ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms

Magdalen Dobson Manohar
Carnegie Mellon University
mrdobson@cs.cmu.edu

Zheqi Shen
UC Riverside
zshen055@ucr.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Laxman Dhulipala
University of Maryland
laxman@umd.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Harsha Vardhan Simhadri
Microsoft Research
harshasi@microsoft.com

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Abstract

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 tradeoffs. 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.

In this paper, 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.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; • **Information systems** → **Retrieval tasks and goals**.

Keywords: nearest neighbor search, vector search, parallel algorithms

ACM Reference Format:
Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *The 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24), March 2–6, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627535.3638475

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03.
https://doi.org/10.1145/3627535.3638475

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 [21]. 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 [71], as well as large language models (LLMs) used in ChatGPT [9] and other applications combining LLMs and vector search [10, 28, 67, 70]. 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 [38, 55, 56, 58, 68].

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. 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 [16] 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 [8, 11, 12].

On the benchmarking side, existing results [16, 71] 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 paper 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 [68], HNSW [55], HCNNG [58] and PyNNDescent [56]. 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. 4.2), and certain data structures overflowing the L3 cache (in HCNNG, see Sec. 4.3). Our implementations, ParlayDiskANN, ParlayHNSW, ParlayHCNNG and ParlayPyNN, 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 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 48 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 [13, 48]) on diverse real-world datasets with a billion points, including one dataset for out-of-distribution (OOD) queries (see more details in Sec. 5.1). 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. 5.1 (six algorithms each with three datasets) took more than 90 hours of computation time on a machine with 64 cores. 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. We plan to release our code. Due to page limits, we provide the full paper with appendix in the supplemental material.

1. A variety of general and specific techniques to parallelize existing graph-based ANNS algorithms to scale to billions of points (Sec. 3).
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. A list of interesting findings about parallel ANNS algorithms on large scale datasets (Sec. 5).

2 Preliminaries

Parallel Model. We use the fork-join model of parallelism [24, 33]. We assume a set of threads that access a shared memory. A process can fork two child software threads to work in parallel. When both children complete, the parent process continues. A parallel for-loop over n items can be simulated by recursively forking $\log n$ levels. Computations in the model can be efficiently executed using a randomized work-stealing scheduler [14, 26].

We say a parallel computation is ***deterministic*** if it gives the same output across multiple runs, i.e., the output is not affected by the runtime scheduler. For randomized algorithms, we assume the randomness is supplied as part of the input (e.g., as a random seed).

Parallel Semisort. Many of our algorithms use a parallel semisort [41] as a subroutine. Given a sequence A of entries, each associated with a *key*, a semisort reorders A such that all entries with the same key are consecutive. Note that the entries or keys do not need to be fully sorted.

Approximate Nearest Neighbor Search (ANNS). In this work, we study a set $\mathcal{P} \subseteq \mathbb{R}^d$ of n points (vectors) in d dimensions. We denote the ***distance*** between two points $p, q \in \mathbb{R}^d$ as $\|p, q\|$. Smaller distance indicates greater similarity. Commonly-used distance functions include Euclidean distance (L_2 norm), and cosine distance ($1 - \cos(\theta)$).

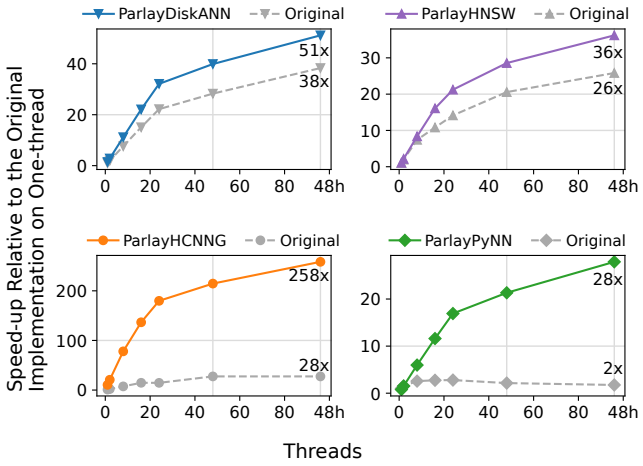


Figure 1. Scalability of original and our new implementations of four ANNS algorithms on various number of threads. Within each subfigure, all numbers are *speedup numbers relative to the original implementation on one thread.* Higher is better. Results were tested on a machine with 48 cores using dataset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

Definition 2.1. (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 most commonly-used measure of accuracy for ANNS, frequently referred to as *recall*.

Definition 2.2. ($k@k'$ recall) Let $\mathcal{P}$ be a set of points in d-dimensions and q a query point. Let $\mathcal{K}$ be the true k-nearest neighbors of q in $\mathcal{P}$. Let $\mathcal{K}' \subset \mathcal{P}$ be an output of an ANNS algorithm of size k'. Then the $k@k'$ recall of q is $\frac{|\mathcal{K} \cap \mathcal{K}'|}{|\mathcal{K}|}$.

The most common choice of recall is 10@10 recall. Throughout the paper, we use the term "recall" to refer to 10@10 recall of an entire query set, i.e., the average recall over all points in the query set.

3 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

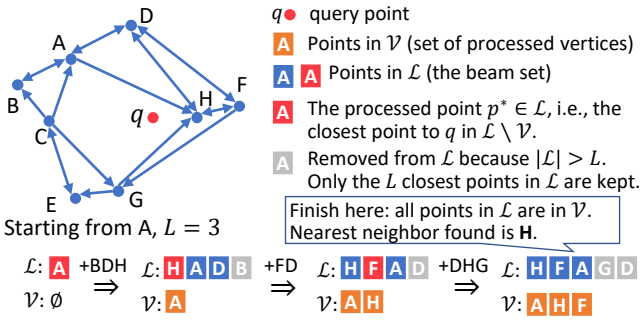


Figure 2. An example of ANNS graph and a greedy search. The blue arrows represent directed edges in the proximity graph, which is a mix of long and short edges. Below is an example of NNS query on point q (red point). The algorithm starts with adding the starting point A as the only point in the beam $\mathcal{L}$, 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.

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. 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. 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.

3.1 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

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 $\mathcal{V}, \mathcal{K}$

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.

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

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 [38, 55, 68] 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. 4.

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.

To address this, we use *prefix-doubling* [25, 34, 39, 40, 65]. 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*** (see Sec. 2) 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.

3.2 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 paper, 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) [58], 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. 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. 4.3 and 4.4, we further discuss our new ideas to address the scalability issue in HCNNG and PyNNDescent.

Parallelizing Clustering-Based Algorithms. To paral-

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 ← 1
while start ≤ |P| do                                   // Prefix-doubling
    end ← min(start × 2, start + θ, |P|)   // θ: batch size upper bound
    BatchInsert(P[start..end])
    start ← end + 1
Function BatchInsert(P')   // Insert a batch P' to the current index
    parallel for p ∈ P' do
        V, K ← greedySearch(p, s, L, 1)
        N_out(p) ← prune(p, V, R)
    B ← ⋃_{p∈P'} N_out(p)                  // All (existing) affected points
    parallel for b ∈ B do
        // N: all points in P' that added b as their neighbors
        N ← {p | p ∈ P' ∧ b ∈ N_out(p)}
        N_out(b) ← N_out(b) ∪ N
        if |N_out(b)| > R then N_out(b) ←
          prune(b, N_out(b), R)
```

lelize 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 [22, 46] to assign points to different branches in parallel. This 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. 4.3.

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 (see Sec. 2), such that the edges incident the same point are consecutive. The graph can be built accordingly.

4 ParlayANN Algorithms

In this section, we further describe four graph-based ANNS algorithms that benefit from our techniques proposed in Sec. 3. In addition to the general techniques, we also employ specific optimizations for each individual algorithm to improve their scalability, which will be introduced below.

4.1 DiskANN

DiskANN [68] 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) [38], 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.

4.2 HNSW

The hierarchical navigable small world (HNSW) algorithm [55] 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 [62]. 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 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.

4.3 HCNNG

The hierarchical clustering-based nearest neighbor graph (HCNNG) [58] 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. 3.2. 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× faster than the original HCNNG implementation (see Fig. 1), and achieves good self-relative speedup.

4.4 PyNNDescent

The PyNNDescent [56] 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 Q and retains the K closest candidates among the points $q \in Q$. 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, 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 ∼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, ParlayPyNN, 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.

4.5 Search and Layout Optimizations

In our experiments we use the *same* beam search algorithm across all of our implementations of ParlayDiskANN, ParlayHCNNG and ParlayPyNN 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 [45]. 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 Experimental Evaluations

In this section, we evaluate ParlayANN and present interesting findings from experiments at the end. We implement ParlayANN using C++ using ParlayLib [22] to support fork-join parallelism. We also use some standard building blocks (e.g., sorting, semisorting, partition) from ParlayLib.

5.1 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 [66]. The widely used *BIGANN dataset*[1] consists of SIFT image sim-

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

ilarity descriptors applied to images [48, 49, 66]. 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 [32]. 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 [19].

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

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

For Figure 1 we used an AWS c7i-series virtual machine with one 4th Generation Intel® Xeon® Gold Processor with 96 vCPUs available to the user, and 192 GiB 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. 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 [68], HNSW [55], HCNNG [58], and PyNNDescent [56], 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 the supplemental material, and provide a more complete list of algorithms that we did not include in the study, along with the reasons for their exclusion.

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 in the supplemental material, which are chosen to give the best performance based on both our own experiments and the literature.

	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 1. Build times (hours) on hundred million scale datasets.

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

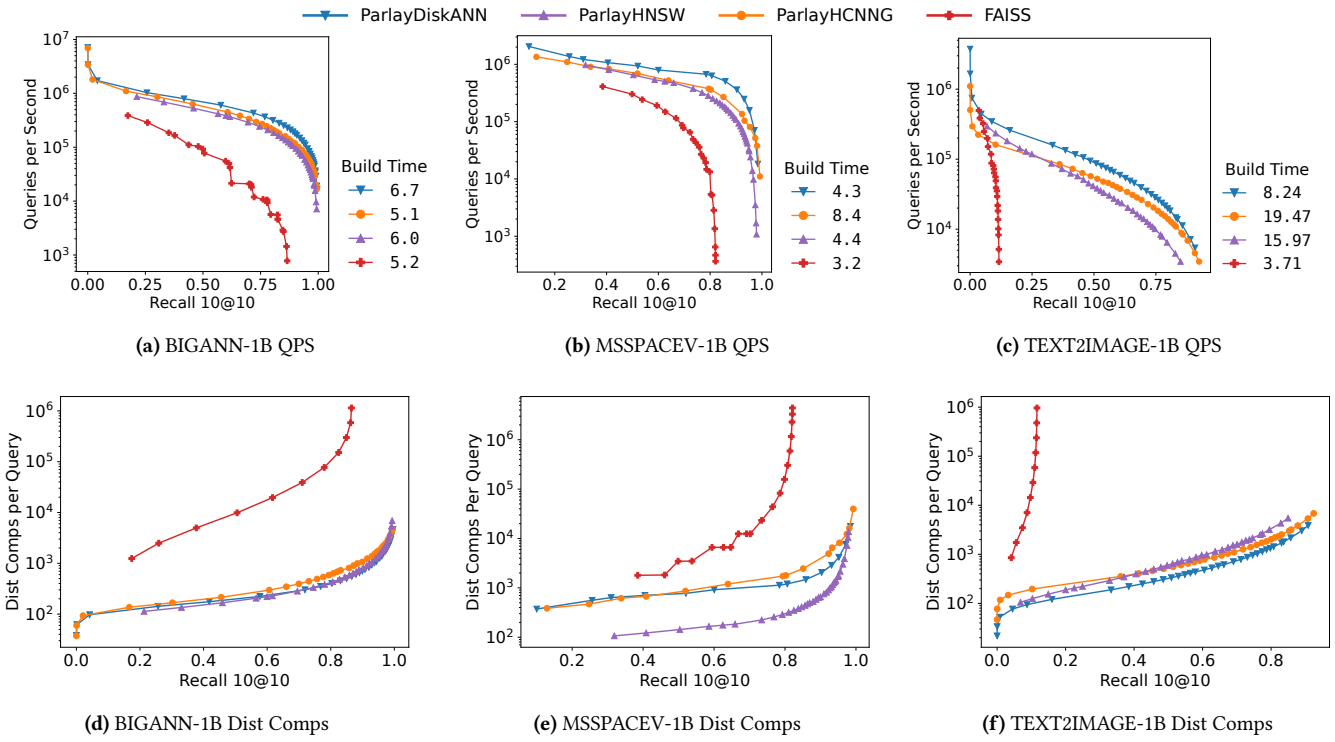
5.2 Comparison with ANN Benchmarks

First of all, we demonstrate the single thread performance of ParlayANN on BIGANN-1M in Fig. 5. We refer to the parameter settings in the ANN Benchmarks framework [16], and compare to the publicly-available numbers on the website. The single-thread performance of ParlayANN roughly match the results on ANN Benchmarks website [18]. Due to FALCONN's poor performance BIGANN-1M, and its correspondingly low performance on the hundred million and billion size datasets, we do not include FALCONN in further figures.

5.3 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. 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).

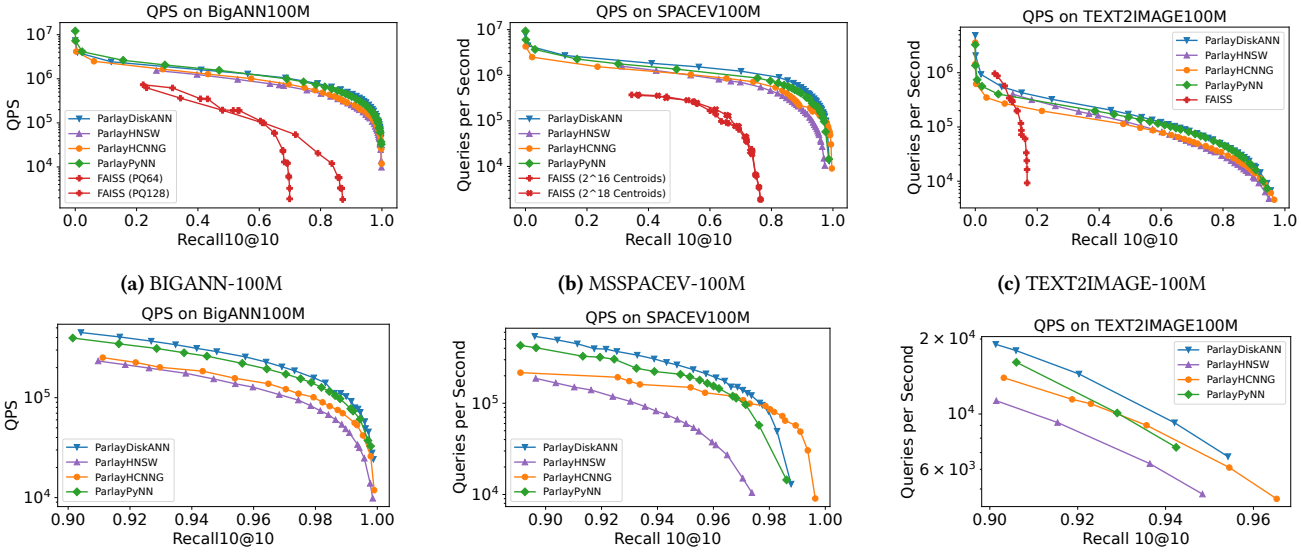
For DiskANN, we find about 1.2× improvement in performance by ParlayANN. The original DiskANN scales well to 30 to 60 threads but eventually the use of locks leads to performance degradation on more threads. HNSW suffers from similar locking-related issues, and ParlayHNSW performs much better with more than 50 threads, and eventually achieves 1.4× better performance. As mentioned in Sec. 3, 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 im-



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



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

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

Figure 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. 1

plementation when using all threads. PyNNDescent's original implementation used Numba [52] for parallelism and did not scale beyond 16 threads on our machine. Our implementation eventually becomes 28× faster than their parallel implementation.

5.4 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. 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

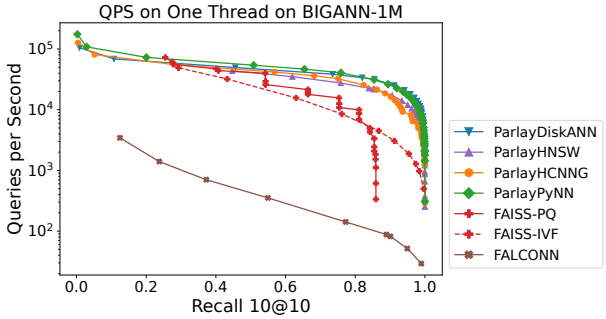


Figure 5. QPS on a single thread on BIGANN-1M. Shown to compare with ANN-benchmarks.

their indexes presented on the side. As mentioned in Sec. 3, ParlayPyNN 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*[2].

For BIGANN and MSSPACEV, FAISS did not achieve a recall higher than 0.8 even with very low QPS. At 0.8 recall, FAISS has orders of magnitude lower QPS than the graph-based algorithms (although at lower recall values, the gap between algorithms is significantly smaller).

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 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.

5.5 Dataset Size Scaling

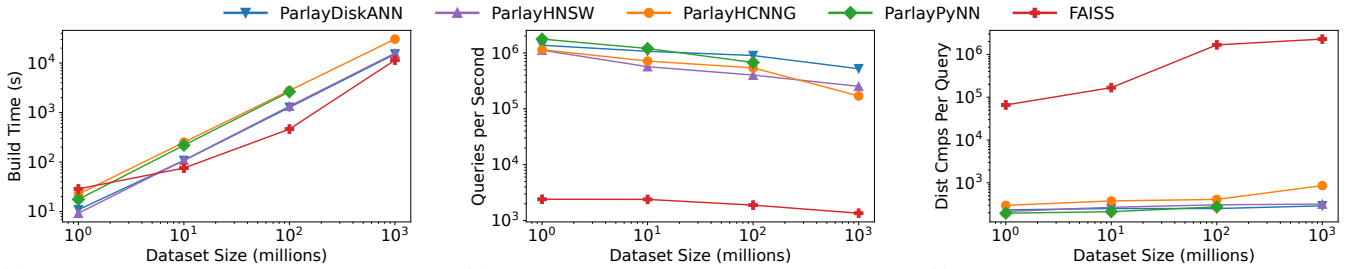
How do ANNS algorithms scale as we increase the size of the dataset? We start with the MSSPACEV dataset as an example to explore this question and present the result in Fig. 6 at a fixed recall of 0.8. In addition to build times and QPS, we also measure the average distance computations per query for 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. 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 ParlayPyNN, 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. 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. 6b and 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 ParlayPyNN 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.

[2]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 [47]). The results we present are the best we achieved after extensive experiments.



(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 6. Figures showing the effect of dataset size on different metrics using the MSSPACEVdataset.

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.

5.6 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.

6 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 [38], HNSW [55], DiskANN [68], but the academic literature includes many other graph-based approaches [1, 2, 4, 6, 7, 27, 29, 35, 37, 43–45, 53, 56, 58, 64, 75].

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 [69], FAISS-IVF [36, 48, 50], and FALCONN [13], along with many others [2, 4, 17, 30, 42, 51, 61, 73].

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) [15, 21, 40, 51], useful for many such applications [23, 31, 74]. Their search methods are subject to the curse of dimensionality, but there are some modified tree-based approaches for high dimensional search [3, 5, 54, 57].

In this paper, 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 [59, 60] for graph-based algorithms, and improving scalability for tree-based algorithms on time series data [63].

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 [20, 49, 69, 73]. The results for FAISS [36], the best known of the algorithms in this class, have been reported for the BIGANN and DEEP billion scale datasets [47]. 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 [68], a graph-based algorithm, gives numbers for BIGANN and DEEP for a billion points. They present limited comparisons to the FAISS [36] and IVFOADC+G+P algorithms [20]. The

SPANN system [30] 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 [30]. The query throughput is therefore much lower than in-memory-based systems we report on in this paper, even accounting for machine size (i.e., number of cores), although they have the advantage of needing less primary memory.

Johnson, Douze, and Jégou [50] 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 [72]. Recent work on BLISS [42] 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 [4, 38, 51, 55].

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 [16]. 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 [66], 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.

7 Conclusion and Future Work

We presented ParlayANN, which implements four parallel deterministic graph-based ANNS algorithms that scale to billion-scale inputs on a single machine with high recall. Our implementations avoid the use of locks, achieve better scalability than existing implementations, and also outperformed existing non-graph implementations in the ability of achieving high recall, especially on OOD queries.

Our experiments illuminate many opportunities for future work. Here we highlight some of the most interesting. One of our most surprising conclusions is the strong performance of HCNNG, a relatively lesser-known ANNS algorithm that does not appear in ANN Benchmarks. This brings us to our first open question:

Open Question 1. Can the techniques from incremental graph algorithms be combined with insights from HCNNG to produce an algorithm which dominates both?

Another surprising result was the clear inability of IVF and LSH algorithms to answer out-of-distribution queries. This brings us to the next open problem:

Open Question 2. How can IVF and LSH algorithms be adapted to perform better on out-of-distribution queries?

While our work focuses on comparison of indexing methods, quantization and/or compression of vector data is an important tool in approximate nearest neighbor search. Another open direction is:

Open Question 3. How can quantization methods be efficiently parallelized and made deterministic, and how do such methods affect the choice of ANNS algorithms?

Some closely-related problems to ANNS are *range searches* (e.g., axis-align or fixed-radius, counting or reporting all, etc.). This brings us to the final open question:

Open Question 4. How do graph-based and other existing ANNS algorithms adapt to various range search problems at billion or larger scale?

Acknowledgments

We thank the anonymous reviewers for their useful comments. Our experimental work was supported in part by Azure cloud compute credits granted by Microsoft Research. The authors were supported by NSF grants DGE1745016, DGE2140739, CCF-2103483, CCF-2238358, CCF-2227669, CCF-2119352, CCF-1919223 and CNS-2317194.

References

[1] 2016. KGraph: A Library for Approximate Nearest Neighbor Search. Webpage. https://github.com/aaalgo/kgraph
[2] 2021. N2. Webpage. Retrieved December 27, 2022 from https://github.com/kakao/n2
[3] 2022. Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk. Webpage. Retrieved January 1, 2023 from https://github.com/spotify/annoy
[4] 2022. OpenSearch k-NN. Webpage. Retrieved December 27, 2022 from https://github.com/opensearch-project/k-NN
[5] 2022. Spacial algorithms and data structures. Webpage. Retrieved December 27, 2022 from https://docs.scipy.org/doc/scipy/reference/spatial.htmla
[6] 2022. Vald: A Highly Scalable Distributed Vector Search Engine. Webpage. Retrieved December 27, 2022 from https://github.com/vdaas/vald
[7] 2022. vespa. Webpage. Retrieved December 27, 2022 from https://github.com/vespa-engine/vespa
[8] 2023. Apache Lucene. https://lucene.apache.org/
[9] 2023. CHATGPT-Retrieval-plugin/readme.md. https://github.com/openai/chatgpt-retrieval-plugin/blob/main/README.md
[10] 2023. Microsoft Bing Search Engine. https://www.bing.com/new
[11] 2023. Pinecone: Vector Database for Vector Search. https://www.pinecone.io/
[12] 2023. Weaviate: The AI Native Vector Database. https://weaviate.io/
[13] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Annual Conference on Neural Information Processing Systems*

(NeurIPS). 1225–1233.

[14] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. 34, 2 (01 Apr 2001).

[15] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*. ACM/SIAM, 271–280.

[16] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020).

[17] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. 2019. PUFFINN: Parameterless and Universally Fast Finding of Nearest Neighbors. In *Annual European Symposium on Algorithms (ESA)*, Vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:16.

[18] ANN Benchmarks Authors. 2023. ANN-Benchmarks. https://ann-benchmarks.com/index.html

[19] Dmitry Baranchuk and Artem Babenko. 2021. Benchmarks for Billion-Scale Similarity Search. Webpage. Retrieved March 16, 2023 from https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search

[20] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *Computer Vision - ECCV 2018 (Lecture Notes in Computer Science, Vol. 11216)*. Springer, 209–224.

[21] Alina Beygelzimer, Sham M. Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML) (ACM International Conference Proceeding Series, Vol. 148)*. ACM, 97–104.

[22] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 507–509. https://doi.org/10.1145/3350755.3400254

[23] Guy E. Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208. https://doi.org/10.1137/1.9781611977042.16

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

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

[26] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. 46, 5 (1999), 720–748.

[27] Leonid Boytsov and Bilegsaikhan Naidan. 2013. Engineering Efficient and Effective Non-metric Space Library. In *Similarity Search and Applications (SISAP) (Lecture Notes in Computer Science, Vol. 8199)*. Springer, 280–293.

[28] Harrison Chase. 2023. Vector DB text generation. https://python.langchain.com/en/latest/modules/chains/index_examples/vector_db_text_generation.html

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

[30] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 5199–5212.

[31] M. Connor and P. Kumar. 2008. Parallel Construction of k-Nearest Neighbor Graphs for Point Clouds. In *Eurographics / IEEE VGTC Symposium on Volume Graphics*, Hans-Christian Hege, David H. Laidlaw, Renato Pajarola, and Oliver G. Staadt (Eds.). Eurographics Association, 25–31. https://doi.org/10.2312/VG/VG-PBG08/025-031

[32] SpaceV Contributors. 2021. SPACEV1B: A billion-Scale vector dataset for text descriptors. Webpage. Retrieved March 16, 2023 from https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

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

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

[36] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous Codes. In *Computer Vision - ECCV 2016 (Lecture Notes in Computer Science, Vol. 9906)*. Springer, 785–801.

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

[38] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[39] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[40] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–272.

[41] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 24–34. https://doi.org/10.1145/2755573.2755597

[42] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. 2022. BLISS: A Billion scale Index using Iterative Repartitioning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 486–495.

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

[44] Masajiro Iwasaki. 2016. Pruned Bi-directed K-nearest Neighbor Graph for Proximity Search. In *Similarity Search and Applications (SISAP) (Lecture Notes in Computer Science, Vol. 9939)*. 20–33.

[45] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *CoRR* abs/1810.07355 (2018). arXiv:1810.07355 http://arxiv.org/abs/1810.07355

[46] Joseph JáJá. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[47] Herve Jegou, Matthijs Douze, Jeff Johnson, Lucas Hosseini, Chengqi Deng, and Alexandr Guzhva. 2023. FAISS Wiki. Webpage. Retrieved March 21, 2023 from https://github.com/facebookresearch/faiss/wiki

[48] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

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

[50] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale

Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547.

[51] Alex Klibisz. 2021. Tour de Elastiknn. Webpage. Retrieved December 27, 2022 from https://elastiknn.com/posts/tour-de-elastiknn-august-2021/

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

[53] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2022. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 15, 2 (2022), 246–258.

[54] Anonymous Maciej Kula, Matthew Ward. 2019. rpforest. Webpage. Retrieved December 20, 2022 from https://github.com/lyst/rpforest

[55] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[56] Leland McInnes. 2020. PyNNDescent for Fast Approximate Nearest Neighbors. Webpage. Retrieved December 15, 2022 from https://pynndescent.readthedocs.io/en/latest/

[57] Marius Muja and David G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *Proceedings of the Fourth International Conference on Computer Vision Theory and Applications (VISAPP)*. INSTICC Press, 331–340.

[58] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognit.* 96 (2019).

[59] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2022. Speed-ANN: Low-Latency and High-Accuracy Nearest Neighbor Search via Intra-Query Parallelism. *arXiv preprint arXiv:2201.13007* (2022).

[60] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2023. 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*. 313–328.

[61] Ninh Pham and Tao Liu. 2022. Falconn++: A Locality-sensitive Filtering Approach for Approximate Nearest Neighbor Search. *CoRR* abs/2206.01382 (2022). https://doi.org/10.48550/arXiv.2206.01382 arXiv:2206.01382

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

[63] Amir Raoofy, Roman Karlstetter, Martin Schreiber, Carsten Trinitis, and Martin Schulz. 2023. Overcoming Weak Scaling Challenges in Tree-Based Nearest Neighbor Time Series Mining. In *International Conference on High Performance Computing*. Springer, 317–338.

[64] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[65] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[66] Harsha Vardhan Simhadri, George Williams, Martin Aumuller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. 2021. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *Annual Conference on Neural Information Processing Systems (NeurIPS) (Proceedings of Machine Learning Research, Vol. 176)*. 177–189.

[67] Colette Stallbaumer. 2023. Introducing Microsoft 365 copilot. https://www.microsoft.com/en-us/microsoft-365/blog/2023/03/16/introducing-microsoft-365-copilot-a-whole-new-way-to-work/

[68] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 13748–13758.

[69] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over one Billion Tweets using Parallel Locality-Sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (2013), 1930–1941.

[70] tawalke. 2023. PR: SK Vectordb Connector Work - Merging forked branch; PR from Fork to SK Branch by Tawalke · pull request 83 · Microsoft/Semantic-Kernel. https://github.com/microsoft/semantic-kernel/pull/83

[71] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978.

[72] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. P. A. Lensch. 2016. Efficient large-scale approximate nearest neighbor search on the GPU. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[73] Yan Xia, Kaiming He, Fang Wen, and Jian Sun. 2013. Joint inverted indexing. In *Proceedings of the IEEE International Conference on Computer Vision*. 3416–3423.

[74] Rahul Yesantharao, Yiqiu Wang, Laxman Dhulipala, and Julian Shun. 2021. Parallel Batch-Dynamic kd-Trees. *CoRR* abs/2112.06188 (2021). arXiv:2112.06188 https://arxiv.org/abs/2112.06188

[75] Jiaru Zhang, Ruhui Ma, Tao Song, Yang Hua, Zhengui Xue, Chenyang Guan, and Haibing Guan. 2022. Hierarchical Satellite System Graph for Approximate Nearest Neighbor Search on Big Data. *ACM/IMS Trans. Data Sci.* 2, 4 (2022).

A Artifact Instructions

Code Availability Our repository can be found at the following https://zenodo.org/records/10223558. The separate repository that we use to run the algorithms and generate plots (which you should install rather than the above) can be found at this https://zenodo.org/records/10223597. This repo is what you should download and install, and it contains a Docker container for installing our other library.

A.1 Getting Started

We use the Big ANN Benchmarks (https://github.com/harsha-simhadri/big-ann-benchmarks/tree/main) repository to generate our plots. We have uploaded a fork of this repository to Zenodo (same link as above); you can use it to install a branch of our library that was also uploaded using Zenodo.

Installation The only prerequisite is Python3.10 and Docker. All commands are assumed to be run in the top-level directory unless otherwise stated. You may wish to use a conda environment for python commands.

1. Download and unzip the repo
2. Run `pip install -r requirements_py3.10.txt`
3. Install docker by following instructions: https://docs.docker.com/engine/install/ubuntu/. You should also to follow the post-install steps for running docker in non-root user mode.
4. Install the necessary Docker images as follows by running `python3.10 install.py -algorithm parlayann`
5. Create the result folder by calling `mkdir results` in the top-level directory

Datasets The evaluation assumes that datasets are stored in the `data/` directory inside the main folder. You should use a symbolic link to a directory on an SSD depending on your memory constraints (this is discussed further in the Evaluation section, note that the resulting saved graphs will also be written to this folder). Download a small toy dataset using:

```
python3.10 create_dataset.py --dataset random-xs
```

Toy Evaluation Finally, run the algorithms on the toy dataset to confirm that they run as expected. The 'run.py' file builds a nearest neighbor graph (as described in Section 3 of our paper) and queries it, recording the results for later analysis.

```
python3.10 run.py --algorithm ParDiskANN
    --dataset random-xs --definitions
    artifact_eval.yaml
python3.10 run.py --algorithm ParHCNNG
    --dataset random-xs --definitions
    artifact_eval.yaml
python3.10 run.py --algorithm
    ParPyNNDescent --dataset random-xs
    --definitions artifact_eval.yaml
python3.10 run.py --algorithm ParHNSW --
    dataset random-xs --definitions
    artifact_eval.yaml
```

Now, generate a plot of results:

```
python3.10 plot.py --dataset random-xs
```

The plot can be found in `results/random-xs.png`.

A.2 Evaluation

Our paper presents results specifically on billion-size datasets. It took around 90 hours on an AWS c6i with 128 vCPUs to build all of the graphs, and requires around 1.5 terabytes of main memory. It additionally requires about 2 TB to store all the datasets, and then an additional 10 TB to store all the graph indices. We assume that the reviewer will not have the relevant time or resources for this evaluation. The evaluation for 100 million size took about 16 hours on an AWS c6i with 128 vCPUs to build each graph and requires about 150 GB of main memory as well as 1 TB storage. We assume that the reviewer may possibly be able to do the 100 million scale evaluation, but we also provide instructions to reproduce the results at the 10 million scale in case that is preferred (the memory requirements scale down by exactly a factor of 10 when going from 100 million to 10 million).

In the next section, we describe how to reproduce the thread scaling results in Figure 1. Then, we provide scripts for reproducing the results in Figure 4 at either the 10 million or 100 million scale.

A.3 Thread Scaling

In Figure 1, we show speedup of build times relative to the original (i.e. not lock-free) algorithm on one thread. Since the artifact does not require the use of other researchers' code, we instead plot build times for our own implementations on the y-axis and number of threads on the x-axis.

First, download the dataset:

```
python3.10 create_dataset.py --dataset bigann-1M
```

The next script builds the graph for each algorithm on [1,2,8,16,24,32,48,64,96] threads. If your evaluation machine has fewer threads, you can access the script and comment out the lines corresponding to the thread counts you wish to exclude. Note that if you are monitoring thread usage using e.g. 'htop', some of the steps outside building (e.g. loading the dataset, saving and loading the graph, etc.) may still use all available threads.

```
bash thread_scaling_bigann.sh
```

After the run concludes, use the following commands to generate the plot:

```
python3.10 plot.py --dataset bigann-1M -
    x threads -y build --out results/
```

```
    threadscale_bigann1M -Y log
```

You should find the plot in the 'results' folder. It should be titled `threadscale_bigann1M.png`.

A.4 Ten Million Scale QPS/Recall Plots (Figure 4)

First, download the datasets:

```
python3.10 create_dataset.py --dataset
    bigann-10M
python3.10 create_dataset.py --dataset
    msspacev-10M
python3.10 create_dataset.py --dataset
    text2image-10M
```

The download of a file occasionally fails due to connectivity and needs to be repeated. Running the download multiple times will not download duplicates of the datasets, so just rerun the same command if you get any error messages.

Next, run each algorithm using the following script:

```
bash run_10M_builds.sh
```

After the run concludes, use the following commands to generate plots:

```
bash create_10M_plots.sh
```

Next, navigate to the 'results/' folder. It should have generated three new QPS/recall plots, one for each dataset.

A.5 Hundred Million Scale QPS/Recall Plots (Figure 4)

The explanation for these instructions is exactly analogous to the instructions for 10M size datasets:

```
python3.10 create_dataset.py --dataset
    bigann-100M
python3.10 create_dataset.py --dataset
    msspacev-100M
python3.10 create_dataset.py --dataset
    text2image-100M
bash run_100M_builds.sh
bash create_100M_plots.sh
```