## Efficient Graph-Based Approximate Nearest Neighbor Search Achieving: Low Latency Without Throughput Loss

Jingjia Luo, Mingxing Zhang, Kang Chen Xia Liao, Yingdi Shan, Jinlei Jiang, Yongwei Wu Tsinghua University Beijing, China

luojj22@mails.tsinghua.edu.cn,zhang\_mingxing@mail.tsinghua.edu.cn,chenkang@tsinghua.edu.cn, liaoxia5018@163.com,shanyd@tsinghua.edu.cn,jjlei@tsinghua.edu.cn,wuyw@tsinghua.edu.cn

#### **Abstract**

The increase in the dimensionality of neural embedding models has enhanced the accuracy of semantic search capabilities but also amplified the computational demands for Approximate Nearest Neighbor Searches (ANNS). This complexity poses significant challenges in online and interactive services, where query latency is a critical performance metric. Traditional graph-based ANNS methods, while effective for managing large datasets, often experience substantial throughput reductions when scaled for intra-query parallelism to minimize latency. This reduction is largely due to inherent inefficiencies in the conventional fork-join parallelism model.

To address this problem, we introduce AVERSEARCH, a novel parallel graph-based ANNS framework that overcomes these limitations through a fully asynchronous architecture. Unlike existing frameworks that struggle with balancing latency and throughput, AVERSEARCH utilizes a dynamic workload balancing mechanism that supports continuous, dependency-free processing. This approach not only minimizes latency by eliminating unnecessary synchronization and redundant vertex processing but also maintains high throughput levels. Our evaluations across various datasets, including both traditional benchmarks and modern largescale model generated datasets, show that AVERSEARCH consistently outperforms current state-of-the-art systems. It achieves up to 2.1×-8.9× higher throughput at comparable latency levels across different datasets and reduces minimum latency by  $1.5 \times$  to  $1.9 \times$ .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Association for Computing Machinery. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00 https://doi.org/10.1145/nnnnnnnnnnnnnnnnnn

#### **ACM Reference Format:**

#### 1 Introduction

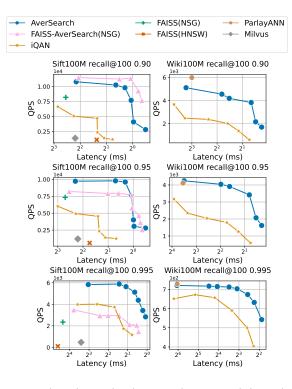
Scaling laws for large language models indicate that increasing model size enhances performance. As a result, the dimensionality of embedding vectors for objects like documents and images has expanded from approximately 100 dimensions in earlier benchmarks (e.g., SIFT [33], DEEP [1]) to thousands in recent models (e.g., OpenAI [46], Cohere [11]). This growth not only improves the accuracy of similarity measurements but also significantly elevates the computational demands of conducting Approximate Nearest Neighbor Searches (ANNS).

For instance, inspired by OpenAI o1's advancements [48], Retrieval-Augmented Generation (RAG) [64] is utilized to improve the generation quality, which increases with the number of retrieval rounds, causing the response time to scale proportionally with retrieval latency. Additionally, in attempts using ANNS in sparse attention calculations [8, 35, 65], retrieval occurs for every layer and token, further emphasizing the need to minimize latency. This scenario presents a challenging balancing act among accuracy, throughput, and latency, a trio of metrics that often conflict with one another. In scenarios with stringent low-latency requirements, goodput, i.e., throughput that meets specific latency constraints, must also be considered.

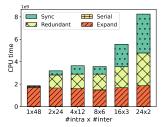
Researchers have developed a multitude of algorithms to improve searching performance, including hashing-based [3, 7, 36], quantization-based [20, 56, 59], tree and graph based methods [17–19, 38, 39, 41, 42, 51, 52, 55, 68], and the combinations of multiple methods [6, 29, 37, 61]. Graph-based algorithms, in particular, are proven by many studies [4, 5, 43] that excel at delivering optimal throughput-recall trade-offs.

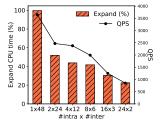
,

Many benchmark studies [4, 5, 43] have shown that graph-based algorithms excel at delivering optimal throughput-recall trade-offs.



**Figure 1.** The relationship between latency and throughput, expressed in queries per second (QPS), varies across different parallelism settings. We leverage all 48 available cores, organizing them into " $intra \times inter$ " groups, ranging from " $1 \times 48$ " to " $24 \times 2$ ". Here, "intra" represents the number of threads dedicated to each query, while "inter" indicates the number of independent concurrent queries. The analysis uses two well-known datasets, SIFT100M and Wiki100M [10], each containing 100 million vectors with dimensions of 128 and 768, respectively. The evaluation covers various recall levels from 0.9 to 0.995.





(a) iQAN CPU time breakdown over a query

**(b)** iQAN valid expansion CPU time percentage V.S. QPS

**Figure 2.** iQAN execution time distribution under different parallelism strategies with Wiki100M and recall at 0.9.

Traditional approaches enhance search performance on graphs by optimizing their structures, whereas recent studies [50, 60] incorporate intra-query parallelism to reduce latency. However, implementing such parallel strategies presents significant challenges. As an illustration, Figure 1 presents the throughput-latency trade-off across different recall levels using iQAN [50], a state-of-the-art parallel graph-based ANNS engine. The figure shows that while increasing intraquery parallelism decreases latency, it also leads to a drop in overall throughput. In the Sift100M 0.995 benchmark, the QPS drops from 3993 to 1166 as the number of intra-query threads increases from 1 to 24, utilizing all 48 threads.

Further analysis reveals that this problem stems from **inherent inefficiencies in the traditional fork-join model** of parallelism. Even with a balanced workload distribution, variations in execution times among threads cause substantial synchronization overhead. Figure 2a breaks down the execution time under different intra-query parallelism settings, highlighting that an increase in intra-query parallelism exacerbates synchronization and redundant processing burdens. This escalation reduces the proportion of genuinely effective work (the "expand" portion), directly contributing to a decline in overall throughput, as depicted in Figure 2b. More details on this analysis will be provided in §3.

In response to these challenges, we introduce AVERSEARCH (Asynchronous vertex Search), a parallel graph-based ANNS framework that maintains high throughput while simultaneously reducing latency through enhanced intra-query parallelism. The cornerstone of AVERSEARCH is its innovative fully asynchronous architecture, which divides the job of query processing among three distinct roles of threads, that crucially, rarely wait on each other. This architecture allows memory-intensive distance calculation to continuously proceed without delays, thereby optimizing memory bandwidth utilization. Moreover, AVERSEARCH utilizes a dynamic online workload balancing approach, enabling faster threads to speculatively process additional vertices rather than idling, while slower threads move on to subsequent iterations without being burdened with unnecessary vertices. This strategy leads to near-zero synchronization overhead and minimizes the processing of redundant tasks.

To evaluate the effectiveness of AVERSEARCH, we conducted experiments on various datasets, including classical ones with approximately 100-dimensional vectors as well as recent large model-generated datasets from OpenAI [46, 47] and Cohere [10] (with dimensions exceeding a thousand). We compared AVERSEARCH not only with iQAN, the leading parallel ANNS engine but also with prominent commercial vector search engines like Milvus [22, 54] and FAISS [15] across various settings. Results demonstrate that AVERSEARCH achieves up to 2.1×-8.9× higher throughput at comparable latency levels across different datasets. Moreover, it is capable of achieving 1.5× to 1.9× lower minimum average latency due to better intra-query parallelism utilization.

Detailed analysis involving memory utilization, vertex processing redundancy, and comparing with quantization-based indexes further justifies our observations and pinpoints the sources of optimization. These findings highlight the capacity of Aversearch to effectively manage the complex tradeoffs between throughput, accuracy, and latency in real-time search applications.

## 2 Background and Related Works

## 2.1 Approximate Nearest Neighbor Search

In high-dimensional embedding spaces, similarity between entities is measured by metrics like inner product or the Euclidean norm, which supports vector search techniques for identifying the top K nearest vectors to a query. The approximate version of this process, known as ANNS, aims to efficiently find the approximate K closest vectors to a query vector  $q \in \mathbb{R}^d$  within a large set of N vectors in a d-dimensional space. The effectiveness of ANNS is measured by its ability to include the actual nearest neighbors among the K vectors identified.

Various indexing ANNS strategies to prevent full database scan include tree-based [52, 55, 62], hash-based [3, 36, 53], quantization-based [56, 59], and graph-based methods. These non-graph-based methods rely on partitioning and clustering the vector database, using pretrained data structures, hashing functions, or quantization codes to enable quick identification of clusters nearest to a query vector. However, they face challenges with scalability and efficiency in high-dimensional spaces due to the well-known "curse of dimensionality". Prior works on graph-based indices have demonstrated superior performance compared to state-of-the-art non-graph-based methods, achieving QPS improvements of up to 20 times [18, 19, 23, 25, 39]. Additionally, graph-based methods [28, 58] maintain a leading position in achieving a higher QPS-recall trade-off in the ANN Benchmarks [4, 5].

#### 2.2 The Latency Requirements for ANNS

With the increasing dimensionality and the adoption of RAG in online and interactive services, latency requirements are becoming more stringent, necessitating a focus beyond throughput alone. Firstly, Large embedding models significantly increase the dimensionality of embedding vectors, thereby introducing substantial latency overhead. Moreover, following the inference-time scaling law, a single user request may demand multiple rounds of RAG to generate a response. Consequently, response time scales proportionally with retrieval latency multiplied by the number of retrievals. Lastly, ANNS is also increasingly applied in long-context LLM inference for attention retrieval [8, 35, 65]. Due to the sparsity of attention, the algorithms leverage softmax's natural ability to select key-value pairs with high attention scores, where retrieval occurs for every layer and token in a serial manner. Therefore, ANNS systems should adopt

new design objectives to accommodate emerging application scenarios. Under strict latency constraints, considering goodput-throughput under varying latency conditions—can better address these evolving demands.

# 2.3 Graph Index Searching and Existing Parallelizing Method

2.3.1 Best-First Search algorithm (BFiS) In graph-based ANNS, a similarity graph G(V, E) is constructed, where each vertex  $v \in V$  represents a vector, and each edge  $(v, u) \in E$  indicates that *u* is among the closest neighbors of *v*. Despite the variety of methods used to construct G, a unified Best-First Search algorithm (BFiS) is applied across all graph-based indices for sequential querying. BFiS initiates its search from a seed vertex, selected either randomly or based on the graph's specifics, and progresses toward the query by navigating through outgoing edges. As we can see from Algorithm 1, it utilizes a priority queue Q to maintain focus solely on the L unchecked nodes nearest to the query (17), with the Lparameter being crucial for controling the search precision. In each iteration, BFiS identifies the nearest unchecked node, and proceeds to expand it. To facilitate further discussion, we introduce the term **expand** as calculating the pairwise distance between all neighbors of the nearest unchecked node and the query (13), marking it as checked, and subsequently adding its neighbors to the priority queue as new unchecked candidates for future expansion. The queue is sorted by distance to the query, ensuring that less promising candidates are replaced as new ones are introduced. The search expands unchecked nodes until none remain in the priority queue.

## Algorithm 1 Best-First Search

**Input:** query q, priority queue Q initialized with entry-node set, graph G, distance function  $\delta$ , queue capacity L, number of ANNs K

```
Output: K approximate nearest neighbors of q

1: while there's unchecked candidate in Q do

2: # path-wise parallel knob in iQAN {

3: v \leftarrow the first unchecked candidate in Q

4: Expand(v, q, Q, G, \delta)

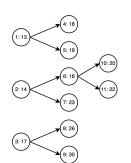
5: }

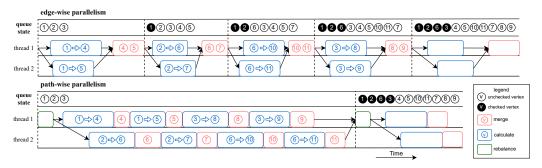
6: if |Q| > L then

7: Q.resize(L)

return the first K vertices in Q
```

2.3.2 Parallel BFiS Distributing different queries across threads sufficed to achieve high throughput but leads to significant latency, especially with large and high-dimensional databases. For example, searching a Wiki100M dataset on a single CPU thread could take up to 73.774 ms for each query, a duration that is impractical for real-time applications like retrieval augmented generation (RAG) and the





**(a)** A directed similarity graph.

**(b)** A timeline depicting different parallelism strategies. In the figure, "queue state" represents the evolving state of the global priority queue over time. " $\widehat{(x)} \Rightarrow \widehat{(y)}$ " indicates edge (x, y) in the similarity graph.

Figure 3. Edge-wise parallelism v.s. path-wise parallelism (§2.3.2)

## Algorithm 2 the Expand operation

**Input:** the expanded candidate v, query q, priority queue Q, graph G, distance function  $\delta$ 

```
1: v.state \leftarrow checked
2: B \leftarrow \emptyset
3: # edge-wise parallel knob
4: for all neighbor u of v in G do
5: if u is not visited then
6: mark u as visited
7: u.state \leftarrow unchecked
8: u.distance \leftarrow \delta(u, q)
9: B \leftarrow B \cup u
10: Q \leftarrow Q.merge(B)
```

whole Wikipedia contain even much more pages than 100M. Consequently, leveraging intra-query parallelism becomes imperative for latency reduction.

Traditional graph processing systems [9, 24, 44] support parallel BFS, but they struggle with BFiS due to strict data dependencies and frequent synchronizations thus a specialized solution is needed. A direct strategy for intra-query parallelism is *edge-wise* parallelism, which delegates distance calculation for different neighbors of the active vertex to various threads (i.e. parallelizing the for loop bellowing l3 of Algorithm 2). However, this approach is hampered by the considerable overhead from frequent synchronization. To circumvent this, the state-of-the-art parallel graph index ANNS algorithm, iQAN [50], observes that candidates at the forefront of *Q* are highly likely to be expanded in subsequent steps, permitting the speculative expansion of multiple candidates without significantly compromising accuracy.

Therefore, iQAN adopts *path-wise* parallelism. Essentially, it parallelizes the code block below l2 of Algorithm 1. With a larger parallel section, iQAN significantly reduces the synchronization frequency. It distributes the priority queue into thread-local queues before each parallel epoch, allowing each thread to expand its assigned paths independently. However,

global synchronization is still necessary for global queue pruning and rebalancing among thread-local queues.

**2.3.3 Example** Figure 3 compares edge-wise and pathwise parallelism with an example graph, where vertices are represented by circles and edges depicted by arrows. In the graph each vertex is labeled in the in the format "ID: distance" to denote its ID and the distance to the query. As we can see, with edge-wise parallelism, the two outgoing edges of vertex ① are assigned to different threads, which calculate the distance between the query and vertex ④ and ⑤ independently. This parallel calculation is immediately subsequent to a global synchronization for merging ④ and ⑤ into the priority queue.

In contrast, with a predefined *width* of 2 in path-wise parallelism, each of the two threads can expand 2 candidates independently in the subsequent parallel epoch. During the parallel epoch, each thread consistently expands the first unchecked vertex in its local sub-queue. This process repeats until all vertices in the sub-queue are checked or the preset expansion limit (2 in this case) is reached.

As depicted in Figure 3 (b), the three unchecked vertices in the global queue are initially distributed to each thread's local sub-queues in a round-robin fashion. Thread 1 then expands vertices ① and ③, incorporating their neighbors into its local sub-queue. Similarly, Thread 2 expands ②, and subsequently expand ⑥ without waiting for global synchronization.

However, iQAN's path-wise parallelism reduces the frequency of global synchronization-based rebalancing, from four times to just once in our example. But, this reduction, whose extent is controlled by the parameter width, comes at the expense of potentially expanding unnecessary vertices, and hence **leads to redundant computation**. For example, if the global queue's length limit L is set to 5, vertices @ would be pruned following the first parallel epoch. But with a width of 3 instead of 2, vertices @ and @ would be expanded before synchronization. Consequently, expanding

® becomes a redundant task, highlighting the inefficiency introduced by an excessively large *width*.

## 3 Intuition and Design Choices

## 3.1 Limitations of Handling Fork-Join Jobs on CPUs

While path-wise parallelism reduces synchronization frequency by extending the parallel phase, it still relies on the traditional fork-join model. In ANNS, distance calculation tasks during expansion are typically managed by parallel threads spawned in the fork phase. In contrast, pruning and rebalancing tasks require a comprehensive view of all subqueues to achieve global ordering, making them suited for the join phase of the fork-join model. However, our analysis reveals a fundamental dilemma in using the fork-join model for graph-based ANNS, leading to scalability issues.

Specifically, as we enhance intra-query parallelism by increasing the number of intra-query threads through iQAN to reduce latency, we encounter a significant drop in throughput, measured in Queries Per Second (QPS). As demonstrated by Figure 4, the throughput decreases by 37.7%-76.7% times when intra-query parallelism is scaled from 1 to 24.

To delve into the root cause of this issue, we analyze the CPU time allocated to each query by dividing it into four main categories. The time dedicated to the joining phase is termed "serial" time, solely managed by the master thread. Conversely, the parallel forking phase's time is dissected into several components: 1) "Expand" time, representing the duration of necessary vertex expansions and is the only period contributing productively to actual progress; 2) "Redundant" time, encompassing the unnecessary expansions that would have been pruned in a serial execution, representing an additional tax due to parallelism. and 3) "Sync" time that encapsulates the remaining CPU time outside the forking and joining phases, typically arising from thread synchronization and load imbalance during transitions between these phases.

As indicated in Figure 4, a detailed breakdown analysis reveals that the decrease in throughput is mainly attributed to the rise in "redundant" and "sync" times. Consequently, the share of CPU time allocated to "expand", the only productive phase, is significantly reduced.

Moreover, further examination of adjusting the *width* parameter, which controls the size of the parallel section, uncovers a dilemma. As illustrated in Figure 5, a larger parallel section could diminish the synchronization frequency but not the associated overhead. This is because a decreased synchronization frequency lessens the chances for pruning and rebalancing in the join phase, which results in more redundant calculations and worsens load imbalance. This reveals handling fork-join tasks on CPUs faces the inherent difficulties in addressing challenges related to synchronization overhead and redundant calculations.

## 3.2 Towards a Fully Asynchronous Architecture

As the dimensions of embedding vectors increase, ANNS evolves into an extremely memory-intensive application. In this context, the key to enhancing both throughput and latency lies in optimizing memory bandwidth utilization during the edge expansion phase outlined in Algorithm 2. Building on our findings, we propose an empirical formula to steer future throughput improvements:

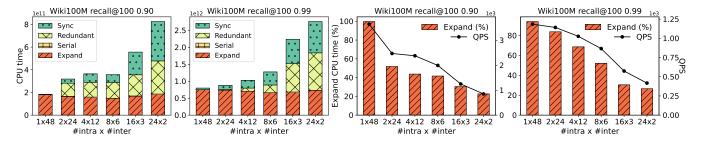
Throughput 
$$\stackrel{\propto}{\sim} EMB = PMB \times (1 - RR)$$

This equation suggests that the overall query throughput is approximately proportional to what we term as "Effective" Memory Bandwidth (EMB). This metric is calculated by multiplying the **physical memory bandwidth (PMB)** by the factor (1 - RR), where RR represents the **Redundant Ratio**. The Redundant Ratio indicates the percentage of vertices that are unnecessarily processed and could have been pruned in a serial execution. In the following discussion, we examine how the traditional fork-join model restricts both PMB and RR, and explore the potential benefits of adopting a **fully asynchronous architecture** to overcome these constraints.

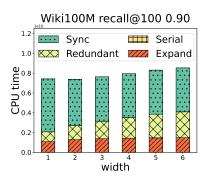
**3.2.1 Improving PMB** Figure 6 demonstrates AVERSEARCH and iQAN's physical memory bandwidth utilization on the SE-Cohere dataset under various parallelism strategies, measured by hardware counters. The configuration "intraxinter" denotes the distribution of 48 threads across inter concurrent queries, with each processed by intra threads. Despite using the same total number of threads, increasing intra-query parallelism reduces PMB significantly. At a parallelism setting of "24 × 2", PMB drops to just 108.39 GB/s, under 40% of the 266 GB/s maximum bandwidth measured by Intel mlc [26].

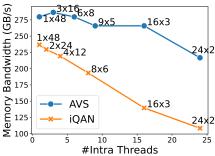
To explore this issue, we designed a micro-benchmark for parallel Euclidean distance calculations. This benchmark processes calculation between 1024-dimensional fp32 vectors using AVX512. Within the conventional fork-join paradigm with Intel OpenMP [13], each thread performs a fixed task of  $32 \times width$  distance calculations per parallel phase, mimicking the expansion of a vertex with 32 out-edges. Our findings, depicted in Figure 7, indicate that the inefficiencies of fork-join models persist even with significant width settings, with a 32-thread fork-join configuration at a width of 64 achieving only 36.19% of the theoretical memory bandwidth. An asynchronous model, where threads continuously process vertices without waiting for new tasks to be assigned, is essential to eliminate bandwidth waste.

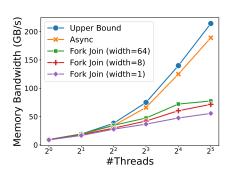
**3.2.2 Reducing RR** Besides the idle time caused by synchronization barriers, redundant computations contribute to another issue: the wasteful use of memory bandwidth. These unnecessary computations arise from exploring areas with minimal potential. In iQAN's fork-join model with static distribution, each thread is assigned a fixed number of *width* vertices, without considering their processing speed.



**Figure 4.** iQAN CPU time breakdown for a single query with varying accuracy and parallelism strategies. "Serial time" for the joining phase; "Expand time" for useful expansions during the forking phase; "Redundant time" for unnecessary expansions; and "Sync time" resulting from thread synchronization and load imbalance.







**Figure 5.** iQAN CPU time breakdown for a single query across varying *width* with 32 threads.

**Figure 6.** Memory bandwidth utilization for different parallelism strategies denoted as "*intra* × *inter*".

**Figure 7.** Comparison of memory bandwidth utilization in a micro-benchmark performing distance calculations.

As width increases, the likelihood of redundant computations grows due to less frequent global rebalancing. We expect dynamic workload distribution through an asynchronous framework, where faster threads process more vertices than slower ones. Although achieving such a perfect balance poses a challenge, **transitioning to an adaptive distribution strategy** could significantly lower the Redundant Ratio, thereby improving both efficiency and resource utilization.

## 4 AverSearch Design

As previously discussed, the key to enhancing ANNS throughput amidst high intra-query parallelism (i.e., achieving low latency) lies in crafting an architecture that: 1) allows memory-intensive distance calculation threads to continuously process vertices without delays for new tasks, thereby optimizing PMB; 2) implements an adaptive distribution strategy that dynamically allocates workloads to threads based on their current paces, thus minimizing RR. And to this end, we introduce a fully asynchronous architecture that deconstructs the traditional BFiS algorithm into three different kinds of components that communicate asynchronously.

Specifically, to enable better parallelization, as depicted in Figure 8, we divide the intra- threads handling the same query into multiple sub-groups, which also partitions the original global priority queue, *Q*, into individual sub-queues

for each thread sub-group. Each subgroup comprises a single sub-queue maintainer (sub-que) thread and multiple distance calculator (dis-cal) threads. In the illustrated case, the thread group handling a single query is divided into two sub-groups, each with one sub-que thread and two dis-cal threads. The dis-cal threads receive source vertices as tasks, process their outgoing edges, and then the distances for destination vertices are sent back to the sub-que thread, which sorts and merges these results into the sub-queue. This setup guarantees that only dis-cal threads perform the memory-intensive Expand operation outlined in Algorithm 2. Ideally, we want the sub-queue never empty so that dis-cal threads will never stall to improve the PMB. Moreover, to tackle the necessity for global rebalancing and to reduce RR, we introduce a global balancer thread. This thread scans all sub-queues to conduct redistribution and pruning actions based on a comprehensive view. Ideally, this redistribution activity should happen as frequently as possible to lower RR effectively.

In this section, we will first use a synchronous design to illustrate how these components cooperate with each other. Then, we discuss how to disentangle the interactions between the global balancer and sub-queue maintainers, the sub-queue maintainer and distance calculators, respectively, ultimately achieving a fully asynchronous architecture.

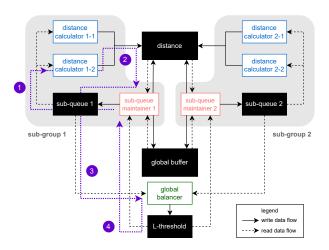
#### 4.1 Straw-man Synchronous Design

To execute the architecture depicted in Figure 8, an initial straightforward synchronous design introduces four types of inter-component dependencies (marked as  $\mathbb{O}$ ,  $\mathbb{O}$ ,  $\mathbb{O}$ , and  $\mathbb{O}$  in Figure 8). Before the search, the two sub-queues are initialized with vertices in the entry node set of the graph. As an illustration, following the edge-wise parallelism strategy, each iteration begins with ( $\mathbb{O}$ ) the sub-que thread dispatching the out-edges of the first unchecked vertex in the sub-queue to each dis-cal thread. After parallel expansion, ( $\mathbb{O}$ ) multiple destination vertices are returned to their respective sub-que thread for sorting. Then ( $\mathbb{O}$ ) the global balancer gathers all vertices in the sub-queues, calculates the global order, keeps the top L vertices – according to the preset global threshold – and prunes the others. Finally, ( $\mathbb{O}$ ) the top L vertices are redistributed to each sub-queue, and a new iteration starts.

## 4.2 Disentangling Global Balancer and Sub-que

The above straw-man implementation faces considerable delays as tasks idly wait for the preceding ones to finish. To improve performance, it's crucial to enable tasks to proceed concurrently with less dependency on their predecessors' results. This involves employing shared data structures that allow for speculative execution without the latest results.

Queue Pruning (disentangle dependency ③). Decoupling the global balancer from sub-queue maintainers is achieved by maintaining a global variable, the "L-threshold". This entails pruning less promising candidates and guiding sub-queue maintainers towards more potential-rich areas. Essentially, this process involves identifying the L-th closest visited vertices according to the global order. Yet, accurate L-threshold calculation and sub-queue rebalancing become complex due to concurrent ongoing updates. Therefore, we



**Figure 8.** An illustration of two thread groups in AverSearch. Threads (represented by white boxes) interact with each other through shared data structures (represented by black boxes) asynchronously. (§4)

devise an algorithm to **approximate a slightly larger L-threshold** based on asynchronous sub-queue snapshots. Specifically, the global balancer uses an array of pointers, one for each sub-queue. These pointers begin at the ends of the sub-queues and advance forward periodically, mimicking a merge-sort operation by moving the pointer with the largest distance forward. The process continues until only *L* candidates remain across all sub-queues after simulating pruning extra candidates by moving the pointers, at which point the largest element among those before the pointers is identified and announced as the L-threshold.

Dynamic Load Balancing (disentangle dependency ④). Given the premise that sub-queue maintainers routinely adjust their sub-queues based on the *L-threshold*, it's implied that the candidates remaining are deemed suitable for future expansion and hence a reduced RR. However, since the pruning process is governed by distance values rather than queue length, disparities can arise, leading to significant load imbalances where some queues may become substantially longer than others. To address this issue, we introduce an additional dynamic load-balancing strategy based on workstealing. This approach is to enable sub-queue maintainers with shorter queues to assist those with longer ones. Subqueue maintainers periodically check if their queue is among the longest. If so, they help alleviate congestion by transferring some of their queue's elements to the shared buffer.

## 4.3 Disentangling Sub-que and Dis-cal

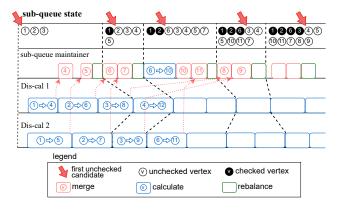
The essence of separating sub-queue maintainer threads from distance calculator threads in our architecture lies in ensuring that distance calculator threads do not wait for new tasks from the sub-queue maintainers nor directly return the calculated distance results to them. To navigate around dependency ②, we establish a distance array for each query. Each array element, corresponding to a vertex, stores the calculated distance from the query point to this vertex, along with a 'ready' flag initially set to 0. Thus, distance calculators simply save the calculated results in the array and mark the 'ready' flag as 1, allowing sub-queue maintainers to asynchronously retrieve the results from the array later.

Regarding dependency ②, if a distance calculator completes its assigned edges in the current iteration, it **doesn't idly wait** for the sub-queue maintainer to finish sorting. Instead, it preemptively fetches the second unchecked vertex from the sub-queue and begins speculative expansion. This speculative process can extend to the third and beyond, but at each speculative execution's beginning, the distance calculator first checks if the current iteration number remains unchanged. If it detects an update, indicating the sub-queue maintainer has completed a sorting and pruning round, the distance calculator resets and starts processing from the first unchecked vertex again.

This method enables adaptive task distribution without the need for a centralized coordinator. Only dis-cal threads

that operate more quickly in a given cycle will process additional speculative vertices, thus minimizing redundancy in comparison to the fixed-width strategy utilized by iQAN. We adopt edge-level rather than vertex-level or path-level parallelism to facilitate a more evenly distributed workload when paired with our strategy's ability to balance workloads dynamically.

Another inefficiency in existing methods is the redundant calculation of distances for destination vertices pointed to by multiple source vertices. However, by maintaining a distance array and 'ready' flags, distance calculators can check and skip unnecessary computations. While this skipping could complicate workload balancing in static parallelism settings, our adaptive strategy naturally mitigates this issue.



**Figure 9.** A timeline depicting the execution of a sub-queue maintainer thread and two distance calculator threads searching in the example graph of Figure 3a.

To further demonstrate our strategy, Figure 9 depicts how a sub-que thread and two dis-cal threads operate within the example graph shown in Figure 3a. The dashed lines in the figure denote misaligned asynchronous epochs, triggered by updates to the first unchecked candidate in the sub-queue, highlighted by red arrows in the figure. After the expansion of each edge, the dis-cal threads periodically check whether the first unchecked candidate in the sub-queue has been updated, guiding their actions for each epoch.

As we can see from the example, initially, the distance calculators expand vertex ①, the first unchecked candidate. Each distance calculator processes an outgoing edge from ①, in an edge-wise approach. Subsequently, as the first unchecked vertex remains unchanged, they speculatively advance to the next unchecked candidates, vertices ② and ③.

In this initial epoch, Dis-cal 1 speculatively processes the neighbors of (2) ( $\Rightarrow$  and (3) ( $\Rightarrow$  ), while Dis-cal 2 handles (2)'s other neighbor ( $\Rightarrow$  ). Concurrently, the sub-queue maintainer retrieves distances for vertices (4) and (5) from the distance array, merges them into the sub-queue, and updates the status of vertex (1) to "checked", thereby marking the start of a new epoch.

As the expansion progresses, upon completing their tasks for vertices ® and ⑦, Dis-cal 1 and 2 both notice that the first unchecked candidate has shifted from vertex ① to ②. But, since ② has already been speculatively expanded, they continue speculative processing of the out-edges for ③ and ④. During this epoch, the sub-queue maintainer effortlessly retrieves distances for ②'s neighbors from the distance array without any delay, benefiting from the speculative calculations done in the prior epoch.

In the subsequent third epoch, vertex ® emerges as the new first unchecked vertex. Upon encountering ®, which neither distance calculators had previously speculatively expanded, they prioritize ®'s expansion rather than continue their speculative execution. From the sub-queue maintainer's perspective, since the distance for ®'s neighbor, vertex ®, is unavailable, it calculates this distance and updates the distance array accordingly. This efficient cycle of speculative processing and asynchronous updating exemplifies the dynamic and continuous workflow enabled by AVERSEARCH, which minimizes latency and optimizes throughput in realtime ANNS applications.

#### 4.4 Implementation Details of AVERSEARCH

To integrate the concepts discussed earlier, this section presents the logic of each component within the thread group. Each thread follows the workflow illustrated in Figure 8, continuously polling the upstream data structure and writing to the downstream data structure. Algorithms 3, 4, and 5 provide the pseudocode for the global balancer, sub-queue maintainer, and distance calculator, respectively.

The global balancer is responsible for calculating the *L*-threshold. While its functionality can be inlined into the subqueue maintainers when the number of threads in each group is relatively small, a dedicated thread is assigned to handle this task when low latency is required and sufficient threads are available. This approach ensures frequent pruning.

The sub-queue maintainers are tasked with expanding candidates in their local sub-queues. They first attempt to read distances from the distance array to avoid redundant computation. After each expansion, they prune their sub-queues based on the *L*-threshold and perform load balancing via the global buffer.

The distance calculator threads compute the out-edges of unchecked candidates in the sub-queue in an edge-wise manner, storing their results in the distance array. They also speculatively process other unchecked vertices in the sub-queue if the queue is not updated in time.

## 5 Evaluation

#### 5.1 Evaluation Methodology

**Datasets.** Our evaluation includes six datasets of float vectors. In addition to the traditional benchmark datasets SIFT100M

## **Algorithm 3** The Global Balancer

```
Requires: Sub-queues Q[], the length limit L
Modifies: The L-threshold L - thresh
 1: while the search has not ended do
       L - thresh \leftarrow the L-th nearest distance among all
    elements in Q[]
```

## Algorithm 4 The Sub-queue Maintainer

```
Requires: Query q, graph G, distance array D[], distance
    function \delta, L-threshold L – thresh, global buffer B
Modifies: Sub-queue Q, distance array D[], global buffer B
 1: while there's unchecked vertice in Q do
        v \leftarrow the first unchecked candidate in Q
 2:
        v.state \leftarrow checked
 3:
        for all neighbor u of v in G do
 4:
            if u is not visited then
 5:
 6:
                if D[u] is not set then
                    D[u] \leftarrow \delta(u, q)
 7:
 8:
                mark u as visited
                mark u's state as unchecked
 9.
                Q \leftarrow Q \cup \{u\}
 10:
        prune candidates larger than L - thresh in Q
11:
        if Q is among the longest sub-queues then
12:
13:
            offload some candidates into the global buffer B
 14:
            pick some candidates from B into Q
15:
```

#### **Algorithm 5** The Distance Calculator

```
Requires: Query q, graph G, sub-queue Q, distance func-
    tion \delta, distance array D[]
Modifies: Distance array D[]
 1: while there's unchecked vertice in Q do
        v \leftarrow the first unchecked candidate in Q
 3:
        while Q's first unchecked candidate is v' do
 4:
 5:
            for all neighbor u of v in G do
                if u is not visited & D[u] is not set then
 6:
                    D[u] \leftarrow \delta(u,q)
 7:
            v \leftarrow the next unchecked candidate in Q
 8:
```

[33] and DEEP100M [1], which include 100 million 128dimensional and 96-dimensional float vectors, we also evaluate AverSearch with three more recent embedding datasets derived from large embedding models. SE-OpenAI [47] and SE-Cohere [10] comprise 400 thousand embedding vectors derived from Simple English Wikipedia [57]. These vectors are encoded using the 1536-dimensional OpenAI textembedding-3-small embedding model [49] and the 768-dimensional 8x6", "12x4", "16x3", and "24x2", and plotted the convex hull Cohere EmbedV3 model [11], respectively. Additionally, we

use WIKI100M, which includes 100 million embedding vectors from a random subset of the entire Wikipedia site, processed using two Cohere models that produce embedding vectors with 768 [10] and 1024 [12] dimension, respectively. **Comparison.** We have implemented AVERSEARCH in over 7000 lines of C++ code. Our implementations support multiple similarity graphs, including NSG [19], SSG [18], and Vamana index [29] constructed using ParlayANN [40]. For each dataset, we evaluate the search performance using the most accurate index that can be constructed within a reasonable time. We use NSG for relatively small datasets, and Vamana index for Wiki100M, since their construction time of NSG exceeds 7 days. Also, we integrate AVERSEARCH into FAISS [16], making intra-query parallelism available within FAISS' framework.

We compare AverSearch and FAISS-AverSearch with iQAN [50], the state-of-the-art parallel ANNS system, ParlayANN [40], the state-of-the-art index construction engine which also support searching, and Milvus [54], FAISS [15], two leading vector search engines. For iQAN and ParlayANN, search hyper-parameters are adopted as suggested by the provided script.

To evaluate the performance of the tested systems, we measure latency and Queries Per Second (QPS) while targeting a predefined accuracy threshold. Accuracy is gauged using recall@100, which verifies the accuracy of the search results by confirming the presence of the true top 100 nearest neighbors. Each test fully utilizes all 48 physical cores of our testing socket and experiments with various parallelism strategies, both inter-query and intra-query. Our accuracy benchmarks range from 0.9 to 0.995, covering a broad spectrum of vector search applications.

**Test Platform.** Our experiments are conducted on a system equipped with an Intel® Xeon® Platinum 8457C processor (2.60GHz), featuring 48 physical cores of a socket and 528 GB of DDR5 DRAM. The maximum physical memory bandwidth, measured by mlc [26], is 266 GB/s.

#### 5.2 Evaluation of the QPS-Latency Curve

We illustrate the trade-off between QPS and latency for AVERSEARCH, FAISS-AVERSEARCH and iQAN in Figures 1 and 10, where QPS is plotted on the y-axis and latency is on the x-axis in reverse order using a logarithmic scale (base 2). For AverSearch, FAISS-AverSearch, and iQAN (i.e., systems with configurable parallelism settings), we adjust the number of intra- and inter-threads to generate the curve. In all configurations, the total thread count (i.e., intra-threads multiplied by inter-threads) is fixed at 48, matching the total available cores on our machine. We evaluated only the "1x48" configuration for systems without tunable parallelism settings. For systems with tunable settings, we evaluated configurations including "1x48", "2x24", "3x16", "4x12", "6x8", in Figures 1 and 10.

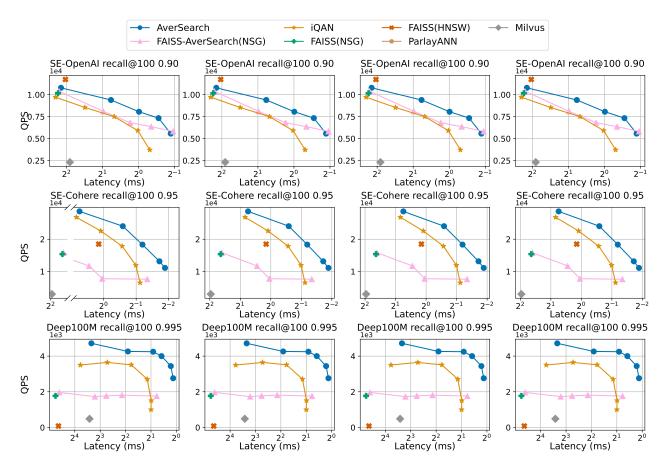


Figure 10. QPS-latency curve under various accuracy targets in different embeddings.

These figures demonstrate a trade-off between achieving low latency and high throughput, as both AVERSEARCH and iQAN experience a decrease in throughput when increasing intra-query parallelism. However, as evident from the figures, AVERSEARCH consistently achieves higher throughput and lower latency than iQAN under the same parallelism settings.

For instance, AverSearch's throughput is 1.23-2.38× higher than iQAN on the DEEP100M and SIFT100M datasets and, **simultaneously**, 1.07 to 2.35 times lower latency when comparing the same parallelism setting with each other. This performance gap persists in recent datasets featuring larger dimensional embeddings, such as the Wiki100M dataset, where AverSearch achieves up to 2.07 times higher throughput. Moreover, if we compare the QPS achieved at a comparable latency level (less than 20% diff), AverSearch can achieve 2.17× (SE-OpenAI) to 8.93× (DEEP100M) higher throughput than iQAN.

The evaluation of FAISS-AVERSEARCH and FAISS utilize the FAISS-constructed NSG index, which differs slightly from the original NSG index adopted in our experiment for AVERSEARCH and iQAN, as some FAISS-specific parameters are included in the FAISS-constructed index. To enable a fair comparison, we utilize the same NSG parameters to construct

both types of NSG index. FAISS-AVERSEARCH demonstrates satisfactory scalability as the number of intra-threads increases, indicating that AVERSEARCH could lower latency without compromising QPS.

We also evaluated the search implementation provided by ParlayANN [40] on Wiki100M and Wiki100M-2, for which the indices are built using it. Since ParlayANN does not provide intra-query parallelism optimization, it demonstrates comparable performance to AVERSEARCH on the "1х48" configuration.

The superior performance of AVERSEARCH is attributed to its fully asynchronous architecture, which significantly reduces synchronization overhead by eliminating the use of any barriers, thereby also reducing latency. Additionally, due to its high resource utilization under settings with intense intra-concurrency, AVERSEARCH typically achieves lower minimum average latency than iQAN. Across various datasets, the minimum average latency reduction ranges from 1.54× (SE-OpenAI) to 1.96× (DEEP100M), highlighting AVERSEARCH's efficiency in real-world scenarios.

#### 5.3 Performance Analysis

As discussed in §3.2, enhancing throughput hinges on improving PMB and reducing the RR. Table 1 presents these metrics measured across various datasets with the recall set to the highest level, 0.995. For instance, AVERSEARCH achieves 1.68× higher PMB and a 74% reduction on RR when compared to iQAN on the Wiki100M dataset. According to the formula outlined in §3.2, this leads to an approximately "149 \* (1-0.08): 89 \* (1-0.31) = 2.2×" higher Effective Memory Bandwidth (EMB), which results in a 1.78× higher throughput according to our evaluation.

In summary, AverSearch consistently achieves a higher PMB (1.35× - 2.32×) than iQAN. This improvement is primarily due to the asynchronous design of the distance calculator threads in AverSearch, which do not need to wait for subqueue maintainers. However, despite this no-wait strategy, the PMB achieved by AverSearch does not always reach the maximum physical bandwidth of the machine, especially for datasets with shorter vectors like SIFT100M and DEEP100M. This limitation often arises because the queue runs out of vertices, preventing further speculative execution. This challenge is largely due to the static assignment of thread roles, showing that while AverSearch offers more flexibility than iQAN, it still faces hurdles in balancing workloads between sub-queue maintainers and distance calculators.

Moreover, there is a notable reduction in RR for the Wiki100M dataset, largely due to our dynamic work dispatching policy. However, the RR for AverSearch is comparable to that of iQAN on the two Simple English datasets, mainly attributed to the smaller sizes of these datasets. This comparison underscores the enhanced effectiveness of AverSearch with larger datasets.

Dataset	PMB		RR	
	Ours	iQAN	Ours	iQAN
Sift100M (128D)	30 (2.18×)	13	0.06	0.22
Deep100M (96D)	32 (2.32×)	13	0.09	0.30
SE-Cohere (768D)	190 (1.91×)	99	0.32	0.36
SE-OpenAI (1536D)	242 (1.78×)	135	0.38	0.36
Wiki100M (768D)	149 (1.68×)	89	0.08	0.31
Wiki100M-2 (1024D)	140 (1.35×)	103	0.34	0.64

**Table 1.** Physical Memory Bandwidth (PMB) and Redundancy Ratio (RR) of AVERSEARCH (ours) and iQAN across various datasets, targeting a recall@100 of 0.995.

## 5.4 AVERSEARCH and Vector Databases

Besides iQAN, we also evaluated AVERSEARCH against leading vector databases such as Milvus [22, 54] and FAISS [15]. For Milvus, we utilize Hierarchical Navigable Small World (HNSW) [39] as the indexing method, as it is the most matured graph-based index available in the system. For FAISS,

we evaluate both HNSW and NSG. We carefully selected the searching parameters to ensure that the L parameter during searches in both Milvus and FAISS matched our system's (actually slightly smaller than our system to ensure a fair comparison), hence achieving comparable levels of recall.

As indicated in Figure 1 and 10, vector databases achieve lower throughput compared to iQAN and AVERSEARCH. Milvus and FAISS cannot process the large Wiki100M datasets due to OOM. However, it is important to note that this is not a head-to-head comparison because fully-fledged vector databases often include additional functionalities that might affect their performance. Moreover, since both Milvus and FAISS support only inter-query parallelism, we assessed their performance using a "1x48" configuration. The results show that AVERSEARCH achieves 1.48× to 9.82× lower latency and typically also a higher throughput (up to 5.68×) compared to these vector databases at "24x2" for high recall. Therefore, our system offers a compelling alternative that achieves significantly lower latency.

#### 5.5 AverSearch and Quantization-based Methods

To assess the significance of accelerating search on the graph-based index, we also evaluated Product Quantization method provided by FAISS [15]. We test the FlatPQ index which compress the database with product quantization codes. Since the FlatPQ index is unable to exceed the accuracy of 0.90 on small datasets, and 0.70 on larger ones, we only test it on the two Simple English datasets with a recall target of 0.90. The 1536-dimensional SE-OpenAI vectors were encoded into 512-length 8-bit codes, and the 768-dimensional SE-Cohere vectors into 256-length 8-bit codes. FlatPQ index achieves 639.55 and 691.55 QPS on SE-OpenAI and SE-Cohere respectively, which is 0.59 and 0.23 times of AVERSEARCH. Although the "curse of dimensionality" limits the accuracy of quantization methods, our system's acceleration can still speed up hybrid methods of quantization- and graph-based methods.

#### 5.6 Breakdown Analysis

To further validate the optimization sources in AVERSEARCH, we present a performance breakdown in Figure 11. In this analysis, the "async" refers to the basic implementation of AVERSEARCH's fully asynchronous architecture, but without the dynamic work-stealing balance discussed in §4.1. As we can see from the figure, "async" is already 1.38× to 2.27× faster than iQAN on smaller datasets like DEEP100M. However, this basic setup may encounter severe load imbalances. Thus, by incorporating the work-stealing technique, we observe significant improvements, with throughput increases ranging from 1.94× to 2.20× on Wiki100M (indicated as "+work-stealing").

Additionally, our analysis reveals that for larger graphs with high-dimensional vectors, such as Wiki100M, it is beneficial to partially inline the distance computation within

the sub-queue maintainers' process. This "+inline" adjustment results in a modest increase in QPS, due to increased PMB. However, this optimization does not translate well to smaller datasets like DEEP100M. Consequently, we have implemented this feature as an optional setting.

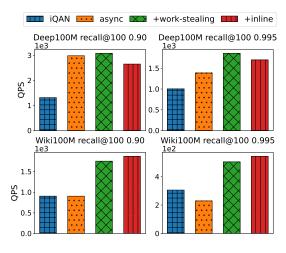


Figure 11. Breakdown Analysis at "24x2".

#### 6 Related Work

In this paper, we focus on enhancing intra-query parallelism for CPU-based graph ANNS indices. Besides the related works discussed in §2, the field has also seen various orthogonal advancements that complement our approach. For instance, Product Quantization (PQ) techniques [2, 20, 30, 31, 55, 56, 59], have been widely adopted to achieve dimensionality reduction. These techniques can be seamlessly integrated with our approach as a preprocessing step, allowing for a more flexible balance between performance and accuracy.

Moreover, many efforts have been made to offload part or the entirety of the distance calculation process to faster computing accelerators, such as GPUs [21, 32, 45, 63, 67] and FPGAs [14, 66]. However, these solutions often require the dataset to fit within the limited memory capacity of the accelerators, which is prohibitively costly for managing large-scale datasets. On the other end of the spectrum, some researchers have turned to high-speed SSDs [34] and the emerging Compute Express Link (CXL) memory [27] to handle larger datasets. These approaches usually retain an in-memory graph index search component, which could benefit from the enhancements presented in our method.

## 7 Conclusion

This paper presents AVERSEARCH, an innovative parallel graph-based ANNS framework with a fully asynchronous architecture. It effectively eliminates unnecessary synchronization (improving PMB) and reduces redundant vertex processing (decreasing RR). These advancements allow AVERSEARCH

to successfully balance the competing demands of throughput, accuracy, and latency. Extensive evaluations across multiple datasets show that AVERSEARCH achieves up to 8.9× higher throughput at the same level of latency, and reduces minimum latency by up to 1.9× compared to leading systems.

## References

- [1] 2012. Deep1B. https://disk.yandex.ru/d/11eDCm7Dsn9GA
- [2] Ameer MS Abdelhadi, Christos-Savvas Bouganis, and George A Constantinides. 2019. Accelerated approximate nearest neighbors search through hierarchical product quantization. In 2019 International Conference on Field-Programmable Technology (ICFPT). IEEE, 90–98.
- [3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. Advances in neural information processing systems 28 (2015).
- [4] Martin Aumueller, Erik Bernhardsson, and Alec Faitfull. 2023. ANN-Benchmarks. https://ann-benchmarks.com/index.html
- [5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems 87 (2020), 101374.
- [6] 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.
- [7] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highlyefficient billion-scale approximate nearest neighbor search. <u>arXiv</u> preprint arXiv:2111.08566 (2021).
- [8] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. 2024. MagicPIG: LSH Sampling for Efficient LLM Generation. arXiv preprint arXiv:2410.16179 (2024).
- [9] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In 2016 IEEE 32nd International Conference on Data Engineering (ICDE). IEEE, 409–420.
- [10] cohere.ai. 2022. Cohere/wikipedia-22-12. https://huggingface.co/dat asets/Cohere/wikipedia-22-12
- [11] cohere.ai. 2022. Develop, test, and experiment with the industry's first multilingual text understanding model that supports 100+ languages. https://cohere.com/blog/multilingual
- [12] cohere.ai. 2023. Cohere/wikipedia-2023-11-embed-multilingual-v3. https://huggingface.co/datasets/Cohere/wikipedia-2023-11-embed-multilingual-v3
- [13] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. <u>IEEE computational</u> science and engineering 5, 1 (1998), 46–55.
- [14] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 217–226.
- [15] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]
- [16] facebookresearch. 2022. facebookresearch/faiss. https://github.com/facebookresearch/faiss
- [17] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. <u>arXiv preprint</u> arXiv:1609.07228 (2016).

- [18] Cong Fu, Changxu Wang, and Deng Cai. 2021. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. <u>IEEE Transactions on Pattern</u> Analysis and Machine Intelligence 44, 8 (2021), 4139–4150.
- [19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreadingout graph. arXiv preprint arXiv:1707.00143 (2017).
- [20] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization for approximate nearest neighbor search. In <u>Proceedings of the IEEE conference on computer vision and pattern</u> recognition. 2946–2953.
- [21] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. Ggnn: Graph-based gpu nearest neighbor search. <u>IEEE</u> <u>Transactions on Big Data 9, 1 (2022), 267–279.</u>
- [22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: a cloud native vector database management system. <a href="Proceedings of the VLDB Endowment 15">Proceedings of the VLDB Endowment 15</a>, 12 (2022), 3548–3561.
- [23] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In <u>Twenty-Second International Joint Conference on</u> Artificial Intelligence.
- [24] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. Proceedings of the VLDB Endowment 8, 9 (2015), 950–961.
- [25] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In <u>Proceedings of the IEEE Conference on</u> Computer Vision and Pattern Recognition. 5713–5722.
- [26] Intel. 2024. Intel Memory Latency Checker v3.11. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html
- [27] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. {CXL-ANNS}:{Software-Hardware} Collaborative Memory Disaggregation and Computation for {Billion-Scale} Approximate Nearest Neighbor Search. In 2023 USENIX Annual Technical Conference (USENIX ATC 23). 585–600.
- [28] Yahoo Japan. 2024. Neighborhood Graph and Tree for Indexing Highdimensional Data. https://github.com/yahoojapan/NGT
- [29] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. Advances in Neural Information Processing Systems 32 (2019).
- [30] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. <u>IEEE transactions on pattern</u> analysis and machine intelligence 33, 1 (2010), 117–128.
- [31] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 861–864.
- [32] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. <u>IEEE Transactions on Big Data</u> 7, 3 (2019), 535–547
- [33] Hervé Jégou Laurent Amsaleg. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/
- [34] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. 2022. VStore: in-storage graph based vector search accelerator. In Proceedings of the 59th ACM/IEEE Design Automation Conference. 997–1002.
- [35] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. 2024. Retrievalattention: Accelerating long-context llm inference via vector retrieval. arXiv preprint arXiv:2409.10516 (2024).

- [36] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: approximate nearest neighbor search via virtual hypersphere partitioning. Proceedings of the VLDB Endowment 13, 9 (2020), 1443–1455.
- [37] Kejing Lu, Chuan Xiao, and Yoshiharu Ishikawa. 2024. Probabilistic Routing for Graph-Based Approximate Nearest Neighbor Search. <u>arXiv</u> preprint arXiv:2402.11354 (2024).
- [38] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. <u>Information Systems</u> 45 (2014), 61– 68
- [39] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. <u>IEEE transactions on pattern analysis and machine</u> intelligence 42, 4 (2018), 824–836.
- [40] Magdalen Dobson Manohar, Zheqi Shen, Guy 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 Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 270–285.
- [41] Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. <u>VISAPP (1)</u> 2, 331-340 (2009), 2.
- [42] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. <u>Pattern Recognition</u> 96 (2019), 106970.
- [43] NeurIPS'2023. 2023. NeurIPS'23 Competition Track: Big-ANN. https://big-ann-benchmarks.com/neurips23.html
- [44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In <u>Proceedings of the</u> <u>twenty-fourth ACM symposium on operating systems principles</u>. 456– 471.
- [45] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2023. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. <u>arXiv preprint</u> arXiv:2308.15136 (2023).
- [46] OpenAI. 2023. OpenAI, all-MiniLM-L6-v2, GTE-small embeddings for Wikipedia Simple English. https://huggingface.co/datasets/Supaba se/wikipedia-en-embeddings
- [47] OpenAI. 2023. OpenAI embeddings for Wikipedia Simple English. https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings
- [48] OpenAI. 2024. Introducing OpenAI o1. https://openai.com/o1/
- [49] OpenAI. 2024. New embedding models and API updates. https://openai.com/blog/new-embedding-models-and-api-updates
- [50] 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.
- [51] Jie Ren, Minjia Zhang, and Dong Li. 2020. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. <u>Advances</u> in Neural Information Processing Systems 33 (2020), 10672–10684.
- [52] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In 2008 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, 1–8.
- [53] 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. <u>Proceedings of the VLDB Endowment</u> 6, 14 (2013), 1930–1941.
- [54] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al.

- , .
- 2021. Milvus: A Purpose-Built Vector Data Management System. In Proceedings of the 2021 International Conference on Management of Data. 2614–2627.
- [55] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. Proceedings of the VLDB Endowment 13, 13 (2020), 3603–3616.
- [56] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. Proceedings of the VLDB Endowment 13, 12 (2020), 3152–3165.
- [57] Wikipedia. 2024. Wikipedia (simple English). https://simple.wikipedia.org/wiki/Main\_Page
- [58] WPJiang. 2024. HWTL\_SDU-ANNS. https://github.com/WPJiang/H WTL SDU-ANNS
- [59] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. 2017. Multiscale quantization for fast similarity search. <u>Advances in neural</u> information processing systems 30 (2017).
- [60] Ming Yang, Yuzheng Cai, and Weiguo Zheng. [n.d.]. CSPG: Crossing Sparse Proximity Graphs for Approximate Nearest Neighbor Search. In <u>The Thirty-eighth Annual Conference on Neural Information</u> Processing Systems.
- [61] Mingyu Yang, Wentao Li, and Wei Wang. 2024. Fast High-dimensional Approximate Nearest Neighbor Search with Efficient Index Time and Space. arXiv preprint arXiv:2411.06158 (2024).

- [62] Rahul Yesantharao. 2022. <u>Parallel Batch-Dynamic kd-trees</u>. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [63] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction. In 2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, 552–564.
- [64] Zhenrui Yue, Honglei Zhuang, Aijun Bai, Kai Hui, Rolf Jagerman, Hansi Zeng, Zhen Qin, Dong Wang, Xuanhui Wang, and Michael Bendersky. 2024. Inference Scaling for Long-Context Retrieval Augmented Generation. arXiv preprint arXiv:2410.04343 (2024).
- [65] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2024. Pqcache: Product quantization-based kvcache for long context llm inference. <u>arXiv</u> preprint arXiv:2407.12820 (2024).
- [66] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2018. Efficient large-scale approximate nearest neighbor search on OpenCL FPGA. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 4924–4932.
- [67] Weijie Zhao, Shulong Tan, and Ping Li. 2020. Song: Approximate nearest neighbor search on gpu. In <u>2020 IEEE 36th International</u> Conference on Data Engineering (ICDE). IEEE, <u>1033-1044</u>.
- [68] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. <u>Proceedings of the VLDB</u> <u>Endowment</u> 16, 8 (2023), 1979–1991.