratic work more manageable. Also, we compute sets of two-hop neighborhoods in batches rather than all at once (i.e., we limit parallelism to limit the amount of intermediate memory used). With these optimizations, we were able to make our implementation, ParlayPyNNDescent, scale to 100M points, but the amount of temporary memory required to store two-hop graph made it infeasible to scale to a billion points.

5.3.5 Search and Layout Optimizations

In our experiments we use the *same* beam search algorithm across all of our implementations of ParlayDiskANN, ParlayHCNNG and ParlayPyNNDescent since they all generate a graph in the same format. The only difference is in how we select a start vertex. Our search algorithm for ParlayHNSW is also very similar, but slightly different since it needs to move between levels of the hierarchy. We have made a handful of modest optimizations to the search for all algorithms over the generic form given in Alg. 1, which we describe here.

Firstly we use an optimized approximate hash table with one-sided errors to quickly identify whether a point is in the visited set $\mathcal{V}$. Each point is inserted to the hash table by finding a random position. When two vertices map to the same position, only one will be stored, and the second will be revisited if encountered. The table size is set as the square of the beam size, which is large enough that revisiting is rare but is small enough to fit the table in the first-level cache. This is especially useful for improving the performance of the original HNSW, where a per-point flag array is used to check membership in $\mathcal{V}$, and in general improved the performance for all our algorithms by 28.6%–44.5%. We also avoid levels of indirection in the graph layout. In particular the edge-list for each vertex is kept at a fixed length so we can calculate its offset from the vertex id. We also use an $(1+\epsilon)$ pruning during the search as suggested by Iwasaki and Miyazaki [106]. In particular we only search vertices which have a distance to the search point that are within a factor of $(1+\epsilon)$ of the current k-th nearest neighbor. The ϵ is tuned based on the desired accuracy, but is never greater than .25. When sweeping the query parameters to obtain different points on the QPS/recall tradeoff curve, we therefore sweep two parameters: the beam size and ϵ.

5.4 Experimental Evaluations

In this section, we evaluate ParlayANN and present interesting findings from experiments at the end. We implement ParlayANN using C++ using ParlayLib [49] to support fork-join parallelism. We also use some standard building blocks (e.g., sorting, semisorting, partition) from ParlayLib.

5.4.1 Experimental Setup

Datasets We utilize three billion-size datasets for the majority of our experiments; we accessed these datasets through the BigANN Benchmarks competition framework, and some of these datasets were released for the competition [157]. The widely used *BIGANN dataset*[1] consists of SIFT image similarity descriptors applied to images [109, 110, 157]. It is encoded as 128-dimensional vectors using 1 byte per vector entry. The *Microsoft SPACEV dataset (MSSPACEV)* encodes web documents and web queries sourced from Bing using the Microsoft SpaceV Superior Model. The goal is to match web queries with appropriate web documents; the dataset consists of 1 byte signed integers in 100 dimensions [71]. The *Text2Image dataset (TEXT2IMAGE)*, released by Yandex Research, consists of a set of images embedded using the SeResNext-101 model, and a set of textual queries embedded using a DSSM model. Its vectors are represented using 4 byte floats in 200 dimensions [38].

Machines For most experiments, we used an AWS c6i-series virtual machine with two 3rd Generation Intel® Xeon® Gold 6314U Processors with 128 vCPUs available to the user, and 1 TB main memory.

[1]Note that throughout the chapter we use BigANN to refer to the benchmarking framework, and BIGANN to refer to the dataset.

	BIGANN	MSSPACEV	TEXT2IMAGE
DiskANN	.42	.35	.70
HNSW	.35	.37	.94
HCNNG	.45	.77	1.75
pyNNDescent	.42	.73	1.23
FAISS	.19	.13	.22

Table 5.2: Build times (hours) on hundred million scale datasets.

For the billion-scale results on TEXT2IMAGE, we used an AWS x2idn-series virtual machine with two 3rd Generation Intel® Xeon® Platinum 6314U Processors with 128 vCPUs available to the user, and 2 TB main memory.

For Figure 5.1 we used an AWS c7i-series virtual machine with two 4th Generation Intel® Xeon® Gold 6443N Processor with 128 vCPUs available to the user, and 1 TB main memory.

Measurement We report build times and QPS using all threads unless stated otherwise; throughout the experiments, we use QPS as opposed to latency, since QPS is more relevant to large multicore machines, and algorithms are typically always within an acceptable latency range. As discussed in Sec. 3.1, ANNS algorithms are primarily evaluated based on the *recall-QPS* curve, i.e., a curve where the y-axis is the QPS and the x-axis is the recall. To obtain points on this tradeoff curve, we perform a parameter sweep. Typically this is done by building a single (fixed) index, and then adjusting the parameters for a search, e.g., the beam-width, and ϵ value.

Baseline Algorithms We compare all our implementations with the original implementations of DiskANN [161], HNSW [126], HCNNG [135], and PyNNDescent [129], on the 1M-scale BIGANN dataset to demonstrate the improvement in scalability and parallelism over the existing implementations. The baseline implementations are carefully chosen from the BigANN benchmark to select the most competitive existing algorithms. The original HNSW implementation is safe for concurrent operations due to using locks, but does not exploit parallelism by default. We added a batch-parallel interface to the original HNSW using ParlayLib. For larger scale experiments, we compare ParlayANN to two non-graph algorithms based on inverted indexing (IVF): FAISS and FALCONN. For completeness, we describe these two algorithms in Section 5.4.2.

Algorithm Parameters Our interest is in optimizing for the high recall regime (from .9 to .999) at the highest QPS possible. For reproducibility, we provide our choices of parameters, which are chosen to give the best performance based on both our own experiments and the literature, in Figure 5.1.

Code Availability Our source code is available at `https://github.com/cmuparlay/ParlayANN`.

	BIGANN	MSSPACEV	TEXT2IMAGE
DiskANN	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = .98$
HNSW	$m = 32$, $efc = 128, \alpha = .82$	$m = 32$, $efc = 128, \alpha = .83$	$m = 32$, $efc = 128, \alpha = 1.1$
HCNNG	$T = 30, Ls = 1000, s = 3$	$T = 50, Ls = 1000, s = 3$	$T = 30, Ls = 1000, s = 3$
pyNNDescent	$K = 40, Ls = 100$, $T = 10, \alpha = 1.2$	$K = 60, Ls = 100$, $T = 10, \alpha = 1.2$	$K = 60, Ls = 100$, $T = 10, \alpha = .9$
FAISS	OPQ64_128, IVF1048576_HNSW32, PQ128x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ64x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ128x4fsr
FALCONN	$l = 30, rot = 1$	$l = 30, rot = 1$	$l = 30, rot = 1$

Table 5.1: Parameters chosen for each dataset. For DiskANN, HCNNG, and pyNNDescent, α denotes the pruning parameter. For DiskANN, R denotes degree bound and L is the beam size. For HNSW, m denotes the degree bound and efc is the efConstruction. For HCNNG and pyNNDescent, Ls denotes leaf size and T denotes number of cluster trees. For HCNNG, s denotes MST degree. For pyNNDescent, K is the degree bound. For FAISS, the first string is the type of vector transform used for building the index, the second denotes the IVF index type, and the third indicates the PQ compression for the queries. For FALCONN, l is the number of hash tables, and rot is the number of rotations.

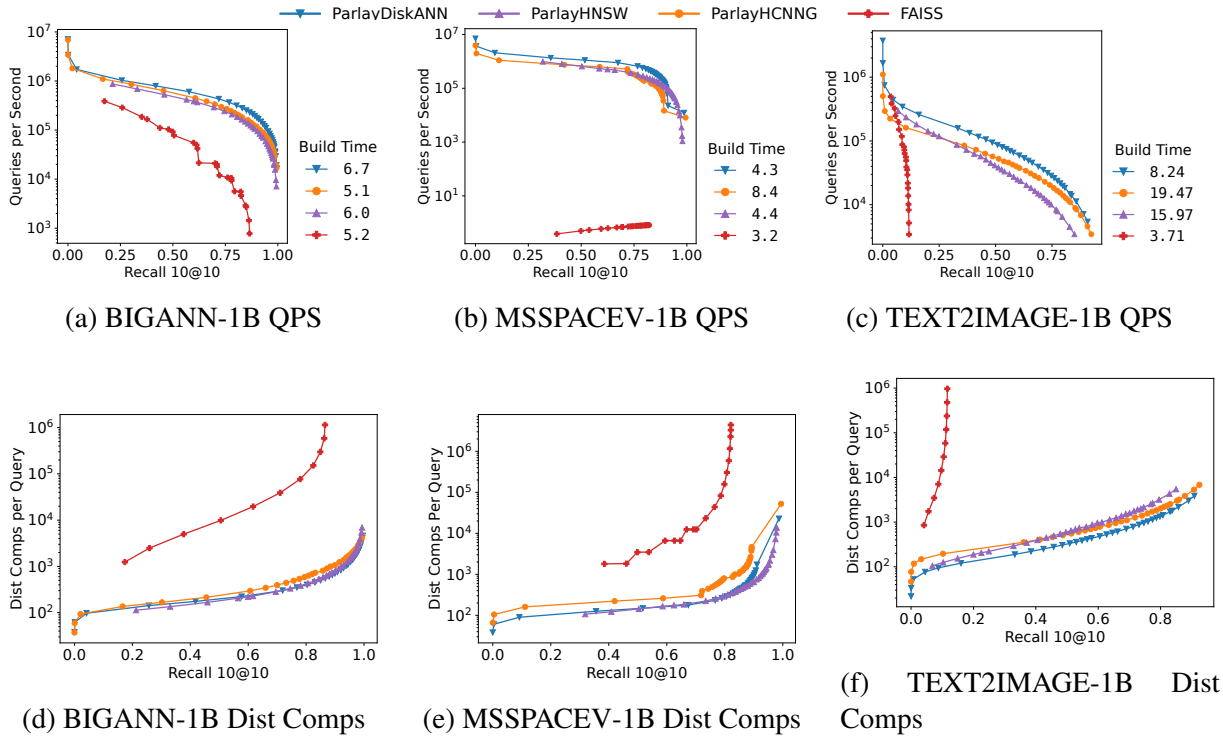
5.4.2 Partition-Based Competitors

Next, we introduce inverted-indexing algorithms (IVF), which constitute a large class of popular algorithms based on locality-sensitive hashing (LSH) and clustering. At a high level, IVF algorithms partition the vectors into buckets called *posting lists*; at query time the query only exhaustively searches elements in only a small number of lists instead of the entire space. We study two IVF-based algorithms in-depth, which we describe next.

FAISS-IVF FAISS is a widely-used and highly optimized library for efficient similarity searching on CPUs and GPUs [75, 109, 110, 112]. It provides a variety of options for *product quantization (PQ)*—that is, compressing vectors to a lower dimensionality to make distance computations faster. The primary search structure of FAISS is the inverted index; points are typically assigned to buckets via single-level clustering or two-level clustering, but can add other techniques such as using HNSW or NSG to help select buckets during search. At the billion scale, a typical FAISS index follows three steps: (1) compute a PQ transform on the vectors for the purpose of building the index, (2) build an inverted file index, possibly with an HNSW structure over the buckets, and (3) compute a PQ compression of the vectors for the purposes of querying the index.

We use the original implementation of FAISS, through the BigANN Benchmarks testing framework [157], in our experiments.

FALCONN FALCONN is a C++ library for LSH-based ANNS algorithms [29]. Locality-sensitive hash (LSH) functions are class of hash functions that can easily cause collisions for close vectors in a space. More specifically, such hash functions perform randomized space partitions of a unit sphere with the same dimension as the input, and mark the vectors in the same partition (as a bucket) with the identical hash value. FALCONN uses multiple hash functions to



(a) BIGANN-1B QPS (b) MSSPACEV-1B QPS (c) TEXT2IMAGE-1B QPS

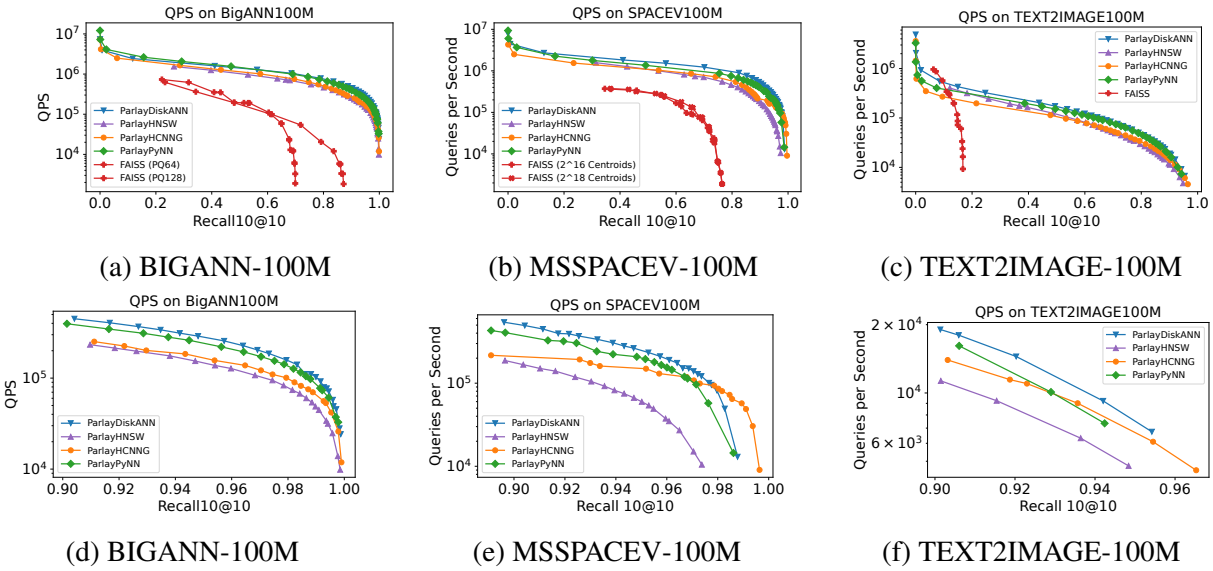
(d) BIGANN-1B Dist Comps (e) MSSPACEV-1B Dist Comps (f) TEXT2IMAGE-1B Dist Comps

Figure 5.3: Build time (hours), QPS, recall, and distance comparisons for all algorithms on billion-size datasets.

create each hash table, thus increasing the number of buckets and decreasing bucket size. This prevents buckets from having too many elements and thus prevents the search from having to access too many candidates. To improve accuracy, it builds multiple (replicated) hash tables for higher probability of success, trading off accuracy for higher memory consumption. A search hashes the queried point using each hash function, then collects all the elements from corresponding buckets among the tables, and retains the nearest neighbors among these candidates. Furthermore, by enabling multi-probe LSH, which assigns a vector to more than one bucket in each hash table, FALCONN is able to go consider more candidates from the additional buckets without needing to create more hash tables, thus both improving accuracy and saving memory.

5.4.3 Comparison with ANN Benchmarks

First of all, we demonstrate the single thread performance of ParlayANN on BIGANN-1M in Fig. 5.5. We refer to the parameter settings in the ANN Benchmarks framework [35], and compare to the publicly-available numbers on the website. The single-thread performance of ParlayANN roughly match the results on ANN Benchmarks website [36]. Due to the poor performance of on BIGANN-1M, and its correspondingly low performance on the hundred million and billion size datasets, we do not include FALCONN in further figures.



(a) BIGANN-100M (b) MSSPACEV-100M (c) TEXT2IMAGE-100M

(d) BIGANN-100M (e) MSSPACEV-100M (f) TEXT2IMAGE-100M

Figure 5.4: QPS-recall curves on all 100-million size datasets. The first row shows the overall QPS/recall curve, while the second row zooms into a higher-recall regime. The build times are given in Tab. 5.2

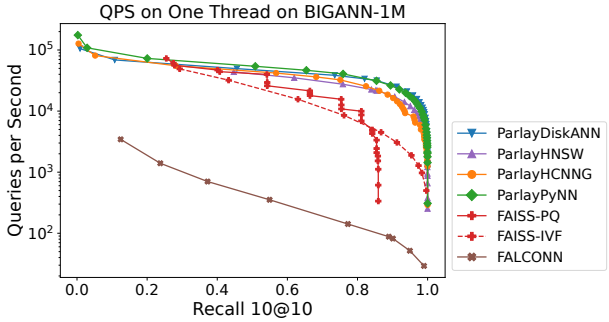


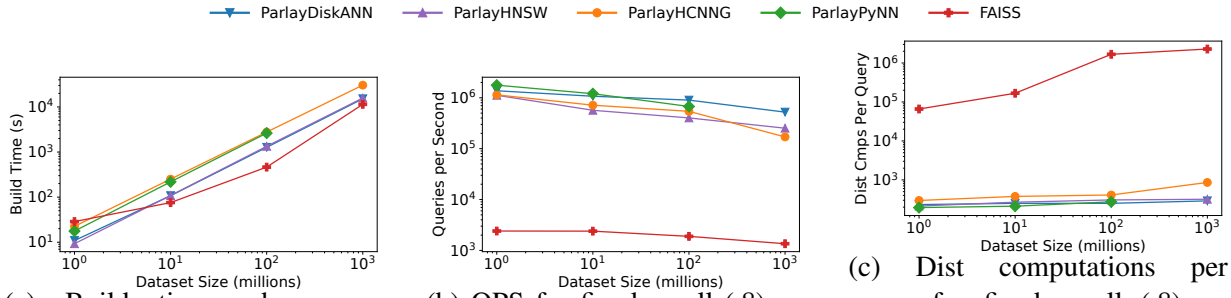
Figure 5.5: QPS on a single thread on BIGANN-1M. Shown to compare with ANN-benchmarks.

5.4.4 Parallelism and Scalability

To substantiate our claims of improving the parallelism of each graph-based algorithm as well as illustrate issues with the parallelism of the original implementations, we compare ParlayANN with the original implementations of each algorithm. We present the performance of building the index (graph) as the number of threads increases in Fig. 5.1. For the same algorithm, all numbers (both original and ours) presented are ***running time speedup relative to the original implementation on one core***. Therefore, the curve provides a direct running time comparison between the original implementation and our implementation (higher is better). For each algorithm, the two implementations always use the same parameters, and achieve similar query quality (except for some where ParlayANN also improved queries and achieved *better* query quality).

For DiskANN, we find about 1.2× improvement in performance by ParlayANN. The original DiskANN scales well to 30 to 60 threads but eventually the use of locks leads to perfor-

mance degradation on more threads. HNSW suffers from similar locking-related issues, and ParlayHNSW performs much better with more than 50 threads, and eventually achieves 1.4× better performance. As mentioned in Sec. 5.2, the original HCNNG only exploits parallelism by building all clustering trees in parallel, and fails to scale beyond T threads as a result. Our ParlayHCNNG was both faster on a single thread and even better when the number of threads increases, and eventually becomes 12× faster than their implementation when using all threads. PyNNDescent's original implementation used Numba [120] for parallelism and did not scale beyond 16 threads on our machine. Our implementation eventually becomes 33× faster than their parallel implementation.



(a) Build times shown on MSSPACEV as dataset size increases.

(b) QPS for fixed recall (.8) on MSSPACEV as dataset size increases.

(c) Dist computations per query for fixed recall (.8) on MSSPACEV as dataset size increases.

Figure 5.6: Figures showing the effect of dataset size on different metrics using the MSSPACEVdataset.

5.4.5 Full Billion-Scale and Hundred-Million Results

In this section we present our results for all algorithms and for three billion-scale datasets as well as their hundred-million scale versions.

Fig. 5.3 shows the QPS-recall and distance-comparison-recall curves for all tested algorithms on the three billion-scale dataset, along with the corresponding time to build their indexes presented on the side. As mentioned in Sec. 5.2, ParlayPyNNDescent is not present in the billion-scale figures since its memory requirements were infeasible for billion-scale datasets; it can be found in the hundred million-scale experiments. It is competitive with the other algorithms at the hundred-million scale.

In general, all our graph-based implementations achieve similar performance in both build and query. All of them can build the billion-scale indexes in around 10 hours. Among them, ParlayHNSW has slightly shorter build time (up to 2.3× faster than the other two), and ParlayDiskANN is slightly better in query (the recall-QPS curve is almost always at the top).

The non-graph algorithms we compared to achieved faster index building time, where FAISS is usually 1.5–3× faster than the graph-based algorithms. However, both of them (especially FALCONN) *struggled to get high recall on all datasets* We made many attempts to achieve the best query quality for FAISS and FALCONN, including increasing the building time and

using the suggested parameters from existing resources (e.g., FAISS Wiki [111]). The results we present are the best we achieved after extensive experiments..

For BIGANN and MSSPACEV, FAISS did not achieve a recall higher than 0.8 even with very low QPS. At a 0.8 recall, FAISS has about 8.51–18.7× lower QPS than the graph-based algorithms.

FAISS achieves QPS close to (but still lower) the graph algorithms at low recall values, but the QPS drops dramatically when a recall higher than 0.6 is desired.

FAISS also performs especially poorly on the out-of-distribution (OOD) dataset TEXT2IMAGE, where both of them only achieved a 0.2 recall at most.

Ultimately, higher build times may be acceptable if the resulting index can achieve high recall and QPS. From this perspective, we find that the graph algorithms adapt better to achieve high-recall and QPS on billion-scale datasets compared with non-graph ones. For BIGANN, all of the three graph-based algorithms eventually can achieve close to 100% recall at about 10^4 QPS. For MSSPACEV, ParlayHCNNG achieves close to 100% recall at 10^4 QPS, while the other two can also achieve a recall above 0.9.

This advantage (high recall) of the graph-based algorithms is especially true for queries that are out-of-distribution (OOD). While the query quality of the non-graph algorithms seemed to be severely affected by the OOD queries, all the three graph-based algorithms were still capable of achieving a recall of 0.8 or more on this challenging OOD dataset (ParlayDiskANN can even achieve a recall at 0.9). At the same recall, the QPS of the graph-based algorithms is 12.2–19.6× slower compared to the other non-OOD datasets.

5.4.6 Dataset Size Scaling

How do ANNS algorithms scale as we increase the size of the dataset? We start with the MSSPACEV dataset as an example to explore this question and present the result in Fig. 5.6 at a fixed recall of 0.8. In addition to build times and QPS, we also measure the average distance computations per query for each algorithm. We study this metric because for most ANNS algorithms on high-dimensional points, the distance comparison are the most expensive part.

For our graph-based algorithms, we found the build times incurred slightly superlinear increases as the dataset size increased (Fig. 5.6a); build times increased by a multiplicative factor of 11–12× when the size of the dataset increased by 10×. For ParlayHNSW and ParlayDiskANN, this superlinear increase can be attributed to the mechanics of the beam search: on a larger graph, beam search takes longer to terminate as there are more suitable candidates in its frontier. For ParlayPyNNDescent, we found that the nearest neighbor descent process consistently took more rounds to terminate for larger dataset sizes. Since the nearest neighbor graph for a larger dataset will likely have a larger diameter, two-hop exploration takes longer to "propagate" through the entire graph. For FAISS, we found an unusually small increase in build time between the 10M and 100M datasets. We attribute this to issues with parallelism that become less of a bottleneck at higher numbers of data points.

For QPS (see Fig. 5.6b), ParlayDiskANN and ParlayHNSW show a steady decrease in QPS as the dataset size increases. Part of the reason for this decrease is that a beam search with the same parameters on a larger graph will not only be *slower* than the same search on a smaller graph, it will also be *less accurate* since it visits a much smaller fraction of all the vertices. Since

Fig. 5.6b and 5.6c keep the recall fixed at 0.8, they must use an increased beam width at larger dataset sizes, thus contributing to lower QPS.

ParlayHCNNG and ParlayPyNNDescent both show steeper drops in QPS at fixed recall than ParlayDiskANN and ParlayHNSW. This may be because they only express close neighbor relationships with their edges. As the data size grows, the relationships they express cover smaller and smaller proportions of the whole dataset. Thus, they require larger (more costly) parameters to obtain the same level of recall as the data size increases.

Somewhat surprisingly, QPS and distance computations for FAISS remained almost the same for the 100M and 1B datasets. We confirmed that this phenomenon persisted through a wide range of parameter choices.

In general, the non-graph based algorithms perform more distance computations but achieve lower recall (and QPS). This indicates that most of their distance computations are less effective than those in graph-based algorithms (i.e., were not contributing to finding closer neighbors). This is possibly an important reason that they achieve much lower QPS than graph-based algorithms on a fixed recall, and indicates the effectiveness of graph-based algorithms for ANNS.

5.4.7 Conclusions from Experiments

We summarize our findings about ANNS algorithms on billion scale pointset below.

1. Graph-based algorithms are especially capable at achieving high recall (greater than .9) at the scale of billions of points for QPS in the 10k–200k range.
2. FAISS can achieve QPS close to the graph-based algorithms at a low recall, but QPS may significantly drop when a recall higher than 0.6 is required.
3. The IVF algorithm FAISS struggled to achieve high recall at a billion scale, while FALCONN achieved such low QPS that we did not include it in our experiments.
4. All algorithms struggle to achieve high QPS on OOD data, but graph-based algorithms adapt much better: they can achieve 0.8 or higher recall with slightly lower QPS, while it is hard to achieve even 0.2 recall for IVF algorithms.

5.5 Related Work

Approximate Nearest Neighbor Search Algorithms Data structures for ANNS fall roughly into four categories: graphs, inverted indices, locality-sensitive hash tables, and trees. A graph-based algorithm constructs a graph where the nodes represent points in the index and the edges represent proximity relationships, and where nearest neighbor queries are answered by applying a heuristic search on the graph. Prominent examples of graph-based algorithms include NSG [84], HNSW [126], DiskANN [161], but the academic literature includes many other graph-based approaches [1, 2, 4, 5, 7, 53, 62, 74, 85, 96, 105, 106, 124, 129, 135, 150, 182].

A commonly-used type of bucketing-based algorithms is the Inverted File Indexing (IVF) algorithms. IVF algorithms truncate the search space of a nearest neighbor algorithm by partitioning vectors into buckets called *posting lists*; queries exhaustively search elements in only a small number of lists instead of the entire space. One assignment method is to use a locality-sensitive

hash (LSH) function. Inverted file structures typically use a clustering algorithm to assign vectors to posting lists, with distance to a representative element used to determine which lists a query is mapped to. Some notable IVF-based algorithms include PLSH [162], FAISS-IVF [75, 109, 112], and FALCONN [29], along with many others [2, 5, 34, 63, 94, 117, 144, 178].

Trees such as kd-trees or cover trees are well-known data structures for computing nearest neighbors in metric space with low dimensionality (either actual or intrinsic) [32, 44, 91, 117], useful for many such applications [46, 69, 180]. Their search methods are subject to the curse of dimensionality, but there are some modified tree-based approaches for high dimensional search [3, 6, 125, 134].

In this work, we focus on improving the scalability of building ANNS indexes based on graphs. There also exists work focusing on improving parallelism and scalability for other ANNS-related topics, such as intra-query parallelism [142, 143] for graph-based algorithms, and improving scalability for tree-based algorithms on time series data [147].

ANNS at a Billion Scale Next, we review what is currently known about scaling ANN algorithms to billion-scale datasets. Early work on ANN measured performance on datasets with up to a billion points using various forms of IVF [39, 110, 162, 178]. The results for FAISS [75], the best known of the algorithms in this class, have been reported for the BIGANN and DEEP billion scale datasets [111]. These works do not include comparisons to graph-based algorithms, and focus on recall for the single nearest neighbor instead of the k nearest neighbors (i.e., $1@n$ instead of $k@k$).

Other works use secondary storage-based algorithms to scale to billion-scale datasets. DiskANN [161], a graph-based algorithm, gives numbers for BIGANN and DEEP for a billion points. They present limited comparisons to the FAISS [75] and IVFOADC+G+P algorithms [39]. The SPANN system [63] uses an inverted index where the posting lists are stored in secondary memory. On billion scale data (BIGANN, DEEP and MSSPACEV) it only compares to DiskANN. These existing works report the latency for one query at a time, presumably because running multiple queries across cores does not scale well due to limited secondary memory bandwidth and/or internal parallelism within the query [63]. The query throughput is therefore much lower than in-memory-based systems we report on in this work, even accounting for machine size (i.e., number of cores), although they have the advantage of needing less primary memory.

Johnson, Douze, and Jégou [112] report billion scale numbers on a GPU-based implementation using an inverted-index-based approach. Here again, the recall rates are low and the implementation is only compared to another GPU-based system [177]. Recent work on BLISS [94] uses the same datasets as we do at a billion scale. They compare their approach to HNSW, but the numbers they report for HNSW are much worse than those we have found and that are reported here (over an order of magnitude). Several systems work on a billion or more points, but do not report numbers or comparisons to other systems [5, 84, 117, 126].

Benchmarking ANNS There are two main works that benchmark ANNS algorithms, one at the scale of millions of points and one at the scale of billions. The first is the ANN Benchmarks repository focusing on million-scale datasets [35]. This is a benchmark suite of ANNS algorithms where any contributor may submit an ANNS algorithm to be included in their public

evaluations. Each algorithm is run by the authors on up to nine million-scale datasets. Lastly, the Billion Scale ANNS Challenge, a competition hosted at NeurIPS 2021 [157], focused on billion-scale ANNS algorithms on three different hardware tracks and six different billion-size datasets, including one range query dataset and two datasets that exhibit OOD characteristics. These existing benchmarks are a valuable resource, but their user-sourced code for each algorithm is subject to implementation differences and is not necessarily a comparison of the algorithmic ideas.

Chapter 6

Range Retrieval with Graph-Based Indices

6.1 Introduction

The object of approximate nearest neighbor search is finding the top-k most similar embeddings. These top-k results can then be post-processed for the desired application. However, there are some applications for which retrieving top-k embeddings for a fixed k is a poor fit. Applications such as duplicate detection, plagiarism checking, and facial recognition require instead to retrieve all results within a certain *radius* of a query rather than top-k, with some post-processing after to verify whether a match exists out of the retrieved items [76, 130, 152]. Furthermore, in real-world search applications some queries may have tens of thousands of matches while others have none or very few. In these cases, search with a small radius, or range retrieval, may be used to both differentiate these query types [163]. Range search can also be a useful subroutine in applications using nearest neighbor graphs, such as clustering or graph learning [95, 122].

Range search differs from top-k search by the diversity in size of the ground truth solutions for a set of queries. In practice, solutions for a set of queries follow a skewed, Pareto-like distribution where the majority of queries tend to have no results within the chosen radius, while a smaller fraction have a small number of results, and a few outliers have thousands to tens of thousands of results. Queries in the middle category are well served by existing similarity search algorithms, which are already highly optimized and efficient. Thus, a good range search algorithm will match the efficiency of top-k search on the middle category of queries, while quickly terminating on queries with no results, and efficiently finding all results for those few queries with thousands of results.

Despite the many applications of range search, there has been very little work on designing algorithms specifically for the task of high-dimensional range search. From a practical perspective, it would be ideal of data structures for top-k search could be reused or lightly adapted for range search, enabling an existing data structure to serve both types of queries. Data structures for top-k search typically fall into one of two categories—partition-based indices, which partition the dataset into cells and exhaustively search a small number of cells at query time, and graph-based indices, which construct a proximity graph over the data points and use a variant

of greedy search to answer queries. Graph-based indices are widely acknowledged as achieving equal or better similarity search performance than partition-based indices [169], but almost no work has studied the question of adapting graph-based indices for range retrieval. Furthermore, top-k search with IVF indices naturally explores thousands of candidates per query by exhaustively checking the distances between the query point and all the points in each cell, while the number of nodes a graph-based search typically explores is only a small multiple of k. It is thus a much less trivial algorithmic task to adapt graph-based search to serve such queries (as we will show in Section 6.3, naive adaptations of existing search methods are not adequate for the task). The graph structure also suggests potential to terminate queries with no results after examining just a handful of vertices. It is natural then to ask whether graph-based indices can be adapted to efficiently serve range queries.

Our Contributions In this chapter, we present a set of algorithms designed for approximate high-dimensional range search on graph-based ANNS indices. Our techniques are modifications of the standard graph search algorithm, meaning that we enable one data structure to efficiently serve both top-k queries and range queries. We present techniques that allow a range query to quickly terminate when the query has no results, and to efficiently answer queries with thousands of results within their radius. We devise an early-stopping heuristic that is capable of predicting whether a query has no results after only a few hops in the graph search. For queries with many results, we design two algorithms, *doubling search* and *greedy search*, that extend the search path to return more results while minimizing wasted work, and compare and contrast situations where each algorithm dominates the other.

For our experimental evaluation, since there is only one publicly released dataset specifically for range search, we evaluate eight state-of-the-art publicly available metric embedding datasets with up to 100 million embeddings and select a suitable radius for each one. We also perform additional analysis of the characteristics of different range search datasets, and how they connect to the magnitude of improvement our algorithms achieve.

We evaluate our algorithms on the set of nine datasets (eight contributed by us, and one public benchmark), and find that our range retrieval algorithms are capable of up to 100x speedup over naive adaptations of top-k search, and in most cases 5-10x speedup. We additionally find that the speedup and scalability of our algorithms extends up to datasets with 100 million points. Figure 6.1 shows a preview of our experimental results on three embedding datasets of sizes one million to 100 million. In each case, for a fixed average precision, we find at least a 10x speedup in throughput; additionally, our algorithms extend to significantly higher accuracy than the graph search baseline.

Outline In Section 6.1.1 we cover related work. In Section 4.2 we cover some needed preliminaries. In Section 6.2 we explore the characteristics of range search datasets and provide heuristics for choosing an acceptable radius, which we use to adapt eight public embedding datasets for range searching. In Section 6.3 we present our algorithms for range search on graph-based indices, and in Section 6.4 we present experimental results.

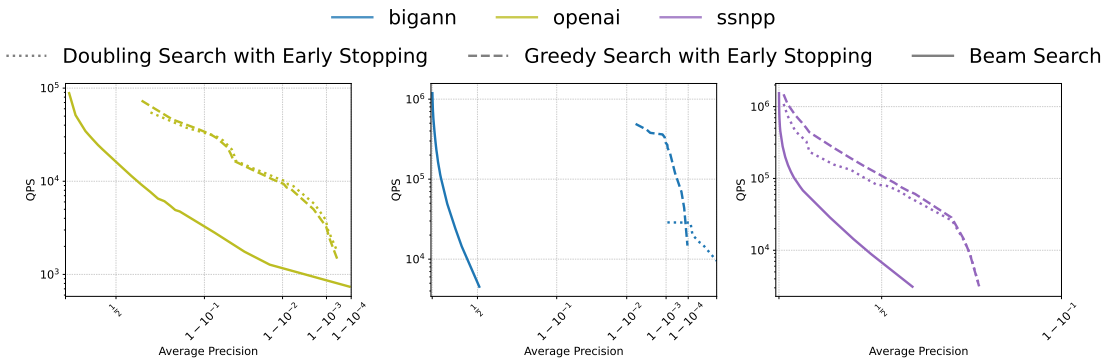


Figure 6.1: A preview of our experimental results on three datasets: from left to right, OpenAI-1M, BIGANN-10M, and SSNPP-100M. The solid line shows the beam search baseline, while the dotted and dashed lines show our two new algorithms. Datasets and algorithms are described in detail in Sections 6.2 and 6.3, respectively.

6.1.1 Related Work

Work on Range Searching Some earlier work addresses range retrieval in high dimensions from a theoretical standpoint [61], or with only minimal experiments that do not extend to the large embeddings used today [168]. Theory results on nearest neighbor search using locality sensitive hashing (LSH) also implicitly apply to range search, since their approximation guarantees are usually in the form of finding neighbors within a $(1 + \epsilon)$ radius of the distance to the true top-k result [103].

On the practical side, Meta AI released a range retrieval dataset aimed towards detecting misinformation [130], which was used as one of the competition datasets in the NeurIPS 2021 Big ANN Benchmarks Challenge [157], but competitors solved this task using naive adaptations of top-k search rather than novel range searching algorithms. A recent work by Szilvasy, Mazare, and Douze [163] addresses the combined task of retrieving range search results with one metric and then re-ranking them using a more sophisticated model. They address the question of designing a range search metric (RSM) specifically for *bulk* search—that is, range search over a group of queries with a fixed computational budget—and benchmark its results on an image search application. They use an inverted file (IVF) index, a common data structure for similarity search [77], to serve their database queries.

The Starling system [170] is an SSD-resident graph-based ANNS index optimized for I/O efficiency. Their work includes benchmarks on range search metrics, and they use an algorithm similar to the doubling beam search algorithm we present in Section 6.3. However, the focus of their paper is not on comprehensive benchmarking of range search, and since their implementation is a disk-resident index and ours is fully in-memory, our experiments are not directly comparable to theirs.

Other Related Work Some variants on the high-level ideas behind our range search algorithms—that is, early termination of queries with few results, and extensions of beam search for queries with many results—have been used before in the context of nearest neighbor search. A work by Li et al. [121] uses a machine learning model to predict when a query can terminate early and still maintain high accuracy. Another work [179] uses graph search with two phases (essentially, an initial cheaper phase and a later, more compute intensive phase for selected queries to gather additional candidates).

6.2 Characteristics of Range Search Datasets

In this section, we take eight existing embedding datasets from top-k retrieval benchmarks and show how to use them for range search—namely, by finding a radius which yields a suitable distribution. We evaluate the characteristics of each dataset; alongside, we also evaluate the one existing range search benchmark.

For a dataset to be meaningful as a range search benchmark, the same distance ϵ must be indicative of a match for each query point. While this is not necessarily the case for all embedding datasets, it is not directly observable without full access to the source material, and even with that access would be incredibly costly to determine. Thus, we use the characteristics of existing benchmarks to determine whether a dataset is suitable, and select a radius if so. Real-world query sets tend to follow a power-law distribution, where most points have no matches, while a handful of points have thousands of matches [130, 163]. Thus, we aim to choose a radius that produces such a power-law distribution for each dataset.

Another important aspect of suitability for range search is whether there exists a choice of radius that is "robust" in the sense that small perturbations of the radius do not result in a wildly different match distribution. Our later experiments show that datasets that are more robust by this measure tend to perform better over the baseline measures than ones which are more easily perturbed.

See Figure 6.1 for a description of each dataset used in our benchmarks.

6.2.1 Density of Matches by Dataset

We begin our investigation by computing a measure of density of each dataset, in order to both find a suitable radius and determine whether the choice of radius is "robust" in the sense that small changes to the radius do not result in drastically more points in each ball around a query point. To do this, we take each dataset and vary the radius, recording what we refer to as the "percent captured" for each radius—that is, for each query point q and radius r, what fraction of the dataset is captured in the ball of radius r around q? Formally, for each query set with size m, the percent captured by radius r with respect to a point set P of size n is

$$\frac{1}{m}\sum_{i=0}^{m}\frac{|\{p \in P \text{ s.t. } d(p, q_i) \leq r\}|}{n}$$

This information is presented for each dataset in Figure 6.2, which shows Euclidean and inner product datasets separately and plots the percent captured from the largest radius achieving 0%

Dataset	Metric	Dimension	Query Size	Selected Radius	Source	Embedding Model
BIGANN	L2	128 (uint8)	10000	10000	Images [109]	SIFT descriptors
DEEP	L2	96 (float)	10000	.02	Images [38]	The last fully-connected layer of the GoogLeNet model
MSTuring	L2	100 (float)	100000	.3	Bing queries [181]	Turing AGI v5
GIST	L2	968 (float)	10000	.5	Images [109]	GIST descriptors
SSNPP	L2	200 (uint8)	100000	96237	Images [130]	SimSearchNet++
OpenAI	L2	1536 (float)	10000	.2	ArXiv articles [88]	OpenAI text-embedding-ada-002 model
Text2Image	IP	200 (float)	100000	-.6	Images [38]	Se-ResNext-101
Wikipedia	IP	768 (float)	5000	-10.5	Wikipedia articles [149]	cohere.ai multilingual 22-12
MSMARCO-WebSearch	IP	768 (float)	9374	-62	Clicked document-query pairs from the ClueWeb22 website corpus [64]	SimANS

Table 6.1: Descriptions of each dataset used in our experiments, along with the radius used for range search. Every dataset is publicly available [26, 156]. The SSNPP dataset is a range search benchmark, while the other datasets were originally published as top-k benchmarks. Unless otherwise indicated, the reference in the "Source" column covers all attributes of the dataset.

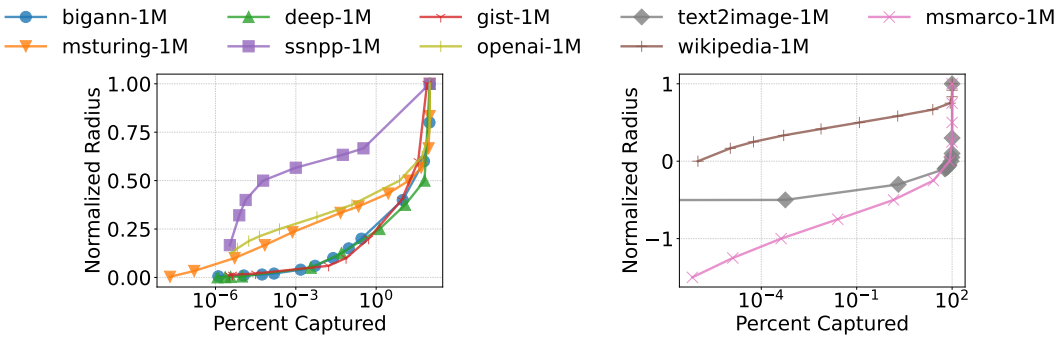


Figure 6.2: Plots of radius versus percent captured for each dataset: Euclidean datasets on the left and inner product datasets on the right. Radius is normalized to enable displaying multiple datasets on one plot. Note that inner product values can be less than zero, so negative ranges of radius are present on the right-hand figure.

Dataset	0	$\leq 10^1$	$\leq 10^2$	$\leq 10^3$	$\leq 10^4$	$\leq 10^5$
BIGANN-1M	9728	143	84	45	0	0
BIGANN-10M	9590	173	99	93	45	0
BIGANN-100M	9413	212	136	101	93	45
DEEP-1M	9923	56	19	2	0	0
MSTuring-1M	95716	2443	20	21	0	0
GIST-1M	8487	830	160	143	134	246
SSNPP-1M	97422	2424	254	0	0	0
SSNPP-10M	91575	7310	954	161	0	0
SSNPP-100M	80795	13719	4357	971	158	0
OpenAI-1M	7030	2564	372	34	0	0
Text2Image-1M	99327	669	4	0	0	0
Wikipedia-1M	4445	482	73	0	0	0
Wikipedia-10M	3385	1328	277	10	0	0
MSMARCO-1M	7022	2199	152	3	0	0

Table 6.2: Table showing the distribution of result sizes for each dataset. Note that not all datasets have the same number of query points; see Table 6.1 for the number of queries and corresponding radius for each dataset. No data point had more than 10^5 results.

capture to the smallest radius achieving 100% capture. As shown in the figure, there is wide variation in each dataset's response to perturbations in the radius. Datasets BIGANN, GIST, DEEP, Wikipedia, and MSMARCO are the most robust to perturbations in the interesting range (for million size datasets, the applicable range is around 10^{-6} to 10^{-5}, as most query points will have no results). In these datasets, it is thus less likely that there will be significant numbers of points near the boundary of the ball of radius r, and thus less work to distinguish points that are just inside the boundary from those which are just outside the boundary. We will see this insight at work in the experimental results in Section 6.4.

Using the insights from this experiment, we choose an appropriate radius for each dataset. The real (non-normalized) choices for each radius are shown in Table 6.1.

6.2.2 Frequency Distribution of Matches

Given a choice of radius for each dataset, we now evaluate the frequency distribution of the number of range results for each query point. The distributions are found in Table 6.2. We observe a general pattern of most queries having no results and a few large outliers, which is more pronounced in BIGANN, DEEP, MSTuring, and Text2Image, and less pronounced in SSNPP, OpenAI, Wikipedia, and MSMARCOWebSearch. GIST is unusual among the datasets in having many extremely large outliers. For BIGANN, SSNPP, and Wikipedia, we also show the frequency distribution for a larger version of the same base dataset. Since these are effectively larger samples from the same distribution, we observe an increase in density of range results.

Algorithm 4: BeamSearch($q, G, \mathcal{P}, S, r, b, \mathcal{M}$).

Input: Query point q, graph G, point set $\mathcal{P}$, starting points S, radius r, beam size b, early stopping metric $\mathcal{M}$

Output: Set $\mathcal{B}$ of closest points, set $\mathcal{V}$ of visited points

$\mathcal{B} \leftarrow S, \mathcal{V} \leftarrow \emptyset$

while $\mathcal{B} \setminus \mathcal{V} \neq \emptyset$ **do**

  $p^* \leftarrow \arg\min_{(p \in \mathcal{B} \setminus \mathcal{V})} \|p, q\|$

  `// Early stopping conditions`

  if $\mathcal{M}(q, \mathcal{B}, \mathcal{V}, r)$ **then Break**

  $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$

  $\mathcal{B} \leftarrow \mathcal{B} \cup (N_{out}(p^*) \setminus \mathcal{V})$

  if $|\mathcal{B}| > b$ **then** trim $\mathcal{B}$ to size b, keeping the closest points to q

return $(\mathcal{B}, \mathcal{V})$

6.3 Algorithms for Range Search on Graph-Based ANNS Indices

In this section we introduce our range search algorithms. We first introduce the data structure and baseline algorithm, a standard beam search. Since the standard beam search is well known to produce excellent results for top-k queries, some range queries (those with an intermediate number of results) can already be efficiently answered with top-k search with a small k. Thus, in this section we focus on improving queries outside this range, both queries with many results and queries with no results. Intuitively, both types of queries call for some dynamic adjustments to the beam search: for queries with no results, ideally terminating quickly before the end of the search, and for queries with many results, dynamically extending the size of the beam in some way.

6.3.1 Data Structure and Baseline Algorithm

As the aim of this chapter is to investigate whether ANNS graphs can be effectively reused for range search, the data structure used for range queries is an ANNS graph. There are many examples of ANNS graphs in the literature [84, 126, 161]. For the experiments in this chapter we use the in-memory construction of DiskANN, specifically the ParlayANN library [127], but the algorithms we present should work on any single-layer ANNS graph that performs well using the standard beam search, and should be modifiable to multi-layer graphs such as HNSW with little effort.

For our baseline algorithm, due to lack of existing algorithms for range search in high dimensions, we use standard beam search on an ANNS graph. This algorithm is described in detail in many publications, and we also show pseudocode in Algorithm 4.

Algorithm 5: GreedySearch($q, G, \mathcal{P}, S, r$).

Input: Query point q, graph G, point set $\mathcal{P}$, starting points S, radius r
Output: Set $\mathcal{V}$ of visited points

$\mathcal{F} \leftarrow S, \mathcal{V} \leftarrow \emptyset$
while $\mathcal{F} \setminus \mathcal{V} \neq \emptyset$ **do**
  $p^* \leftarrow \arg\min_{(p \in \mathcal{F} \setminus \mathcal{V})} \|p, q\|$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$ **foreach** $f \in (N_{out}(p^*) \setminus \mathcal{V})$ **do**
    if $\|f, q\| \leq r$ **then** $\mathcal{F} = \mathcal{F} \cup \{f\}$
return $\mathcal{V}$

Algorithm 6: EarlyStopping($q, \mathcal{B}, \mathcal{V}, \mathcal{M}, \mathcal{C}$).

Input: Query point q, current beam $\mathcal{B}$, visited set $\mathcal{V}$, early stopping metric $\mathcal{M}$, cutoff parameter set $\mathcal{C}$
Output: Bool value for early stopping. True if conditions are met.

if $\mathcal{M}(q, \mathcal{B}, \mathcal{V}, \mathcal{C})$ **then return** True

6.3.2 Improving Range Search for Queries with Many Results

We first investigate how to approach graph-based range search for queries that have many results. A standard beam search can only capture as many points as are contained within the beam of size B, and thus any queries with significantly more than B results cannot achieve high accuracy. The natural question is whether it is possible to distinguish between such queries with more than B results and expand the set of returned candidates for those queries, without wasting time on queries with fewer than B results. In this section we discuss two algorithmic approaches to this idea.

Doubling Beam Search The first, and perhaps most natural, algorithm extending beam search to queries with variable number results is a dynamic beam search: that is, run the beam search with some starting beam B for a certain number of steps, then either terminate or double the beam and continue searching for more candidates. The algorithm terminates if fewer than some fraction λ of returned candidates are valid range candidates, and continues otherwise. In practice, we found that the value of λ did not change the results very much, so we set it to 1 as a rule. That algorithm is shown using beam search as a subroutine in Algorithm 8.

Greedy Search Another approach to dynamically resizing the queue length is to run a beam search with *unbounded* queue length, but much stricter criteria on which nodes are explored. In this approach, candidates are first generated by an initial beam search with beam B. If the number of candidates within the radius falls above a certain threshold (again, some fraction λ of the candidates returned, which we found in practice not to matter significantly), the search moves onto this so-called "greedy search," which has an unbounded queue but only adds nodes to the queue if they are within the ball of radius r around the query point for the specified radius. This approach takes advantage of the fact that nodes in the same ball of radius r are likely to be in connected clusters, and avoids the overhead of the doubling beam search by performing very

Algorithm 7: EarlyStoppingExample($q, \mathcal{B}, \mathcal{V}, v, \mathcal{C} = \{r, vl, esr\}$).

Input: Query point q, current beam $\mathcal{B}$, visited set $\mathcal{V}$, number of visited points v, radius r, limit of number of visits vl, early stopping radius esr

Output: Bool value for early stopping.

```
// When the closest point in the beam is not within the radius
```

return ($\arg\min_{p \in \mathcal{B}} \|p, q\| > r$, **and**

```
// When the number of visited points is larger than the given
   cutoff
```

$v \geq vl$, **and**

```
// When the current point visited is not within the early
   stopping radius
```

$\arg\min_{p^* \in \mathcal{V} \setminus \mathcal{B}} \|p^*, q\| > esr$)

Algorithm 8: DoublingSearch($q, G, \mathcal{P}, S, r, b, \mathcal{M}$).

Input: Query point q, graph G , point set $\mathcal{P}$, starting points S, radius r, beam size b, early stopping metric $\mathcal{M}$

Output: Set of neighbors $\mathcal{N}$ witin range

while *true* do
- $\mathcal{N} \leftarrow \emptyset$
- `// Early stopping is done inside the beam search`
- $(\mathcal{B}, \mathcal{V}) \leftarrow BeamSearch(q, G, \mathcal{P}, S, r, \mathcal{M}, b)$
- $S \leftarrow S \cup \mathcal{V}$
- **for** $n \in \mathcal{B}$ **do**
  - **if** $\|n, q\| \leq r$ **then** $\mathcal{N} = \mathcal{N} \cup \{n\}$
- **if** $|\mathcal{N}| < b$ **then** **Break**
- $b \leftarrow 2 \times b$

return $\mathcal{N}$

few distance comparisons to points which are not valid results. See Algorithm 5 for a complete description, and Section 6.4 for experimental analysis.

6.3.3 Improving Range Search for Queries with No Results

This section introduces an algorithm which improves range search for queries with no results. We start from the established paradigm of starting a search with fixed beam B and continuing with Algorithm 5 or Algorithm 8 if the returned list contains only candidates within the radius. It would be ideal to terminate the initial beam search early if the query is not finding any results, and the termination condition should use terms that are already computed during the beam search, or inexpensive to compute. A 2020 paper from Li, Zhang, Anderson, and He [121] addresses a similar question—finding conditions for terminating top-k search early—and finds that several existing terms can be used to predict whether a search can terminate early. Those terms included the coordinates of the query point, the distance from the query point to the current top-k neighbor for fixed k, and the ratio of the distance from the query point to top-k neighbor and the distance

Algorithm 9: GreedyRangeSearch($Q, G, \mathcal{P}, S, r, b, \mathcal{M}$).

Input: Query points Q, graph G, point set $\mathcal{P}$, starting points S, radius r, beam size b, early stopping metric $\mathcal{M}$

Output: Set of neighbors $\mathcal{V}$ within range for each of the query points

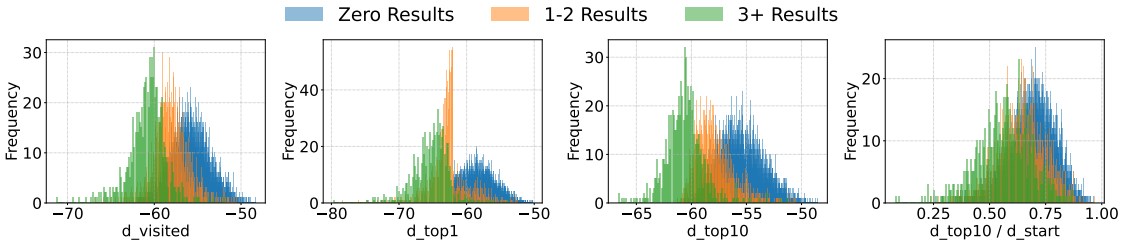
parallel for $q \in Q$ **do**
  $\mathcal{N} \leftarrow \emptyset$
  $(\mathcal{B}, \mathcal{V}) \leftarrow BeamSearch(q, G, \mathcal{P}, S, r, b, \mathcal{M})$
  `// Early stopping is done inside the beam search`
  foreach $n \in \mathcal{B}$ **do**
    if $\|n, q\| \leq r$ **then** $\mathcal{N} = \mathcal{N} \cup \{n\}$
  if $|\mathcal{N}| < b$ **then** $\mathbf{V}[q] \leftarrow \mathcal{N}$
  else
    `// Add results from beam search as starting points for greedy search`
    $\mathcal{V}[q] \leftarrow GreedySearch(Q, G, \mathcal{P}, \mathcal{N}, r)$
return $\mathcal{V}$

from the query point to the start node.

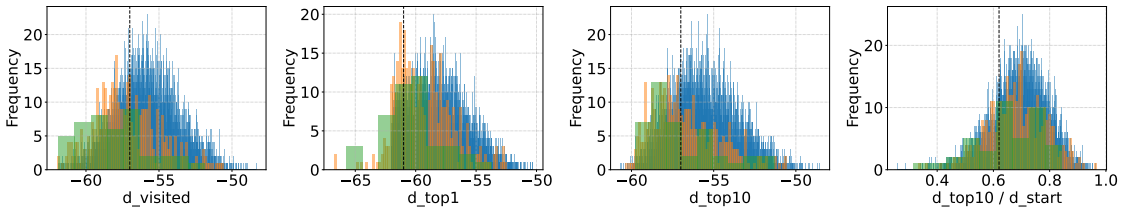
To apply these ideas to range search, the key question is whether any of these metrics are effective at distinguishing queries with zero results from queries with one or more result. To investigate this, we look at each dataset and separate each query set into three groups: those with no results, those with 1-2 results, and those with more than three results. Then, we histogram the values of these metrics at different steps in the beam search, distinguishing between these three groups. We evaluate the following metrics, the last three of which are proposed in [121].

- d_{visited}: distance from query point q to the point being visited at step i of a beam search.
- d_{top1}: distance from query point q to the closest known neighbor of q at step i of a beam search.
- d_{top10}: distance from query point q to the tenth-closest known neighbor at step i of a beam search.
- $d_{\text{top10}}/d_{\text{start}}$: ratio of d_{top10} to the distance from query point q to the starting point of the beam search.

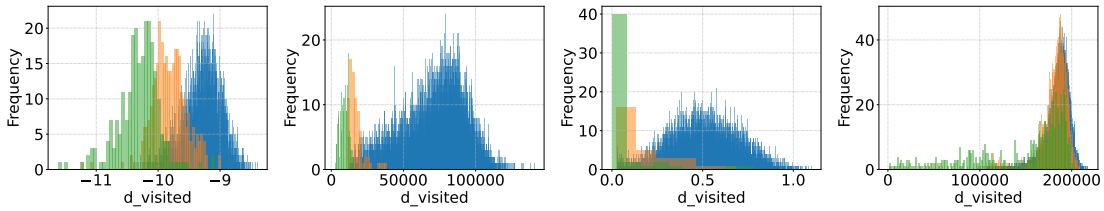
Figure 6.3a histograms the metrics above for MSMARCOWebSearch-1M at step 20 of a beam search with beam 100. Algorithm 6 shows a natural and general algorithm for imposing an early stopping condition on range search, using any one of these metrics. In all cases, when a search has already found at least one candidate within the specified range, the search does not terminate early. Otherwise, the early stopping condition is applied after a certain number of steps in the beam search. The algorithm terminates when the chosen metric is above a pre-specified cutoff. Thus, in Figure 6.3a, we show the same histograms as in Figure 6.3b, with points where the search has already found a point inside the radius excluded. The figure also shows vertical lines indicating a potential cutoff for Algorithm 6; points with more than one range result to the right of the line would be "incorrectly" cut off from further search, while points with zero points to the right of the line would be "incorrectly" allowed to proceed instead of terminating early. The goal is to identify a cutoff that yields a more favorable QPS/recall curve. Figure 6.3c shows



(a) Histograms plotting selected early stopping metrics for all query points using dataset MSMARCOWebSearch-1M.



(b) Histograms plotting selected early stopping metrics on dataset MSMARCOWebSearch-1M, excluding query points which have already found a candidate within the radius.



(c) Histograms showing the value of d_{visited} on more selected datasets. From left to right: Wikipedia-1M, BIGANN-1M, DEEP-1M, and SSNPP-1M.

Figure 6.3: Histograms of early stopping metrics for selected metrics and datasets. All metric values were taken at step 20 of a beam search with beam 100. Queries are separated by color based on the number of range results.

one choice of metric—d_{visited} on four different datasets.

Based on this data, a larger set of which is shown in Section , we come to three conclusions: first, some datasets show potential for significant improvement using early stopping, while others do not. Second, if a dataset shows potential for improvement using one metric, it also tends to show improvement for all other metrics. Third, using d_{visited} seems to be the best early stopping metric, when examined using the exclusion criteria in Figure 6.3b. We hypothesize that d_{visited} is the best metric as it is the most frequently updated and contains the most granular information, but the positive results from other metrics suggests that approaches using machine learning such as those in the Li et al. paper [121] to combine the information from all of these metrics might also be useful. We present experimental analysis of this algorithm in Section 6.4. See Algorithm 7 for an example of how to apply the d_{visited} metric during a search, and Algorithm 9 for an example of the end-to-end algorithm using greedy search.

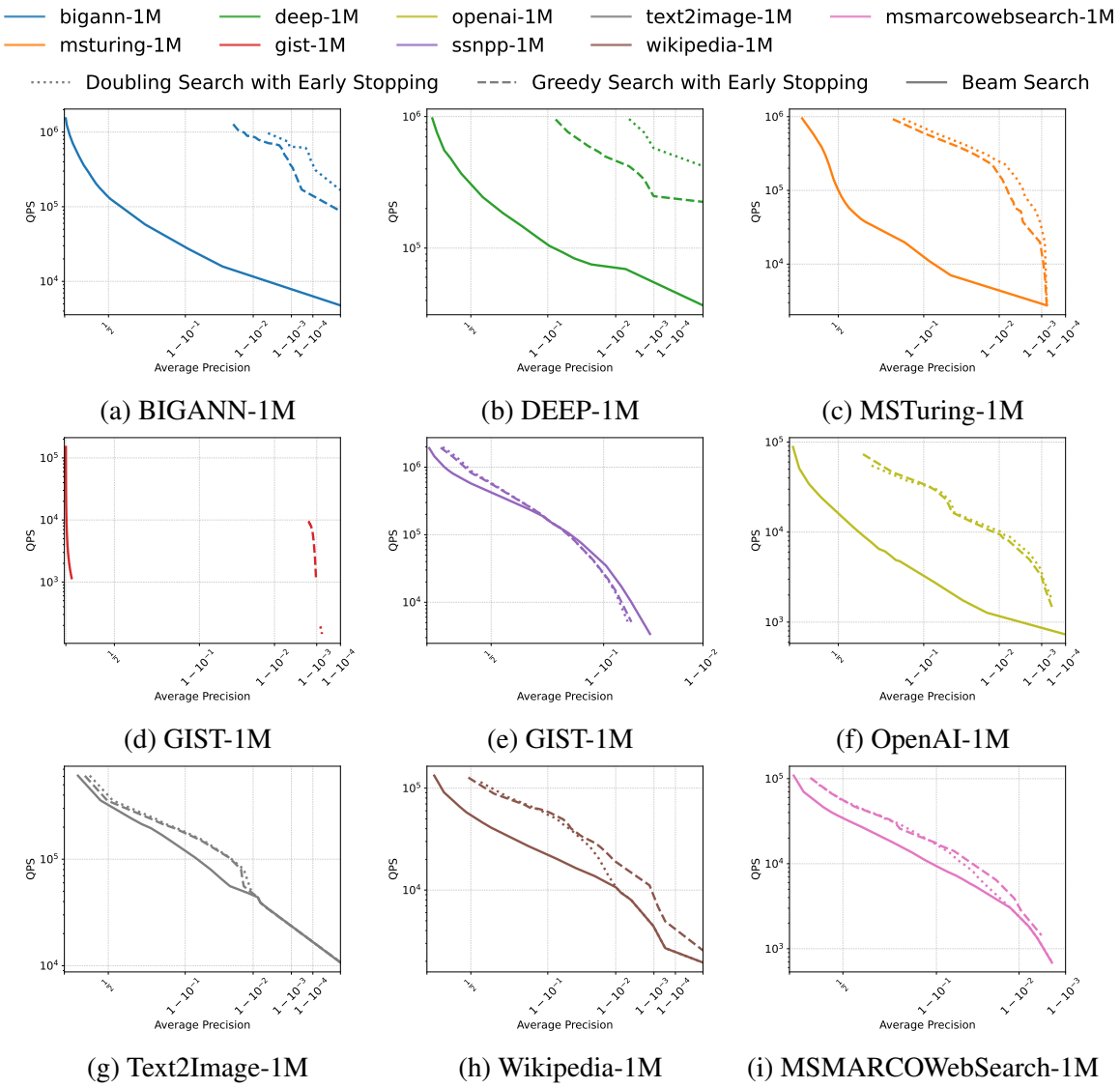
6.4 Experimental Results

In this section, we conduct an experimental evaluation of our algorithms on the nine datasets covered in Section 6.2. In addition to plotting the QPS and precision tradeoffs for each dataset, we evaluate the time breakdown of each algorithm and draw conclusions about the reasons for their effectiveness.

Experimental Setup and Code Our algorithms were implemented as extensions of the ParlayANN [128] library, which is implemented in C++ using ParlayLib [49] for fork-join parallelism as well as standard building blocks such as sorting and filtering. The code will be made available after the anonymous review period. All experiments were conducted on an Azure Standard_L32s_v3 virtual machine, with a 3rd Generation Intel® Xeon® Platinum 8370C (Ice Lake) processor in a hyper-threaded configuration. It has 32 vCPUs available to the user and a 256 GB main memory. Experiments use all 32 vCPUs unless otherwise stated.

To produce a curve of QPS versus average precision, the starting beamwidth is varied on a number of searches and the Pareto frontier is reported.

QPS and Precision Tradeoffs We begin our experimental evaluation by plotting queries per second (QPS) versus precision for all nine datasets. Data on all datasets except SSNPP can be found in Figure 6.4; SSNPP is shown in Figure 6.5, which illustrates the effects of size scaling on the algorithms using datasets SSNPP and BIGANN. Overall, our algorithms generate improvement over the naive baseline on every dataset, but we find significant diversity in both the magnitude of the improvement and the relative ordering of the two algorithms presented in Section 6.3. We find that DEEP, BIGANN, and MSTuring experience massive improvement over the baseline (close to 100x improvement in throughput in some cases), with the doubling beam search slightly faster than the greedy search on the 1 million scale, but significantly slower at the 100 million scale. OpenAI also sees large improvement in the realm of one order of magnitude, with greedy search and beam search having virtually identical performance. Wikipedia and MSMarco experience decent improvement with the greedy search somewhat outperforming doubling beam search. GIST achieves such significant improvement over the baseline due to its

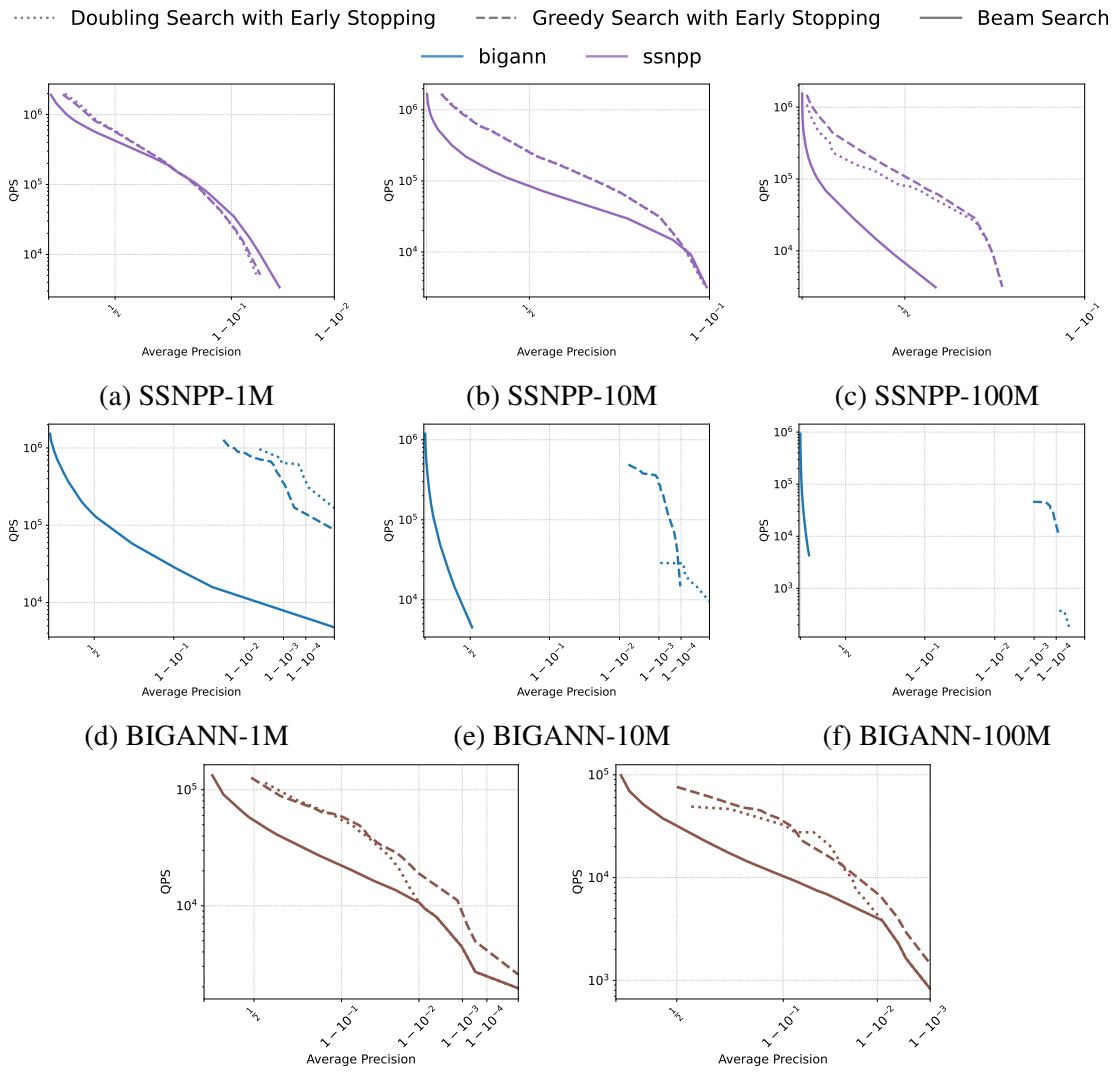


(a) BIGANN-1M (b) DEEP-1M (c) MSTuring-1M

(d) GIST-1M (e) GIST-1M (f) OpenAI-1M

(g) Text2Image-1M (h) Wikipedia-1M (i) MSMARCOWebSearch-1M

Figure 6.4: Average precision vs QPS for eight datasets and three range search algorithms. For GIST-1M, the lines for doubling search and greedy search are very short due to even the smallest of initial beam sizes producing recall in the .999 range.

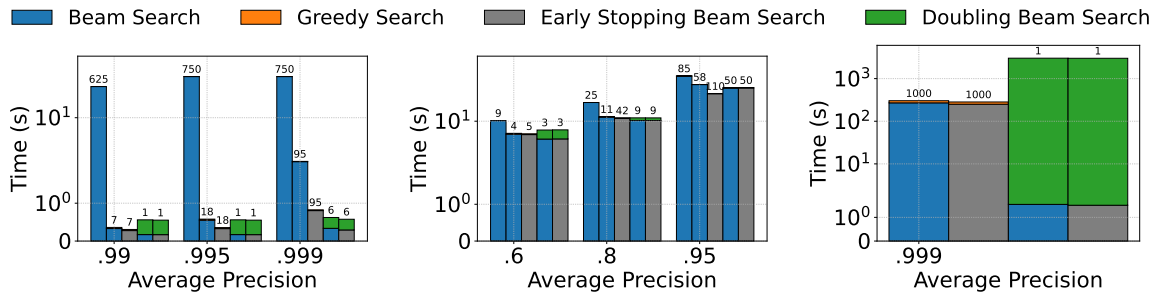


(a) SSNPP-1M (b) SSNPP-10M (c) SSNPP-100M

(d) BIGANN-1M (e) BIGANN-10M (f) BIGANN-100M

(g) Wikipedia-1M (h) Wikipedia-10M

Figure 6.5: Average precision vs QPS showing SSNPP, BIGANN, and Wikipedia at different scales.



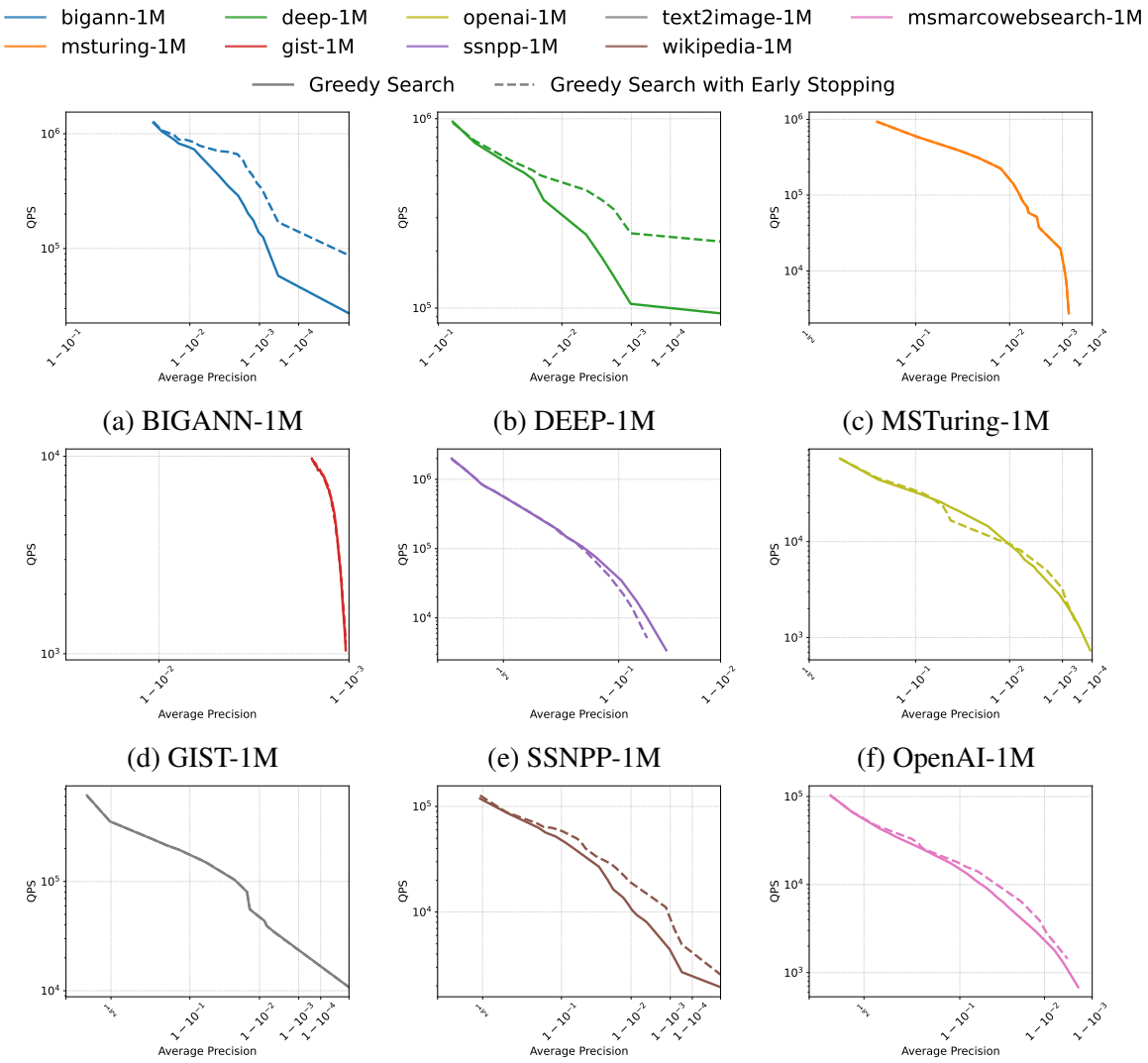
(a) BIGANN-1M (b) MSMARCOWebSearch-1M (c) GIST-1M

Figure 6.6: Figures breaking down the cost in seconds (single threaded time) of each type of search for three datasets for selected average precision. The label at the top of each column indicates the beam width of the initial search. Each collection from left to right shows: the beam search baseline, beam search followed by greedy search, beam search with early stopping followed by greedy search, beam search followed by doubling beam search, and beam search with early stopping followed by doubling beam search. Note that in some cases, the time spent on the second phase of search is so short that it is not visible. GIST-1M has only one selected recall, as every search setting yields .999 recall within a .001 tolerance; the beam search baseline is not present for GIST, since it cannot achieve the desired recall (see Figure 6.4d).

outlier status with respect to the frequency distribution of its matches, as it is the only dataset with hundreds of points with more than 10000 range results. Greedy search also significantly outperforms doubling beam search for GIST. Text2Image sees modest improvement below 90% recall. SSNPP, which we investigate for three different sizes, first shows little to no improvement at the 1 million scale, then some improvement at the 10 million scale, then significant improvement at size 100 million.

Scaling Figure 6.5 examines the performance of our algorithms on three datasets. Similarly to existing range search benchmarks, we use the same radius for each data scale, so each order of magnitude effectively represents a more dense space, and the average number of results per query point increases. Unsurprisingly, the comparative advantage of our algorithms increases significantly over the baseline, which is only capable of returning up to about 1000 points per query before becoming hopelessly inefficient. Interestingly, we also see that the greedy search seems to have a significant advantage over the doubling beam search as the density of the dataset increases (this is also true of GIST-1M, which has some extraordinarily dense outliers). This is likely because in denser graphs, the greedy search is particularly adept at efficiently finding all points within the radius, which are likely to be in a dense connected cluster in the graph structure.

Analyzing Time Spent on Algorithm Components To aid in our analysis, we investigate the time spent on different phases of each algorithm in Figure 6.6. Each algorithm consists of an initial beam search phase—either with or without the early stopping criteria in Algorithm 6—and then a following phase of either greedy search or successive calls to doubling beam search. We show this visual breakdown for three datasets at selected fixed average precision: BIGANN-1M,

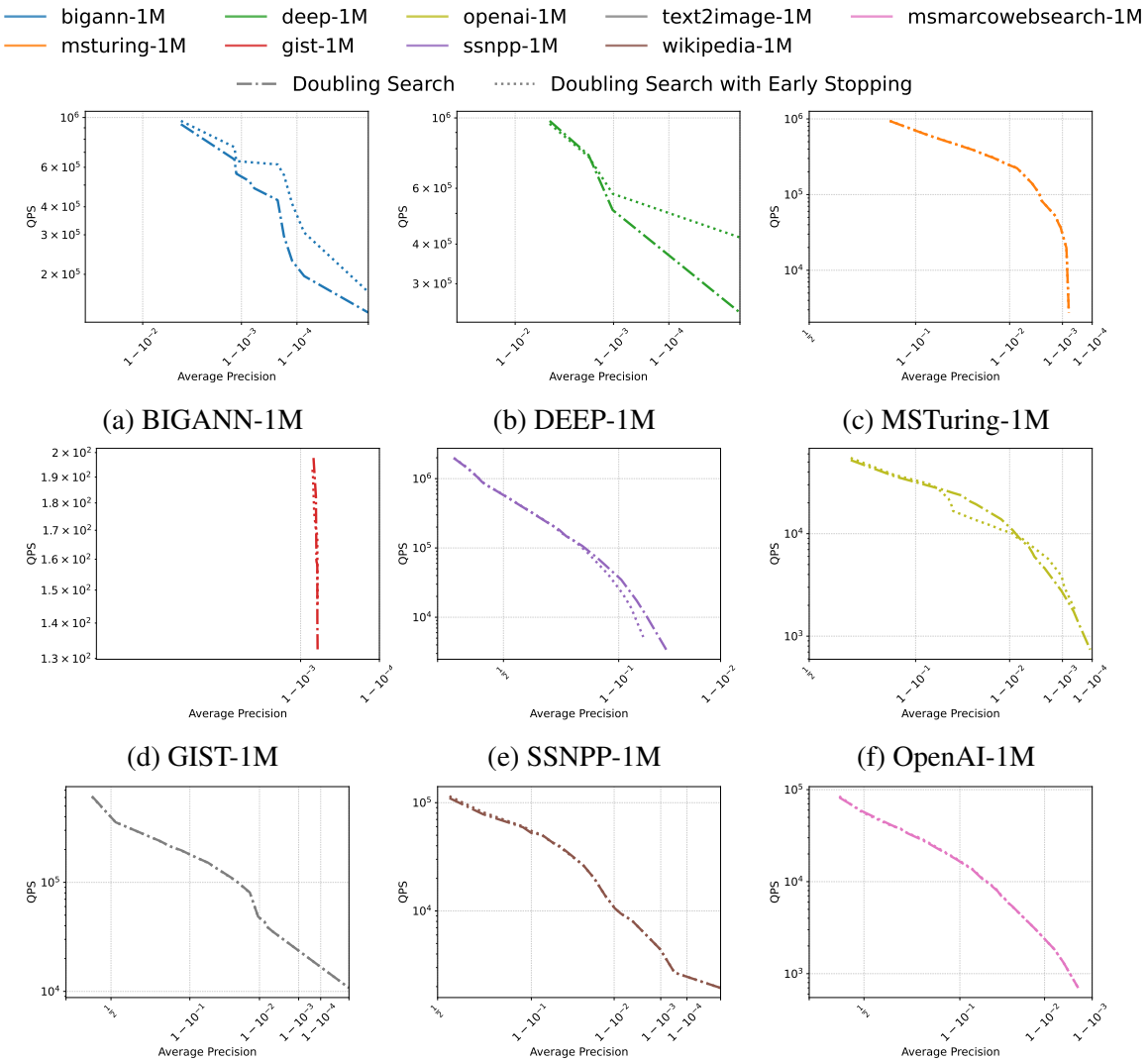


(a) BIGANN-1M (b) DEEP-1M (c) MSTuring-1M

(d) GIST-1M (e) SSNPP-1M (f) OpenAI-1M

(g) Text2Image-1M (h) Wikipedia-1M (i) MSMARCOWebSearch-1M

Figure 6.7: Average precision vs QPS for all nine datasets using greedy search, comparing use of early stopping versus without early stopping.



(a) BIGANN-1M (b) DEEP-1M (c) MSTuring-1M

(d) GIST-1M (e) SSNPP-1M (f) OpenAI-1M

(g) Text2Image-1M (h) Wikipedia-1M (i) MSMARCOWebSearch-1M

Figure 6.8: Average precision vs QPS for all nine datasets using doubling search, comparing use of early stopping versus without early stopping.

MSMARCOWebSearch-1M, and GIST-1M.

The time breakdowns provide significant insight into the results in Figures 6.4 and 6.5. First, note that for all datasets, the initial beam search phase is a sizable portion of the overall cost of computation. This is because the initial beam search takes place on all points, but further searches only take place on points with one or more valid range result (since the distances used are exact, a point with zero results will never pass on to the second round of computation).

Greedy Search vs Doubling Search Next, let us examine the differences between doubling search and greedy search. Doubling search spends a significantly longer portion of time in the second phase of search compared to greedy search, but is capable of achieving higher recall from a smaller starting beam. The characteristics of each individual dataset seem to determine which approach provides the best tradeoff between QPS and average precision. Greedy search requires a larger initial beam to achieve a particular recall value, but the second round of search is extremely cheap as it only visits points within the radius. On the other hand, doubling beam search requires a smaller initial beam, but spends more time in the second phase of search, which performs comparatively more distance comparisons to points which are not valid range results. It also has some overheads related to repeated calls to the beam search function. Based on these factors and the experimental results, it appears that doubling search dominates greedy search when a) a small initial beam is sufficient for doubling search to achieve high recall, and b) a fairly small number of rounds of doubling search are required. These conditions are met with datasets BIGANN-1M (and DEEP-1M, although its time breakdown is not pictured), but GIST-1M requires too many rounds of doubling, and MSMARCO-1M requires too high an initial beam. We also see that in the scaling studies in Figure 6.5, the greedy search overtakes the doubling search as the density of matches increases, also likely because many rounds of doubling search become necessary.

Effects of Early Stopping Finally, we consider the benefit of early stopping. Early stopping clearly decreases the time taken for a fixed beam; for example, in Figure 6.6a, early stopping beam search with beam 95 takes less than half the time compared to beam search with beam 95 without early stopping. On the other hand, as illustrated in Figure 6.6b, early stopping can make the search significantly less accurate due to points with valid range results being terminated too early. The effects of early stopping on selected datasets are shown in Figures 6.7 and 6.8; overall, early stopping is beneficial on datasets where significant separation exists between the distributions of points with no results and points with one or more results, and not beneficial otherwise. We also find that early stopping is more beneficial for greedy search than for doubling search. This is likely because doubling search uses a smaller initial beam to get to the same recall, and since the distance calculations are exact, early stopping is only useful during the first round of search. Since the beam on the first round of search is quite small, the number of steps required to use the early stopping metric may not be reached as often as it is with greedy search.

Comparison with Top-k Search We add a final note on comparison with top-k search. To investigate this, we ran top-10 searches on OpenAI-1M and MSTuring-1M, and measured the QPS at 90% 10@10 recall. For OpenAI-1M, we found the QPS at 90% recall was around 5000,

compared to around 10000 QPS for range search at 90% precision. For MSTuring-1M, the QPS at 90% recall for top-10 search was about 30000, compared to over 100000 for range search using our algorithms. This suggests that range benchmarking may actually be thought of as an easier problem than top-k search.

6.5 Investigation of Range Search Metadata

Since to the best of our knowledge there is only one high-dimensional embedding dataset designed specifically for range search, in this work we chose a radius for a variety of embedding datasets. However, since the radius was chosen somewhat artificially, there is some doubt about whether the radius indicates a meaningful boundary. In order to investigate this, we use one dataset, the OpenAI ArXiv dataset consisting of embeddings of abstracts from ArXiv articles using the cohere.ai multilingual 22-12 model, where we had access to the underlying title and abstract metadata. Since the title and abstract metadata was available only for the base embeddings and not the query embeddings, we created a new query embedding set consisting of 10000 abstracts, which were then withheld from the training set. For this query set we chose a radius of .16, which gave a similar distribution to the one shown in Table 6.2.

After computing the ground truth with respect to the chosen radius, we performed the following experiment: we partitioned the dataset into three groups: a group containing all queries with no results within the radius, a group containing queries with 1-100 results within the radius, and a group containing queries with 100+ results. For the group with no results, we randomly selected two elements and computed their top-2 nearest neighbors, viewing the metadata of the query and the top-2 neighbors. For the two groups with results, we randomly selected two elements within the radius, and two elements from the next k nearest neighbors of the query point, where k is the number of results within the radius of that particular point. The titles and abstracts of each of the six examples are displayed below. Some of the LaTeX formatting within the metadata was not correct, so we have reconstructed it to the best of our ability. Still, some small errors may remain.

Analysis The results of the investigation are somewhat inconclusive. Examination of the six examples does not yield a result where the points outside the radius are substantially less similar than the points within the radius. Whether a more subtle distinction exists is a difficult and subjective determination. Furthermore, there is no way to conclude that the inconclusive result is because the radius is not meaningful as opposed to, say, the embedding model not being capable of the specialized inference needed to rank academic abstracts from least to most relevant. We leave this problem to further study and the development of more specialized datasets for range search for future work.

Query 1, with no results within radius

Query 1 Title: 'Improved parallel WaveGAN vocoder with perceptually weighted spectrogram loss'

Query 1 Abstract: 'This paper proposes a spectral-domain perceptual weighting technique for Parallel WaveGAN-based text-to-speech (TTS) systems. The recently proposed Parallel

WaveGAN vocoder successfully generates waveform sequences using a fast non-autoregressive WaveNet model. By employing multi-resolution short-time Fourier transform (MR-STFT) criteria with a generative adversarial network, the light-weight convolutional networks can be effectively trained without any distillation process. To further improve the vocoding performance, we propose the application of frequency-dependent weighting to the MR-STFT loss function. The proposed method penalizes perceptually-sensitive errors in the frequency domain; thus, the model is optimized toward reducing auditory noise in the synthesized speech. Subjective listening test results demonstrate that our proposed method achieves 4.21 and 4.26 TTS mean opinion scores for female and male Korean speakers, respectively.'

Top-1 Result Title: 'Speech Dereverberation Using Fully Convolutional Networks'

Top-1 Result Abstract: 'Speech derverberation using a single microphone is addressed in this paper. Motivated by the recent success of the fully convolutional networks (FCN) in many image processing applications, we investigate their applicability to enhance the speech signal represented by short-time Fourier transform (STFT) images. We present two variations: a "U-Net" which is an encoder-decoder network with skip connections and a generative adversarial network (GAN) with U-Net as generator, which yields a more intuitive cost function for training. To evaluate our method we used the data from the REVERB challenge, and compared our results to other methods under the same conditions. We have found that our method outperforms the competing methods in most cases. '

Top-2 Result Title: 'Enhancement and Recognition of Reverberant and Noisy Speech by Extending Its Coherence'

Top-2 Result Abstract: 'Most speech enhancement algorithms make use of the short-time Fourier transform (STFT), which is a simple and flexible time-frequency decomposition that estimates the short-time spectrum of a signal. However, the duration of short STFT frames are inherently limited by the nonstationarity of speech signals. The main contribution of this paper is a demonstration of speech enhancement and automatic speech recognition in the presence of reverberation and noise by extending the length of analysis windows. We accomplish this extension by performing enhancement in the short-time fan-chirp transform (STFChT) domain, an overcomplete time-frequency representation that is coherent with speech signals over longer analysis window durations than the STFT. This extended coherence is gained by using a linear model of fundamental frequency variation of voiced speech signals. Our approach centers around using a single-channel minimum mean-square error log-spectral amplitude (MMSE-LSA) estimator proposed by Habets, which scales coefficients in a time-frequency domain to suppress noise and reverberation. In the case of multiple microphones, we preprocess the data with either a minimum variance distortionless response (MVDR) beamformer, or a delay-and-sum beamformer (DSB). We evaluate our algorithm on both speech enhancement and recognition tasks for the REVERB challenge dataset. Compared to the same processing done in the STFT domain, our approach achieves significant improvement in terms of objective enhancement metrics (including PESQ—the ITU-T standard measurement for speech quality). In terms of automatic speech recognition (ASR) performance as measured by word error rate (WER), our experiments indicate that the STFT with a long window is more effective for ASR. '

Query 2, with no results within radius

Query 2 Title: 'Condon domains - these non-magnetic diamagnetic domains'

Query 2 Abstract: 'The paper, not pretending for a complete and detailed review, is intended mainly for a wide community of physicists, not only specialists in this particular subject. The author gives a physical picture of the periodic emergence of instabilities and well-known diamagnetic domains (Condon domains) in metals resulting from the strong de Haas-van Alphen effect. The most significant experiments on observation and study of the domain state in metals are described. In particular, the recent achievements in this area using muon spin rotation, as well as the amazing phenomenon of "supersoftness" observed in the magnetostriction experiments, are presented. Novel, not previously discussed features of the phenomenon related to the metal compressibility are enlightened. '

Top-1 Result Title: 'Metastability in the formation of Condon domaines'

Top-1 Result Abstract: 'Metastability effects in the formation of Condon non-spin magnetic domains are considered. A possibility for the first-order phase transition occurrence in a three-dimensional electron gas is described in the case of two-frequency de-Haas-van Alphen magnetization oscillations originating from two extremal cross sections of the Fermi surface. The appearance of two additional domains is shown in the metastable region in aluminum. The phase diagram temperature-magnetic field exhibits the presence of second-order and first- order phase transitions in the two-frequency case. '

Top-2 Result Title: 'Morphology of Condon Domains Phase in Plate-Like Sample'

Top-2 Result Abstract: 'Based on Shoenberg assumption of magnetic flux density dependence of diamagnetic moments which accounts for an instability of strongly correlated electron gas at the conditions of dHvA effect and diamagnetic phase transition (DPT) to non-uniform phase, we investigate the morphology of the Condon domains (CD) in plate-like sample theoretically. At one period of dHvA oscillations the intrinsic structure of inhomogeneous diamagnetic phase (IDP) is governed by the first order phase transitions between different non-uniform phases similar to the high-anisotropy magnetic systems of spin origin, and strongly affected by temperature, magnetic field and impurity of the sample due to the electron correlations. The phase diagrams of evolution of IDP with temperature and small-scale magnetic field in every period of dHvA oscillations are calculated. '

Query 3, with 1-100 results within radius

Query 3 Title: 'Photo-production of the pentaquark Θ^+ with positive and negative parities'

Query 3 Abstract: 'We investigate the production of the pentaquark Θ^+ baryon via the $\gamma n \to K^- \Theta^+$ and $\gamma p \to \bar{K}^0 \Theta^+$ processes, focusing on the parity of the Θ^+. Using the effective Lagrangians, we calculate the total and differential cross sections with the spin of the Θ^+ presumed to be 1/2. We employ the coupling constant of the $KN\Theta$ vertex determined by assuming its mass and the decay width to be $1540\mathrm{MeV}$ and $15\mathrm{MeV}$. That of the $K^*N\Theta$ is taken to be about a half of the $KN\Theta$ coupling constant. We estimate the cutoff parameter by reproducing the total cross section of the $\gamma p \to K^+\Lambda$ reaction. It turns out that the total cross section for the $\gamma n \to K^- \Theta^+$ process is about four times larger than that of the $\gamma p \to \bar{K}^0 \Theta^+$. We also find that the cross sections for the production of the positive-parity Θ are about ten times as large as those for the negative-parity ones. '

Result Within Radius Title: 'Photoproduction of $Theta^+(1540, 1/2^+)$ reexamined with new theoretical information'

Result Within Radius Abstract: 'We reinvestigate the photoproduction of the exotic pen-

taquark baryon $Theta^+(1540, 1/2^+)$ from the gamma $N- > KbarTheta^+$ reaction process within the effective Lagrangian approach, taking into account new theoretical information on the K-N-Theta and K*-N-Theta coupling strengths from the chiral quark-soliton model (chiQSM). We also consider the crossing-symmetric hadronic form factor, satisfying the on-shell condition as well. Due to the sizable vector and tensor couplings for the vector kaon, $g_{K*NTheta}$ and $f_{K*NTheta}$, which are almost the same with the vector coupling $g_{KNTheta}$ 0.8 for the pseudoscalar kaon, the K*-exchange contribution plays a critical role in the photon beam asymmetries. '

Result Within Radius Title: 'High-resolution search for the Θ^+ pentaquark via a pion-induced reaction at J-PARC'

Result Within Radius Abstract: 'The pentaquark Θ^+ has been searched for via the $\pi^- p \to K^- X$ reaction with beam momenta of 1.92 and 2.01 GeV/c at J-PARC. A missing mass resolution of 2 MeV (FWHM) was achieved but no sharp peak structure was observed. The upper limits on the production cross section averaged over the scattering angle from 2° to 15° in the laboratory frame were found to be less than 0.28 μb/sr at the 90% confidence level for both the 1.92- and 2.01-GeV/c data. The systematic uncertainty of the upper limits was controlled within 10%. Constraints on the Θ^+ decay width were also evaluated with a theoretical calculation using effective Lagrangian. The present result implies that the width should be less than 0.36 and 1.9 MeV for the spin-parity of $1/2^+$ and $1/2^-$, respectively. '

Result Outside Radius Title: 'Vector and axial-vector structures of the $Theta^+$'

Result Outside Radius Abstract: 'We present in this talk recent results of the vector and axial-vector transitions of the nucleon to the pentaquark baryon $Theta^+$, based on the SU(3) chiral quark-soliton model. The results are summarized as follows: $K^* - N - Theta$ vector and tensor coupling constants turn out to be $g_{K^*NTheta} = 0.81$ and $f_{K^*NTheta} = 0.84$, respectively, and the KN Theta axial-vector coupling constant to be $g_A^* = 0.05$. As a result, the total decay width for $Theta^+$ to NK becomes very small: $Gamma_{Theta->NK} = 0.71MeV$, which is consistent with the DIANA result $Gamma_{Theta->NK} = 0.36 + -0.11MeV$. '

Result Outside Radius Title: 'First observation of $\gamma\gamma \to p\bar{p}K^+K^-$ and search for exotic baryons in pK systems'

Result Outside Radius Abstract: 'The process $\gamma\gamma \to p\bar{p}K^+K^-$ and its intermediate processes are measured for the first time using a 980 fb^{-1} data sample collected with the Belle detector at the KEKB asymmetric-energy e^+e^- collider. The production of $p\bar{p}K^+K^-$ and a $\Lambda(1520)^0$ ($\bar{\Lambda}(1520)^0$) signal in the pK^- ($\bar{p}K^+$) invariant mass spectrum are clearly observed. However, no evidence for an exotic baryon near 1540 MeV/c^2, denoted as $\Theta(1540)^0$ ($\bar{\Theta}$ $(1540)^0$) or $\Theta(1540)^{++}$ ($\Theta(1540)^{--}$), is seen in the pK^- ($\bar{p}K^+$) or pK^+ ($\bar{p}K^-$) invariant mass spectra. Cross sections for $\gamma\gamma \to p\bar{p}K^+K^-$, $\Lambda(1520)^0\bar{p}K^+ + c.c.$ and the products $\sigma(\gamma\gamma \to \Theta(1540)^0\bar{p}K^+ + c.c.)BR(\Theta(1540)^0 \to pK^-)$ and $\sigma(\gamma\gamma \to \Theta(1540)^{++}\bar{p}K^- + c.c.)BR(\Theta(1540)^{++} \to pK^+)$ are measured. We also determine upper limits on the products of the χ_{c0} and χ_{c2} two-photon decay widths and their branching fractions to $p\bar{p}K^+K^-$ at the 90% credibility level. '

Query 4, with 1-100 results within radius

Query 4 Title: 'Strain fields in twisted bilayer graphene'

Query 4 Abstract: "Van der Waals heteroepitaxy allows deterministic control over lattice mismatch or azimuthal orientation between atomic layers to produce long wavelength superlattices. The resulting electronic phases depend critically on the superlattice periodicity as well as

localized structural deformations that introduce disorder and strain. Here, we introduce Bragg interferometry, based on four-dimensional scanning transmission electron microscopy, to capture atomic displacement fields in twisted bilayer graphene with twist angles $< 2\mathrm{deg}$. Nanoscale spatial fluctuations in twist angle and uniaxial heterostrain are statistically evaluated, revealing the prevalence of short-range disorder in this class of materials. By quantitatively mapping strain tensor fields we uncover two distinct regimes of structural relaxation – in contrast to previous models depicting a single continuous process – and we disentangle the electronic contributions of the rotation modes that comprise this relaxation. Further, we find that applied heterostrain accumulates anisotropically in saddle point regions to generate distinctive striped shear strain phases. Our results thus establish the reconstruction mechanics underpinning the twist angle dependent electronic behaviour of twisted bilayer graphene, and provide a new framework for directly visualizing structural relaxation, disorder, and strain in any moiré material. ”

Result Within Radius Title: ‘Electronic structure of transferred graphene/h-BN van der Waals heterostructures with nonzero stacking angles by nano-ARPES’

Result Within Radius Abstract: “In van der Waals heterostructures, the periodic potential from the Moiré superlattice can be used as a control knob to modulate the electronic structure of the constituent materials. Here we present a nanoscale angle-resolved photoemission spectroscopy (Nano-ARPES) study of transferred graphene/h-BN heterostructures with two different stacking angles of $2.4\mathrm{deg}$ and $4.3\mathrm{deg}$ respectively. Our measurements reveal six replicas of graphene Dirac cones at the superlattice Brillouin zone (SBZ) centers. The size of the SBZ and its relative rotation angle to the graphene BZ are in good agreement with Moiré superlattice period extracted from atomic force microscopy (AFM) measurements. Comparison to epitaxial graphene/h-BN with $0\mathrm{deg}$ stacking angles suggests that the interaction between graphene and h-BN decreases with increasing stacking angle. ”

Result Within Radius Title: ‘Angle-Dependent van Hove Singularities and Their Breakdown in Twisted Graphene Bilayers’

Result Within Radius Abstract: “The creation of van der Waals heterostructures based on a graphene monolayer and other two-dimensional crystals has attracted great interest because atomic registry of the two-dimensional crystals can modify the electronic spectra and properties of graphene. Twisted graphene bilayer can be viewed as a special van der Waals structure composed of two mutual misoriented graphene layers, where the sublayer graphene not only plays the role of a substrate, but also acts as an equivalent role as the top graphene layer in the structure. Here we report the electronic spectra of slightly twisted graphene bilayers studied by scanning tunneling microscopy and spectroscopy. Our experiment demonstrates that twist-induced van Hove singularities are ubiquitously present for rotation angles theta less than about 3.5o, corresponding to moiré-pattern periods D longer than 4 nm. However, they totally vanish for theta ¿ 5.5o (D ¡ 2.5 nm). Such a behavior indicates that the continuum models, which capture moiré-pattern periodicity more accurately at small rotation angles, are no longer applicable at large rotation angles. ”

Result Outside Radius Title: “Relaxation of moiré patterns for slightly misaligned identical lattices: graphene on graphite”

Result Outside Radius Abstract: “We study the effect of atomic relaxation on the structure of moiré patterns in twisted graphene on graphite and double layer graphene by large scale atomistic simulations. The reconstructed structure can be described as a superlattice of ‘hot

spots' with vortex-like behaviour of in-plane atomic displacements and increasing out-of-plane displacements with decreasing angle. These lattice distortions affect both scalar and vector potential and the resulting electronic properties. At low misorientation angles (¡$\sim$1°) the optimized structures deviate drastically from the sinusoidal modulation which is often assumed in calculations of the electronic properties. The proposed structure might be verified by scanning probe microscopy measurements. "

Result Outside Radius Title: 'Correlated Insulator Behaviour at Half-Filling in Magic Angle Graphene Superlattices'

Result Outside Radius Abstract: "Van der Waals (vdW) heterostructures are an emergent class of metamaterials comprised of vertically stacked two-dimensional (2D) building blocks, which provide us with a vast tool set to engineer their properties on top of the already rich tunability of 2D materials. One of the knobs, the twist angle between different layers, plays a crucial role in the ultimate electronic properties of a vdW heterostructure and does not have a direct analog in other systems such as MBE-grown semiconductor heterostructures. For small twist angles, the moiré pattern produced by the lattice misorientation creates a long-range modulation. So far, the study of the effect of twist angles in vdW heterostructures has been mostly concentrated in graphene/hexagonal boron nitride (h-BN) twisted structures, which exhibit relatively weak interlayer interaction due to the presence of a large bandgap in h-BN. Here we show that when two graphene sheets are twisted by an angle close to the theoretically predicted 'magic angle', the resulting flat band structure near charge neutrality gives rise to a strongly-correlated electronic system. These flat bands exhibit half-filling insulating phases at zero magnetic field, which we show to be a Mott-like insulator arising from electrons localized in the moiré superlattice. These unique properties of magic-angle twisted bilayer graphene (TwBLG) open up a new playground for exotic many-body quantum phases in a 2D platform made of pure carbon and without magnetic field. The easy accessibility of the flat bands, the electrical tunability, and the bandwidth tunability though twist angle may pave the way towards more exotic correlated systems, such as unconventional superconductors or quantum spin liquids. "

Query 5, with over 100 results within radius

Query 5 Title: 'Search for Higgs boson decays into a Z boson and a light hadronically decaying resonance using 13 TeV pp collision data from the ATLAS detector'

Query 5 Abstract: 'A search for Higgs boson decays into a Z boson and a light resonance in two-lepton plus jet events is performed, using a pp collision dataset with an integrated luminosity of 139 fb^{-1} collected at $\sqrt{s} = 13$ TeV by the ATLAS experiment at the CERN LHC. The resonance considered is a light boson with a mass below 4 GeV from a possible extended scalar sector, or a charmonium state. Multivariate discriminants are used for the event selection and for evaluating the mass of the light resonance. No excess of events above the expected background is found. Observed (expected) 95$\%$ confidence-level upper limits are set on the Higgs boson production cross section times branching fraction to a Z boson and the signal resonance, with values in the range 17 pb to 340 pb (16^{+6}_{-5} pb to 320^{+130}_{-90} pb) for the different light spin-0 boson mass and branching fraction hypotheses, and with values of 110 pb and 100 pb (100^{+40}_{-30} pb and 100^{+40}_{-30} pb) for the η_c and J/ψ hypotheses, respectively. '

Result Within Radius Title: 'New results on Higgs boson properties'

Result Within Radius Abstract: 'We present the latest ATLAS and CMS measurements of

several properties of the Higgs boson, such as signal-strength modifiers for the main production modes, fiducial and differential cross sections, and the Higgs mass. We have analyzed the 13 TeV proton-proton LHC collision data recorded in 2016, corresponding to integrated luminosities up to $36.1\ \mathrm{fb}^{-1}$. Results for the $\mathrm{H} \to \mathrm{ZZ} \to 4\ell$ ($\ell = \mathrm{e}\mu$), $\mathrm{H} \to \gamma\gamma$, and $\mathrm{H} \to \tau\tau$ decay channels are presented. In addition, searches for new phenomena in the $\mathrm{H} \to \gamma\gamma + E_{\mathrm{T}}^{\mathrm{miss}}$ and $\mathrm{H} \to \mathrm{b\bar{b}} + E_{\mathrm{T}}^{\mathrm{miss}}$ decay channels are presented. '

Result Within Radius Title: 'Search for the Standard Model Higgs boson decaying into $b\bar{b}$ produced in association with top quarks decaying hadronically in pp collisions at $\sqrt{s}$=8 TeV with the ATLAS detector'

Result Within Radius Abstract: 'A search for Higgs boson production in association with a pair of top quarks ($t\bar{t}H$) is performed, where the Higgs boson decays to $b\bar{b}$, and both top quarks decay hadronically. The data used correspond to an integrated luminosity of 20.3 fb^{-1} of pp collisions at $\sqrt{s} = 8$ TeV collected with the ATLAS detector at the Large Hadron Collider. The search selects events with at least six energetic jets and uses a boosted decision tree algorithm to discriminate between signal and Standard Model background. The dominant multijet background is estimated using a dedicated data-driven technique. For a Higgs boson mass of 125 GeV, an upper limit of 6.4 (5.4) times the Standard Model cross section is observed (expected) at 95% confidence level. The best-fit value for the signal strength is $\mu = 1.6 \pm 2.6$ times the Standard Model expectation for $m_H = 125$ GeV. Combining all $t\bar{t}H$ searches carried out by ATLAS at $\sqrt{s} = 8$ and 7 TeV, an observed (expected) upper limit of 3.1 (1.4) times the Standard Model expectation is obtained at 95% confidence level, with a signal strength $\mu = 1.7 \pm 0.8$. '

Result Outside Radius Title: 'Higgs Searches in ATLAS'

Result Outside Radius Abstract: 'This talk covers the results of a search for the Standard Model Higgs boson in proton-proton collisions with the ATLAS detector at the LHC. The datasets used correspond to integrated luminosities of approximately 4.8 fb-1 collected at sqrt(s) = 7 TeV in 2011 and 5.8 fb-1 at $\sqrt{s} = 8TeV$ in 2012. Individual searches in the channels $H- > ZZ(*)- > 4l, H- > \gamma\gamma$ and $H- > WW(*)- > e\nu\mu\nu$ in the 8 TeV data are combined with previously published results of searches for H-¿ZZ(*), WW(*), bb and $\tau + \tau-$ in the 7 TeV data and results from improved analyses of the $H- > ZZ(*)- > 4l$ and $H- > \gamma\gamma$ channels in the 7 TeV data. Clear evidence for the production of a neutral boson with a measured mass of 126.0 +/- 0.4 (stat) +/- 0.4 (sys) GeV is presented. This observation, which has a significance of 5.9 standard deviations, corresponding to a background fluctuation probability of 1.7 x 10-9, is compatible with the production and decay of the Standard Model Higgs boson. First measurements of the couplings of this particle are presented and are compatible with a SM Higgs boson hypothesis. '

Result Outside Radius Title: "Search for a heavy vector boson decaying to two gluons in $p\bar{p}$ collisions at $\sqrt{s} = 1.96$ TeV'

Result Outside Radius Abstract: "We present a search for a new heavy vector boson Z' that decays to gluons. Decays to on-shell gluons are suppressed, leading to a dominant decay mode of $Z' \to g^*g$. We study the case where the off-shell gluon g^* converts to a pair of top quarks, leading to a final state of $t\bar{t}g$. In a sample of events with exactly one charged lepton, large missing transverse momentum and at least five jets, corresponding to an integrated luminosity of 8.7 fb^{-1} collected by the CDF II detector, we find the data to be consistent with the standard model. We set upper limits on the production cross section times branching ratio of this chromophilic Z'

at 95% confidence level from 300 fb to 40 fb for Z' masses ranging from 400 GeV$/c^2$ to 1000 GeV$/c^2$, respectively. ”

Query 6, with over 100 results within radius

Query 6 Title: ‘Leptogenesis in Type Ib seesaw models’

Query 6 Abstract: ‘We study leptogenesis in three different realisations of the type Ib seesaw mechanism, where the effective masses of the neutrinos are obtained by the spontaneous symmetry breaking of two different Higgs doublets. In the minimal type Ib seesaw model, where two right-handed neutrinos form a Dirac mass, we show that it is impossible to produce the correct baryon asymmetry, even including a pseudo-Dirac mass splitting. In an extended type Ib seesaw model, with a third very heavy Majorana right-handed neutrino, together with the low scale Dirac pair of right-handed (RH) neutrinos and an extra singlet Higgs boson, we find that the imbalance of matter and antimatter can be explained by resonant leptogenesis. In the resulting low scale effective type Ib seesaw mechanism, we derive the allowed range of the seesaw couplings consistent with resonant leptogenesis. Dark matter may also be included via the right-handed neutrino portal. The Dirac RH neutrino masses may lie in the 1-100 GeV mass range, accessible to the future experiments SHiP and FCC-ee, allowing the type Ib seesaw mechanism with leptogenesis and dark matter to be tested. ’

Result Within Radius Title: ‘Radiative type-I seesaw model with dark matter via U(1)_B-L gauge symmetry breaking at future linear colliders’

Result Within Radius Abstract: ‘We discuss phenomenology of the radiative seesaw model in which spontaneous breaking of the U(1)$_{B-L}$ gauge symmetry at the TeV scale gives the common origin for masses of neutrinos and dark matter (Kanemura et al., 2012). In this model, the stability of dark matter is realized by the global U(1)$_{DM}$ symmetry which arises by the B$-$L charge assignment. Right-handed neutrinos obtain TeV scale Majorana masses at the tree level. Dirac masses of neutrinos are generated via one-loop diagrams. Consequently, tiny neutrino masses are generated at the two-loop level by the seesaw mechanism. This model gives characteristic predictions, such as light decayable right-handed neutrinos, Dirac fermion dark matter and an extra heavy vector boson. These new particles would be accessible at collider experiments because their masses are at the TeV scale. The U(1)$_{B-L}$ vector boson may be found at the LHC, while the other new particles could only be tested at future linear colliders. We find that the dark matter can be observed at a linear collider with $\sqrt{s}$=500 GeV and that light right-handed neutrinos can also be probed with $\sqrt{s}$=1 TeV. ’

Result Within Radius Title: ‘Neutrino Dark Energy in Grand Unified Theories’

Result Within Radius Abstract: ‘ We studied a left-right symmetric model that can accommodate the neutrino dark energy (
d) proposal. Type III seesaw mechanism is implemented to give masses to the neutrinos. After explaining the model, we study the consistency of the model by minimizing the scalar potential and obtaining the conditions for the required vacuum expectation values of the different scalar fields. This model is then embedded in an SO(10) grand unified theory and the allowed symmetry breaking scales are determined by the condition of the gauge coupling unification. Although $SU(2)_R$ breaking is required to be high, its Abelian subgroup $U(1)_R$ is broken in the TeV range, which can then give the required neutrino masses and predicts new gauge bosons that could be detected at LHC. The neutrino masses are studied in details in this model, which shows that at

least 3 singlet fermions are required. '

Result Outside Radius Title: 'Probing Leptogenesis at Future Colliders'

Result Outside Radius Abstract: 'We investigate the question whether leptogenesis, as a mechanism for explaining the baryon asymmetry of the universe, can be tested at future colliders. Focusing on the minimal scenario of two right-handed neutrinos, we identify the allowed parameter space for successful leptogenesis in the heavy neutrino mass range between 5 and 50 GeV. Our calculation includes the lepton flavour violating contribution from heavy neutrino oscillations as well as the lepton number violating contribution from Higgs decays to the baryon asymmetry of the universe. We confront this parameter space region with the discovery potential for heavy neutrinos at future lepton colliders, which can be very sensitive in this mass range via displaced vertex searches. Beyond the discovery of heavy neutrinos, we study the precision at which the flavour-dependent active-sterile mixing angles can be measured. The measurement of these mixing angles at future colliders can test whether a minimal type I seesaw mechanism is the origin of the light neutrino masses, and it can be a first step towards probing leptogenesis as the mechanism of baryogenesis. We discuss how a stronger test could be achieved with an additional measurement of the heavy neutrino mass difference. '

Result Outside Radius Title: 'Neutrinoless Double Beta Decay: Low Left-Right Symmetry Scale?'

Result Outside Radius Abstract: 'Experiments in progress may confirm a nonzero neutrinoless double beta decay rate in conflict with the cosmological upper limit on neutrino masses and thus require new physics beyond the Standard Model. A natural candidate is the Left-Right symmetric theory, which led originally to neutrino mass and the seesaw mechanism. In the absence of cancelations of large Dirac Yukawa couplings, we show how such a scenario would require a low scale of Left-Right symmetry breaking roughly below 10 TeV, tantalizingly close to the LHC reach. '

6.6 Additional Data on Early Stopping Metrics

We provide data on early stopping metrics for four additional datasets in Figures 6.9, 6.10, 6.11, and 6.12.

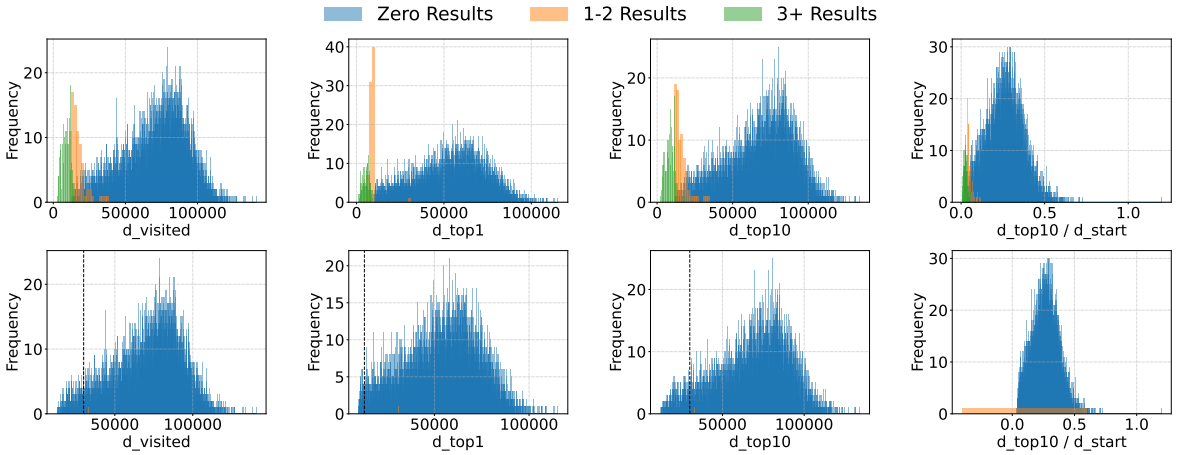


Figure 6.9: Early stopping metrics for BIGANN-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius.

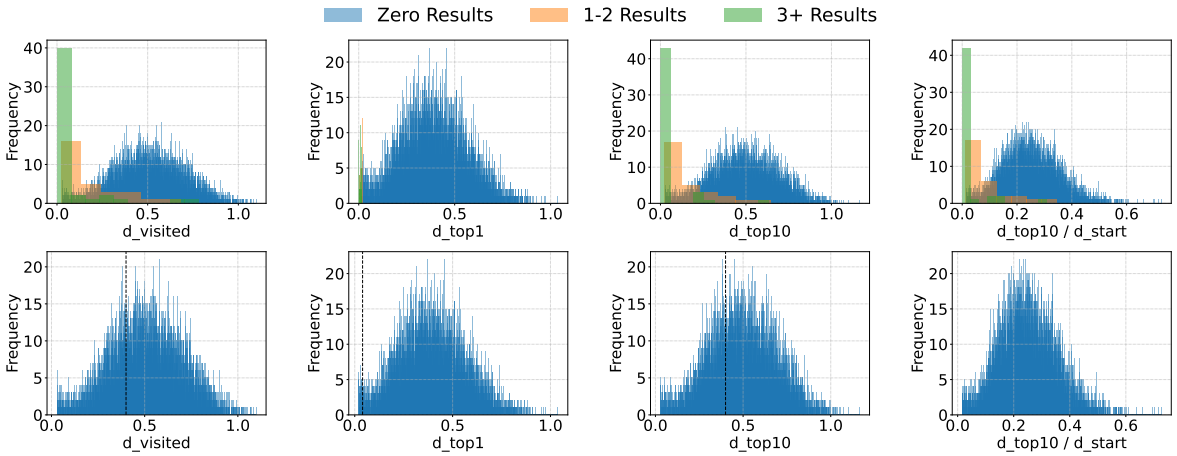


Figure 6.10: Early stopping metrics for DEEP-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius.

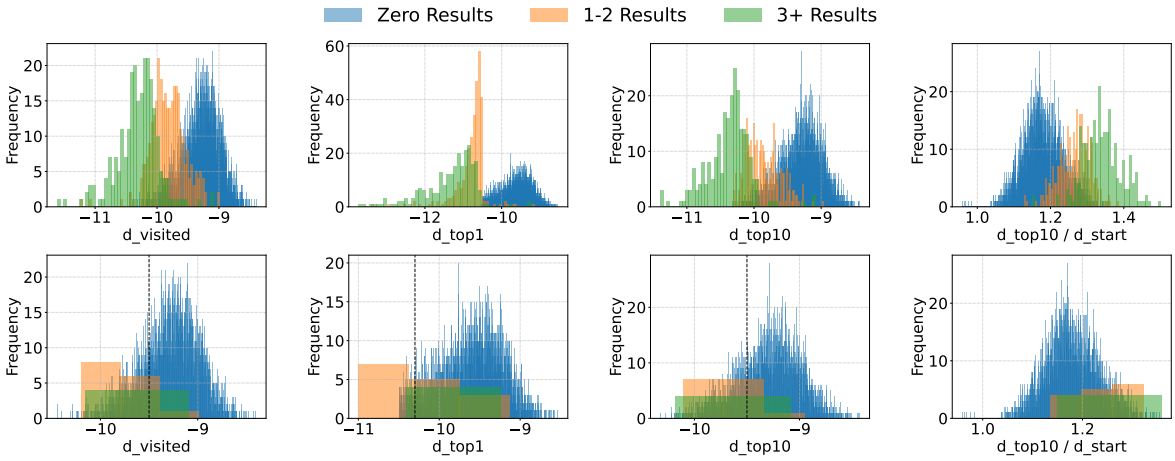


Figure 6.11: Early stopping metrics for Wikipedia-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius.

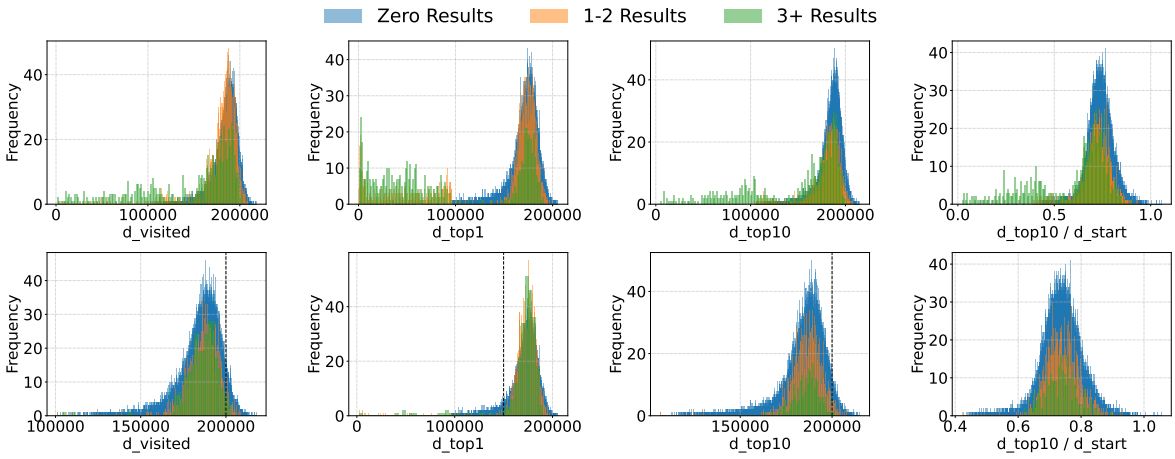


Figure 6.12: Early stopping metrics for SSNPP-1M, taken at step 20 of a beam search with beam 100. The top row shows all results, while the bottom row shows only results for beam searches that have not yet found a candidate within the radius.

Part III

Conclusion

Chapter 7

Conclusion

This thesis argued that "*With new techniques in parallelism and concurrency, data structures for nearest neighbor search can be made work-efficient, dynamic in both fully concurrent and parallel batch-dynamic settings, and effective at related geometric tasks such as range searching*." Taking each point in turn, we address how the thesis supported them.

- **Work-efficient**: In Chapter 3, we introduce the zd-tree, a data structure for nearest neighbor search that has provable guarantees on the worst-case span of its build. Furthermore, we showed that in practice the zd-tree maintains near-linear speedup on both build and query up to 144 threads. In Chapter 5, we showed from a practical perspective that four graph-based ANNS algorithms—DiskANN, HNSW, HCNNG, and PyNNDescent—could be built in parallel without the use of locks. We demonstrate their near-linear parallel speedup on up to 192 threads. Furthermore, we demonstrate that our algorithms achieve superior speedup over their original, lock-based formulations.
- **Dynamic in both fully concurrent and batch-dynamic settings**: In Chapter 3, on the zd-tree, we show that the zd-tree can support both batch insertions and deletions. Both insertions and deletions can be provably shown to have low span, and practical experiments show that they are fast on many threads, and that increasing the batch size drastically decreases the cost of updates. Furthermore, in Chapter 4, we show that the CLEANN-tree, a generalization of the zd-tree, supports provably correct, linearizable, and lock-free updates and queries. Experimental results show that the CLEANN-Tree achieves near-linear speedup up to 144 threads.
- **Effective at related geometric tasks such as range search**: In Chapter 4, we show that the CLEANN-tree supports efficient range queries in addition to k-nearest neighbor queries. In Chapter 6, we show that existing graph-based indices for high-dimensional ANNS can be efficiently adapted for range search. We introduce techniques that allow queries with no results to terminate early, as well as techniques for efficiently extending search for queries with thousands to tens of thousands of results. Our techniques yield significant improvement over a naive baseline, and as far as we know are the first dedicated algorithms for range search on graph-based ANNS indices.

7.0.1 Open Problems

In this section we propose some open problems based on the work in this thesis.

Open Problems in Low-Dimensional Search

Transactions for Lock-Free Batch Updates The CLEANN-tree supports concurrent point updates and insertions, and the zd-tree supports batch-dynamic updates of a single type (either insert or delete, and not concurrent with queries). Since batch-dynamic updates decrease the total work per point by amortizing some of the work of the sort and the modifications of bounding boxes, it may be desirable to take advantage of batching even in the concurrent setting. Thus work extending the CLEANN-tree to lock-free and linearizable batch updates would be a useful extension.

Data-Adaptive Lock-Free KD-Trees The zd-tree and the CLEANN-tree are not data adaptive in the sense that the splits of the bounding box containing the dataset are pre-determined. Certain adversarial datasets could force the depth of the tree to be linear in the number of data points (to the converse, the zd-tree is impervious to adversarial *orderings* of insertion). Some cases would therefore be better served by an adaptive splitting algorithm and the ability to perform rotations in the tree. To the best of our knowledge, there is no such adaptive data structure for low-dimensional nearest neighbor search that supports insertions, deletions, and k-nearest neighbor queries. Use of versioned pointers and lock-free locks to produce such a data structure would be a valuable contribution.

Open Problems in High-Dimensional Search

Data-Adaptive Stopping Criteria One point illuminated by both our work in early stopping criteria for range search and the Li et al. paper [121] is that some top-k searches and range searches can be terminated early, before the beam search would have otherwise terminated. Applying machine learning or other auto-tuning techniques to predict whether a search should continue has significant potential to improve both top-k and range queries.

Distributed Search at the Trillion Scale While ParlayANN showed that billion-scale nearest neighbor search is feasible on a large multicore machine, there is almost no academic work on trillion-scale nearest neighbor search (the only work we are aware of is a brief case study from Meta AI Research). Working at the trillion scale necessarily requires a *distributed* algorithm, and the distributed setting poses many interesting questions. One of the most basic questions is how to partition a graph across multiple machines. How should edges that would span two machines be handled? Should data be partitioned in a locality-aware manner, and would this enable queries to be routed to fewer machines? Should the fundamental graph construction change?

Filtered Search The problem of *filtered search*, which augments top-k search with a requirement that returned results must satisfy some logical predicate, is an important practical extension

of nearest neighbor search. It was the topic of the NeurIPS 2023 Practical Vector Search Challenge [158], and has been the subject of a handful of academic papers []. However, none of these works succeed at providing a general solution which works for arbitrary predicates and for all levels of filter selectivity. Given the popularity of filtered search, a robust solution to this problem would be of high impact.

Bibliography

[1] Kgraph: A library for approximate nearest neighbor search. Webpage, 2016. URL https://github.com/aaalgo/kgraph.

[2] N2. Webpage, 2021. URL https://github.com/kakao/n2.

[3] Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk. Webpage, 2022. URL https://github.com/spotify/annoy.

[4] Vald: A highly scalable distributed vector search engine. Webpage, 2022. URL https://github.com/vdaas/vald.

[5] Opensearch k-nn. Webpage, 2022. URL https://github.com/opensearch-project/k-NN.

[6] Spacial algorithms and data structures. Webpage, 2022. URL https://docs.scipy.org/doc/scipy/reference/spatial.htmla.

[7] vespa. Webpage, 2022. URL https://github.com/vespa-engine/vespa.

[8] Neurips'23 competition track: Big-ann, 2023. URL https://big-ann-benchmarks.com/neurips23.html.

[9] Microsoft bing search engine, 2023. URL https://www.bing.com/new.

[10] Chatgpt-retrieval-plugin/readme.md, Mar 2023. URL https://github.com/openai/chatgpt-retrieval-plugin/blob/main/README.md.

[11] Apache lucene, 2023. URL https://lucene.apache.org/.

[12] Vector database built for scalable similarity search, 2023. URL https://milvus.io/.

[13] Pinecone: Vector database for vector search, 2023. URL https://www.pinecone.io/.

[14] Vector database, 2023. URL https://qdrant.tech/.

[15] Apr 2023. URL https://graphics.stanford.edu/data/3Dscanrep/.

[16] Weaviate: The ai native vector database, 2023. URL https://weaviate.io/.

[17] Chroma, 2024. URL https://docs.trychroma.com/.

[18] pgvector: Embeddings and vector similarity, 2024. URL https://supabase.com/docs/guides/database/extensions/pgvector.

[19] Vector search, 2024. URL https://docs.rockset.com/documentation/

docs/vector-search.

[20] Built-in vector database, 2024. URL https://www.singlestore.com/built-in-vector-database/.

[21] S.J. Aarseth, M. Henon, and R. Wielen. Numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics*, 37(2), 1974.

[22] Archita Agarwal, Zhiyu Liu, Eli Rosenthal, and Vikram Saraph. Linearizable iterators for concurrent data structures. *CoRR*, abs/1705.08885, 2017. URL http://arxiv.org/abs/1705.08885.

[23] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Principles of Database Systems (PODS)*. ACM, 2016.

[24] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Cláudio T. Silva. Point set surfaces. In *12th IEEE Visualization Conference, IEEE Vis 2001, San Diego, CA, USA, October 24-26, 2001, Proceedings*. IEEE Computer Society, 2001.

[25] Pierre Alliez and Andreas Fabri. CGAL: the computational geometry algorithms library. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference, SIGGRAPH '16, Anaheim, CA, USA, July 24-28, 2016, Courses*. ACM, 2016.

[26] Laurent Amsaleg and Herve Jegou. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/, 2010.

[27] Evangelos Anagnostopoulos, Ioannis Z. Emiris, and Ioannis Psarros. Low-quality dimension reduction and high-dimensional approximate nearest neighbor. In *International Symposium on Computational Geometry (SoCG)*, volume 34 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[28] Evangelos Anagnostopoulos, Ioannis Z. Emiris, and Ioannis Psarros. Randomized embeddings with slack and high-dimensional approximate nearest neighbor. *ACM Trans. Algorithms*, 14(2), 2018.

[29] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 1225–1233, 2015.

[30] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 14–27, 2018.

[31] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2), Apr 2001.

[32] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280. ACM/SIAM, 1993.

[33] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of*

the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA. ACM/SIAM, 1994.

[34] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. PUFFINN: parameterless and universally fast finding of nearest neighbors. In *Annual European Symposium on Algorithms (ESA)*, volume 144, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[35] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87, 2020.

[36] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN benchmarks. `http://ann-benchmarks.com/`, 2021.

[37] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 299–308, 2013.

[38] Dmitry Baranchuk and Artem Babenko. Benchmarks for billion-scale similarity search. Webpage, 2021. URL `https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search`.

[39] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Computer Vision - ECCV 2018*, volume 11216 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2018.

[40] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. *ACM Transactions on Parallel Computing (TOPC)*, 7(3), June 2020.

[41] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022.

[42] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. The flock library. `https://github.com/cmuparlay/flock`, 2022.

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

[44] Alina Beygelzimer, Sham M. Kakade, and John Langford. Cover trees for nearest neighbor. In William W. Cohen and Andrew W. Moore, editors, *ACM International Conference on Machine Learning (ICML)*, volume 148 of *ACM International Conference Proceeding Series*. ACM, 2006.

[45] Marcel Birn, Manuel Holtgrewe, Peter Sanders, and Johannes Singler. Simple and fast nearest neighbor search. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*. SIAM, 2010.

[46] Guy E Blelloch and Magdalen Dobson. Parallel nearest neighbors in low dimensions with batch updates. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 195–208. SIAM, 2022.

[47] Guy E. Blelloch and Yuanhao Wei. Verlib: Concurrent versioned pointers. In *ACM Sym-*

posium on Principles and Practice of Parallel Programming (PPOPP), 2024.

[48] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478. ACM, 2016. doi: 10.1145/2935764.2935766. URL https://doi.org/10.1145/2935764.2935766.

[49] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020. URL https://cmuparlay.github.io/parlaylib/.

[50] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[51] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.

[52] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), 1999.

[53] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In *Similarity Search and Applications (SISAP)*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2013.

[54] Paul B Callahan and S Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1), 1995.

[55] Timothy Chan. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions, 2006. URL https://tmc.web.engr.illinois.edu/pub_ann.html.

[56] Timothy M. Chan. Well-separated pair decomposition in linear time? *Inf. Process. Lett.*, 107(5), 2008.

[57] Harrison Chase. Vector db text generation, Mar 2023. URL https://python.langchain.com/en/latest/modules/chains/index_examples/vector_db_text_generation.html.

[58] Bapi Chatterjee. Concurrentkdtree. https://github.com/bapi/ConcurrentKDTree_o, 2016.

[59] Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *IEEE International Conference on Distributed Computing and Networking (ICDCN)*, pages 9:1–9:10, 2017.

[60] Bapi Chatterjee, Ivan Walulya, and Philippas Tsigas. Concurrent linearizable nearest neighbour search in lockfree-kd-tree. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 11:1–11:10. ACM, 2018. doi: 10.1145/3154273.3154307. URL https://doi.org/10.1145/3154273.3154307.

[61] Bernard Chazelle, Ding Liu, and Avner Magen. Approximate range searching in higher

dimension. *Comput. Geom.*, 39(1):24–29, 2008. doi: 10.1016/J.COMGEO.2007.05.008. URL https://doi.org/10.1016/j.comgeo.2007.05.008.

[62] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. *SPTAG: A library for fast approximate nearest neighbor search*, 2018. URL https://github.com/Microsoft/SPTAG.

[63] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighborhood search. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 5199–5212, 2021.

[64] Qi Chen, Xiubo Geng, Corby Rosset, Carolyn Buractaon, Jingwen Lu, Tao Shen, Kun Zhou, Chenyan Xiong, Yeyun Gong, Paul Bennett, Nick Craswell, Xing Xie, Fan Yang, Bryan Tower, Nikhil Rao, Anlei Dong, Wenqi Jiang, Zheng Liu, Mingqin Li, Chuanjie Liu, Zengzhong Li, Rangan Majumder, Jennifer Neville, Andy Oakley, Knut Magne Risvik, Harsha Vardhan Simhadri, Manik Varma, Yujing Wang, Linjun Yang, Mao Yang, and Ce Zhang. Ms marco web search: A large-scale information-rich web dataset with millions of real click labels. In *Companion Proceedings of the ACM on Web Conference 2024*, WWW '24, 2024.

[65] Yewang Chen, Lida Zhou, Yi Tang, Jai Puneet Singh, Nizar Bouguila, Cheng Wang, Huazhen Wang, and Ji-Xiang Du. Fast neighbor search by using revised *k*-d tree. *Inf. Sci.*, 472, 2019.

[66] Ulrich Clarenz, Martin Rumpf, and Alexandru Telea. Finite elements on point based surfaces. In *Symposium on Point Based Graphics (PBG)*. Eurographics Association, 2004.

[67] Kenneth L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the Tenth Annual Symposium on Computational Geometry, Stony Brook, New York, USA, June 6-8, 1994*. ACM, 1994.

[68] Kenneth L. Clarkson. Nearest neighbor queries in metric spaces. *Discret. Comput. Geom.*, 22(1), 1999.

[69] M. Connor and P. Kumar. Parallel construction of k-nearest neighbor graphs for point clouds. In Hans-Christian Hege, David H. Laidlaw, Renato Pajarola, and Oliver G. Staadt, editors, *Eurographics / IEEE VGTC Symposium on Volume Graphics*, pages 25–31. Eurographics Association, 2008. doi: 10.2312/VG/VG-PBG08/025-031. URL https://doi.org/10.2312/VG/VG-PBG08/025-031.

[70] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Trans. Vis. Comput. Graph.*, 16(4), 2010.

[71] SpaceV Contributors. Spacev1b: A billion-scale vector dataset for text descriptors. Webpage, 2021. URL https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B.

[72] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[73] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.

[74] Wei Dong, Moses Charikar, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 577–586. ACM, 2011.

[75] Matthijs Douze, Hervé Jégou, and Florent Perronnin. Polysemous codes. In *Computer Vision - ECCV 2016*, volume 9906 of *Lecture Notes in Computer Science*, pages 785–801. Springer, 2016.

[76] Matthijs Douze, Giorgos Tolias, Ed Pizzi, Zoë Papakipos, Lowik Chanussot, Filip Radenovic, Tomás Jenícek, Maxim Maximov, Laura Leal-Taixé, Ismail Elezi, Ondrej Chum, and Cristian Canton-Ferrer. The 2021 image similarity dataset and challenge. *CoRR*, abs/2106.09672, 2021. URL `https://arxiv.org/abs/2106.09672`.

[77] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *CoRR*, abs/2401.08281, 2024. doi: 10.48550/ARXIV.2401.08281. URL `https://doi.org/10.48550/arXiv.2401.08281`.

[78] Javier Duarte. Ucsd phys 141/241: Computational physics i. `https://jduarte.physics.ucsd.edu/phys141/README.html`, 2022.

[79] Jan Elseberg, S. Magnenat, Roland Siegwart, and Andreas Nuchter. Comparison on nearest-neigbour-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics (JOSER)*, 3, 01 2012.

[80] Jeff Erickson. Nice point sets can have nasty delaunay triangulations. *Discret. Comput. Geom.*, 30(1), 2003.

[81] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1), 1985.

[82] Shachar Fleishman, Daniel Cohen-Or, and Cláudio T. Silva. Robust moving least-squares fitting with sharp features. *ACM Trans. Graph.*, 24(3), 2005.

[83] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[84] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, 2019. doi: 10.14778/3303753.3303754. URL `http://www.vldb.org/pvldb/vol12/p461-fu.pdf`.

[85] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(8):4139–4150, 2022.

[86] Gago, Silvia, Schlatter, and Dirk. Bounded expansion in web graphs. *Commentationes Mathematicae Universitatis Carolinae*, 50(2), 2009.

[87] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srini-

vasan, Amit Singh, and Harsha Vardhan Simhadri. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In Ying Ding, Jie Tang, Juan F. Sequeda, Lora Aroyo, Carlos Castillo, and Geert-Jan Houben, editors, *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*, pages 3406–3416. ACM, 2023.

[88] Ryan Greene, Ted Sanders, Lilian Weng, and Arvind Neelakantan. New and improved embedding model. `https://openai.com/index/new-and-improved-embedding-model/`, 2022.

[89] Yan Gu, Julian Shun, Yihan Sun, and Guy E Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[90] Yan Gu, Zachary Napier, and Yihan Sun. Analysis of work-stealing and parallel cache complexity. In *ACM-SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 46–60. SIAM, 2022.

[91] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. Parallel cover trees and their applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

[92] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. Parallel longest increasing subsequence and van emde boas trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2023.

[93] Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with OPTIK. In Rafael Asenjo and Tim Harris, editors, *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2016. URL `https://doi.org/10.1145/2851141.2851146`.

[94] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. BLISS: A billion scale index using iterative re-partitioning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 486–495. ACM, 2022.

[95] Jonathan Halcrow, Alexandru Mosoi, Sam Ruth, and Bryan Perozzi. Grale: Designing networks for graph learning. In *SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, page 2523–2532, 2020. ISBN 9781450379984. doi: 10.1145/3394486.3403302.

[96] Ben Harwood and Tom Drummond. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5713–5722, 2016.

[97] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2006.

[98] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[99] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*

(TOPLAS), 12(3), 1990.

[100] Linjia Hu, Saeid Nooshabadi, and Majid Ahmadi. Massively parallel kd-tree construction and nearest neighbor search algorithms. In *2015 IEEE International Symposium on Circuits and Systems, ISCAS 2015, Lisbon, Portugal, May 24-27, 2015*. IEEE, 2015.

[101] Yihao Huang, Shangdi Yu, and Julian Shun. Faster parallel exact density peaks clustering. *CoRR*, abs/2305.11335, 2023. doi: 10.48550/arXiv.2305.11335. URL https://doi.org/10.48550/arXiv.2305.11335.

[102] Jeffrey Ichnowski and Ron Alterovitz. Concurrent nearest-neighbor searching for parallel sampling-based motion planning in so(3), se(3), and euclidean spaces. In *Algorithmic Foundations of Robotics XIII, Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics, WAFR 2018, Mérida, Mexico, December 9-11, 2018*, volume 14 of *Springer Proceedings in Advanced Robotics*, pages 69–85. Springer, 2018. doi: 10.1007/978-3-030-44051-0_5. URL https://doi.org/10.1007/978-3-030-44051-0_5.

[103] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 604–613. ACM, 1998. doi: 10.1145/276698.276876. URL https://doi.org/10.1145/276698.276876.

[104] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org.

[105] Masajiro Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. In *Similarity Search and Applications (SISAP)*, volume 9939 of *Lecture Notes in Computer Science*, pages 20–33, 2016.

[106] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *CoRR*, abs/1810.07355, 2018. URL http://arxiv.org/abs/1810.07355.

[107] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. Ood-diskann: Efficient and scalable graph ANNS for out-of-distribution queries. *CoRR*, abs/2211.12850, 2022. doi: 10.48550/arXiv.2211.12850. URL https://doi.org/10.48550/arXiv.2211.12850.

[108] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1992. ISBN 0201548569.

[109] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 33(1), 2010.

[110] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics (ICASSP)*, pages 861–864. IEEE, 2011. URL https://doi.org/10.1109/ICASSP.2011.5946540.

[111] Herve Jegou, Matthijs Douze, Jeff Johnson, Lucas Hosseini, Chengqi Deng, and

Alexandr Guzhva. Faiss wiki. Webpage, 2023. URL https://github.com/facebookresearch/faiss/wiki.

[112] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547, 2021.

[113] Guy L. Steele Jr. and Jean-Baptiste Tristan. Adding approximate counters. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2016.

[114] Haim Kaplan and Robert Endre Tarjan. Purely functional representations of catenable sorted lists. In *ACM Symposium on Theory of Computing (STOC)*, 1996.

[115] David R. Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 2002.

[116] Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Principles of Database Systems (PODS)*. ACM, 2013.

[117] Alex Klibisz. Tour de elastiknn. Webpage, 2021. URL https://elastiknn.com/posts/tour-de-elastiknn-august-2021/.

[118] Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski. Jiffy: A lock-free skip list with batch updates and snapshots. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022.

[119] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3), 1980.

[120] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[121] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2539–2554. ACM, 2020. doi: 10.1145/3318464.3380600. URL https://doi.org/10.1145/3318464.3380600.

[122] Hao Li, Xiaojie Liu, Tao Li, and Rundong Gan. A novel density-based clustering algorithm using nearest neighbor graph. *Pattern Recognit.*, 102:107206, 2020. doi: 10.1016/J.PATCOG.2020.107206. URL https://doi.org/10.1016/j.patcog.2020.107206.

[123] Roberto Javier López-Sastre, Daniel Oñoro-Rubio, Pedro Gil-Jiménez, and Saturnino Maldonado-Bascón. Fast reciprocal nearest neighbors clustering. *Signal Process.*, 92 (1), 2012.

[124] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. Hvs: Hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *Proc. VLDB Endow.*, 15(2):246–258, 2022.

[125] Anonymous Maciej Kula, Matthew Ward. rpforest. Webpage, 2019. URL https://github.com/lyst/rpforest.

[126] Yury Malkov and Dmitry Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 2018.

[127] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In Michel Steuwer, I-Ting Angelina Lee, and Milind Chabbi, editors, *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, pages 270–285. ACM, 2024. doi: 10.1145/3627535.3638475. URL https://doi.org/10.1145/3627535.3638475.

[128] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. Parlayann. Webpage, 2025. URL https://github.com/cmuparlay/ParlayANN.

[129] Leland McInnes. Pynndescent for fast approximate nearest neighbors. Webpage, 2020. URL https://pynndescent.readthedocs.io/en/latest/.

[130] MetaAI. Using ai to detect covid-19 misinformation and exploitative content. Webpage, 2020. URL https://ai.facebook.com/blog/using-ai-to-detect-covid-19-misinformation-and-exploitative-content/.

[131] Scott A. Mitchell and David M. Day. Flexible approximate counting. In *ACM International Database Engineering and Applications Symposium (IDEAS)*, 2011.

[132] Niloy J. Mitra, An Thanh Nguyen, and Leonidas J. Guibas. Estimating surface normals in noisy point cloud data. *Int. J. Comput. Geom. Appl.*, 14(4-5), 2004.

[133] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21 (10), October 1978.

[134] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340), 2009.

[135] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognit.*, 96, 2019.

[136] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2014.

[137] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, page 368–384, 2022.

[138] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[139] Renato Pajarola. Stream-processing points. In *16th IEEE Visualization Conference*. IEEE Computer Society, 2005.

[140] Christos H Papadimitriou and Paris C Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems (TODS)*, 9(1):89–99, 1984.

[141] Mark Pauly, Richard Keiser, Leif Kobbelt, and Markus H. Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3), 2003.

[142] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. Speed-ann: Low-latency and high-accuracy nearest neighbor search via intra-query parallelism. *arXiv preprint arXiv:2201.13007*, 2022.

[143] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 313–328, 2023.

[144] Ninh Pham and Tao Liu. Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search. *CoRR*, abs/2206.01382, 2022. doi: 10.48550/arXiv.2206.01382.

[145] Alexander Ponomarenko, Yury Malkov, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor search small world approach. In *International Conference on Information and Communication Technologies & Applications*, volume 17, 2011.

[146] Parikshit Ram and Kaushik Sinha. Revisiting kd-tree for nearest neighbor search. In *SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2019.

[147] Amir Raoofy, Roman Karlstetter, Martin Schreiber, Carsten Trinitis, and Martin Schulz. Overcoming weak scaling challenges in tree-based nearest neighbor time series mining. In *International Conference on High Performance Computing*, pages 317–338. Springer, 2023.

[148] D. Reed. Naming and synchronization in a decentralized computer system. Technical Report LCS/TR-205, EECS Dept., MIT, September 1978.

[149] Nils Reimers. Datasets: Cohere/wikipedia-22-12-en-embeddings. `https://huggingface.co/datasets/Cohere/wikipedia-22-12-en-embeddings`, 2022.

[150] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[151] Gavin Salam and Matteo Cacciari. Jet clustering in particle physics, via a dynamic nearest neighbour graph implemented with cgal. 04 2006.

[152] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 815–823. IEEE Computer Society, 2015. doi: 10.1109/CVPR.2015.7298682. URL `https://doi.org/10.1109/CVPR.2015.7298682`.

[153] Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over

databases with local bounded expansion. In *International Conference on Database Theory (ICDT)*, volume 68 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[154] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

[155] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2012.

[156] Harsha Simhadri, George Williams, Martin Aumüller, Artem Babenko, Dmitry Baranchuk, Qi Chen, Matthijs Douze, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. BigANN benchmarks: Billion-scale approximate nearest neighbor search challenge. `https://big-ann-benchmarks.com/`, 2021.

[157] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In Douwe Kiela, Marco Ciccone, and Barbara Caputo, editors, *NeurIPS 2021 Competitions and Demonstrations Track, 6-14 December 2021, Online*, volume 176 of *Proceedings of Machine Learning Research*, pages 177–189. PMLR, 2021. URL `https://proceedings.mlr.press/v176/simhadri22a.html`.

[158] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Benjamin Landrum, Mazin Karjikar, Laxman Dhulipala, Meng Chen, Yue Chen, Rui Ma, Kai Zhang, Yuzheng Cai, Jiayang Shi, Yizhuo Chen, Weiguo Zheng, Zihao Wan, Jie Yin, and Ben Huang. Results of the bigann: Neurips'23 competition. *CoRR*, abs/2409.17424, 2024. doi: 10.48550/ARXIV.2409.17424. URL `https://doi.org/10.48550/arXiv.2409.17424`.

[159] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ANN index for streaming similarity search. *CoRR*, abs/2105.09613, 2021. URL `https://arxiv.org/abs/2105.09613`.

[160] Colette Stallbaumer. Introducing microsoft 365 copilot, Mar 2023. URL `https://www.microsoft.com/en-us/microsoft-365/blog/2023/03/16/introducing-microsoft-365-copilot-a-whole-new-way-to-work/`.

[161] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. Rand-nsg: Fast accurate billion-point nearest neighbor search on a single node. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information*

Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pages 13748–13758, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Abstract.html.

[162] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, 2013.

[163] Gergely Szilvasy, Pierre-Emmanuel Mazaré, and Matthijs Douze. Vector search with small radiuses. *CoRR*, abs/2403.10746, 2024. doi: 10.48550/ARXIV.2403.10746. URL https://doi.org/10.48550/arXiv.2403.10746.

[164] Hans Tangelder and Andreas Fabri. dD spatial searching. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.2 edition, 2020. URL https://doc.cgal.org/5.2/Manual/packages.html#PkgSpatialSearchingD.

[165] tawalke. Pr: Sk vectordb connector work - merging forked branch; pr from fork to sk branch by tawalke · pull request 83 · microsoft/semantic-kernel, Mar 2023. URL https://github.com/microsoft/semantic-kernel/pull/83.

[166] Nitish Upreti, James Codella, and Harsha Vardhan Simhadri. Azure cosmos db vector search with diskann part 1: Full space search. Webpage, 2024. URL https://devblogs.microsoft.com/cosmosdb/azure-cosmos-db-vector-search-with-diskann-part-1-full-space-search/.

[167] Pravin M. Vaidya. An optimal algorithm for the all-nearest-neighbors problem. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1986.

[168] Jie Wang, Jian Lu, Zheng Fang, Tingjian Ge, and Cindy X. Chen. Pl-tree: An efficient indexing method for high-dimensional data. In Mario A. Nascimento, Timos K. Sellis, Reynold Cheng, Jörg Sander, Yu Zheng, Hans-Peter Kriegel, Matthias Renz, and Christian Sengstock, editors, *Advances in Spatial and Temporal Databases - 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings*, volume 8098 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2013. doi: 10.1007/978-3-642-40235-7_11. URL https://doi.org/10.1007/978-3-642-40235-7_11.

[169] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, 2021. doi: 10.14778/3476249.3476255. URL http://www.vldb.org/pvldb/vol14/p1964-wang.pdf.

[170] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proc. ACM Manag. Data*, 2(1):V2mod014:1–V2mod014:27, 2024. doi: 10.1145/3639269. URL https://doi.org/10.1145/3639269.

[171] Yiqiu Wang, Rahul Yesantharao, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. Pargeo: A library for parallel computational geometry. In *European Symposium on*

Algorithms (ESA), volume 244 of *LIPIcs*, pages 88:1–88:19.

[172] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1982–1995. ACM, 2021. doi: 10.1145/3448016.3457296. URL https://doi.org/10.1145/3448016.3457296.

[173] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.*, 13(12):3152–3165, 2020. doi: 10.14778/3415478.3415541. URL http://www.vldb.org/pvldb/vol13/p3152-wei.pdf.

[174] Yuanhao Wei. General Techniques for Efficient Concurrent Data Structures. 8 2023. doi: 10.1184/R1/23739501.v1. URL https://kilthub.cmu.edu/articles/thesis/General_Techniques_for_Efficient_Concurrent_Data_Structures/23739501.

[175] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2021.

[176] Yuanhao Wei, Naama Ben-David, and Guy E. Blelloch. Lock-free locks revisited. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022.

[177] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. P. A. Lensch. Efficient large-scale approximate nearest neighbor search on the gpu. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[178] Yan Xia, Kaiming He, Fang Wen, and Jian Sun. Joint inverted indexing. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3416–3423, 2013.

[179] Xiaoliang Xu, Mengzhao Wang, Yuxiang Wang, and Dingcheng Ma. Two-stage routing with optimized guided search and greedy algorithm on proximity graph. *Knowl. Based Syst.*, 229:107305, 2021. doi: 10.1016/J.KNOSYS.2021.107305. URL https://doi.org/10.1016/j.knosys.2021.107305.

[180] Rahul Yesantharao, Yiqiu Wang, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic k-d trees. *arXiv preprint arXiv:2112.06188*, 2021.

[181] Hongfei Zhang, Xia Song, Chenyan Xiong, Corby Rosset, Paul N. Bennett, Nick Craswell, and Saurabh Tiwary. Generic intent representation in web search. In Benjamin Piwowarski, Max Chevalier, Éric Gaussier, Yoelle Maarek, Jian-Yun Nie, and Falk Scholer, editors, *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2019, Paris, France, July 21-25, 2019*, pages 65–74. ACM, 2019. doi: 10.1145/3331184.3331198. URL https://doi.org/10.1145/3331184.3331198.

[182] Jiaru Zhang, Ruhui Ma, Tao Song, Yang Hua, Zhengui Xue, Chenyang Guan, and Haibing Guan. Hierarchical satellite system graph for approximate nearest neighbor search on big

data. *ACM/IMS Trans. Data Sci.*, 2(4), 2022.

[183] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *OSDI*, Boston, MA, 2023. ISBN 978-1-939133-34-2.

[184] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5), 2008.