iQAN: Fast and Accurate Vector Search with Efficient Intra-Query Parallelism on Multi-Core Architectures

Zhen Peng
College of William & Mary
Williamsburg, VA USA
zpeng01@wm.edu

Minjia Zhang
Microsoft AI and Research
Bellevue, WA USA
minjiaz@microsoft.com

Kai Li
Kent State University
Kent, OH USA
kli17@kent.edu

Ruoming Jin
Kent State University
Kent, OH USA
rjin1@kent.edu

Bin Ren
College of William & Mary
Williamsburg, VA USA
bren@cs.wm.edu

Abstract

Vector search has drawn a rapid increase of interest in the research community due to its application in novel AI applications. Maximizing its performance is essential for many tasks but remains preliminary understood. In this work, we investigate the root causes of the scalability bottleneck of using intra-query parallelism to speedup the state-of-the-art graph-based vector search systems on multi-core architectures. Our in-depth analysis reveals several scalability challenges from both system and algorithm perspectives. Based on the insights, we propose *iQAN*, a parallel search algorithm with a set of optimizations that boost convergence, avoid redundant computations, and mitigate synchronization overhead. Our evaluation results on a wide range of real-world datasets show that *iQAN* achieves up to 37.7× and 76.6× lower latency than state-of-the-art sequential baselines on datasets ranging from a million to a hundred million datasets. We also show that *iQAN* achieves outstanding scalability as the graph size or the accuracy target increases, allowing it to outperform the state-of-the-art baseline on two billion-scale datasets by up to 16.0× with up to 64 cores.

CCS Concepts: • **Information systems → Nearest-neighbor search**; *Searching with auxiliary databases.*

Keywords: approximate nearest neighbor search, graph-based, vector search, intra-query parallelism

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577527

1 Introduction

Nearest neighbor search (NNS) recently gained popularity due to its core role in building semantic-based search systems for unstructured data such as images, texts, and video using neural embedding models. In semantic-based vector search, N unstructured data are encoded into embedding vectors via neural networks in high dimensional space $\mathbb{R}^d$, where d often ranges from 100 to 1000 and N ranges from millions to billions. The search finds the K nearest embeddings for a given query based on the distance metric between vectors. Semantic-based search enables novel applications and has been adopted in many real-world applications. For example, major e-commerce players (e.g., Amazon [46]) build semantic search engines that embed both product catalog and the search query and then recommend products whose embeddings are closest to the embedded search query; Youtube [12] embeds videos to vectors for video recommendation; Web-scale search engines embed text (e.g., word2vec [43], doc2vec [32]) and images (e.g., VGG [54]) for text/image retrieval [11, 56], and many more.

Since the search often occurs on online interactive applications for every query, significant challenges have been raised to reduce the *latency* of NNS. Numerous efforts aim to reduce the search latency while achieving high accuracy by designing various approximate nearest neighbor search (ANNS) algorithms, including hashing-based methods[2, 3, 13, 25], quantization-based methods [21, 27, 66, 67], tree-based methods [10, 53, 64], and graph-based methods [19, 37, 68]. Among them, the similarity graph-based algorithms have emerged as a remarkably effective class of methods for high-dimensional ANNS, outperforming other approaches on a wide range of datasets to achieve the best accuracy-vs-latency [4, 19, 38]. These graph-based methods have also been integrated with many large-scale production systems [11, 19, 38], where optimizations for fast search and high accuracy have a clear, practical impact because production systems have stringent latency and high accuracy requirements: delayed or inaccurate responses directly hurt user satisfaction and affect revenue [17].

Despite their promising results, graph-based methods still have challenges that limit their use in real-world scenarios. In particular, as the data size grows, achieving both low latency and high accuracy simultaneously becomes increasingly challenging. Current solutions often resort to inter-query parallelism by dispatching queries across multiple processors or nodes to be processed simultaneously [9, 19]. This approach scales from a throughput perspective but does not help reduce query latency because each query still needs to perform the same amount of vector computations to find the nearest neighbors. Another natural idea to reduce latency is to exploit intra-query parallelism on individual nodes with multi-core processors. For example, one may parallelize the node expansion in each iteration step of the sequential search algorithm, hoping that multiple worker threads can check the closeness of multiple neighbors in parallel while performing the same computations on each step as the sequential algorithm. Surprisingly, this solution performs quite poorly and may even perform much worse than a well-tuned sequential algorithm. To date, however, few studies have examined why intra-query parallel search has difficulties in achieving speedups for graph-based ANNS on multi-core architectures.

In this paper, we exploit intra-query parallelism for graph-based ANNS and ask the questions: how to achieve low search latency and high accuracy for graph-based ANNS algorithms on multi-core architectures? Especially, the algorithm should have a clear advantage over the state-of-the-art ANNS implementations. We start by presenting several studies that reveal the root causes of the poor scalability from directly parallelizing the existing sequential graph traversal strategy. Based on our studies, we present *iQAN*, a parallel graph search algorithm that offers both low latency and high accuracy simultaneously while carrying excellent scalability as the graph size or accuracy target increases. *iQAN* introduces a new parallelism scheme called *path-wise parallelism* that allows reducing the iteration depths significantly. This optimization avoids long sequential dependencies during the search, but it raises a challenge to the search efficiency: it introduces a considerable amount of redundant computations, leading to wasted compute and memory bandwidth consumption. To mitigate redundant computations, *iQAN* introduces a *staged expansion* scheme that only performs path-wise parallelism at where they are most effective when searching. Finally, the scalability of parallel graph search is limited by the frequent global synchronizations needed for greedy routing. To reduce the synchronization overhead, *iQAN* introduces a *redundancy-aware synchronization* strategy that allows multiple worker threads to perform searches concurrently while avoiding redundant computations and a large number of synchronizations, without losing accuracy. In summary, this paper makes the following contributions:

- Performing in-depth studies to reveal the root causes of the poor scalability of state-of-the-art vector search algorithms on multi-core architectures (Section 4).
- Introducing intra-query parallelism optimizations, i.e., path-wise parallelism, staged expansion, and redundancy-aware synchronization to accelerate the search (Section 5).
- Evaluating *iQAN* and showing order of magnitude latency improvement against state-of-the-art solutions (Section 6).

We implement *iQAN* in C++ on Linux Our evaluations results on a wide range of datasets show that *iQAN* achieves up to 37.7× and 76.6× lower latency than state-of-the-art solutions (NSG [19], HNSW [38]) for a wide range of accuracy targets and datasets ranging from a million to a hundred million data points, from ∼100 dimensions to ∼1000 dimensions embedding vectors, on different multi-core architectures. We also observe that *iQAN* achieves outstanding scalability as the graph size or the accuracy target increases, allowing it to outperform the state-of-the-art solution on two billion-scale datasets by up to 16.0× with up to 64 cores. Finally, we present a performance breakdown and thorough analysis to study the performance impact of its key optimizations and comparison with alternative methods.

2 Background and Related Work

2.1 ANN Search Optimizations

The literature on nearest neighbor search is vast, and hence, we focus our attention on the most relevant works here. There has been a lot of work on building effective ANN indices to accelerate the search process. Earlier works focus on space partitioning-based methods. For example, Tree-based methods (e.g., KD-tree [53] and R* tree [10]) hierarchically split the data space into lots of regions that correspond to the leaves of a tree structure and only search a limited number of promising regions. However, the complexity of these methods becomes no more efficient than brute-force search as the dimension becomes large (e.g., >16) [33]. Prior works also have spent extensive efforts on locality-sensitive hashing-based methods [2, 3, 13, 25], which map data points into multiple buckets with a certain hash function such that the collision probability of nearby points is higher than the probability of others. These methods have solid theoretical foundations. LSH and its variations are often designed for large sparse vectors with hundreds of thousands of dimensions. In practice, LSH-based methods have been outperformed by other methods, such as graph-based approaches, by a large margin on large-scale datasets [4, 19, 38].

More recently, Malkov and Yashunin found graphs that satisfy the small-world property exhibit excellent navigability in finding nearest neighbors. They introduce the Hierarchical Navigable Small World (HNSW) [38], which builds a hierarchical k-NN graph with additional long-range links that help create the small-world property. For each query, it then performs a walk, which eventually converges to the nearest neighbor in logarithmic complexity. Subsequently, Fu et al. proposed NSG, which approximates Monotonic Relative Neighbor Graph (MRNG) [19] that also involves long-ranged links for enhancing connectivity. Both HNSW and NSG have been proved to outperform prior methods by checking much fewer data points to achieve the same recall [4, 6, 16, 19, 28, 34, 64, 66].

2.2 Parallel Graph Processing

Parallel graph algorithms. There are numerous efforts that aim to parallelize generic search schemes on graphs (e.g., BFS [52], DFS [44], random walk [58], beam search [41], and bucketing [55, 72]). However, many of these algorithms were designed without considering having a vector associated with each vertex and a target of achieving high recall under a stringent latency constraint. In contrast, we analyze the search efficiency challenges of ANN and propose optimizations to handle them to allow vector-based similarity search to scale on modern multi-core architectures. Among different algorithms, the most related work to ours is perhaps Δ-stepping [72], which stages the expansion of nodes in order to avoid redundant expansions. We have applied Δ-stepping to vector search and will provide a more detailed discussion in Section 5.5 and comparison in Section 6.

Parallel graph frameworks. There has been many graph engines and frameworks developed in the past decade. Some of them are shared-memory, focusing on processing in-memory datasets within a computation node [47, 60], e.g., Galois [45], Ligra [52], Polymer [71], GraphGrind [57], GraphIt [72], Graptor [62], and GraphBLAS [5]. Some are distributed systems [49], e.g., Pregel [36], GraphLab [35], PowerGraph [22], and Gluon [14]. Some efforts focus on out-of-core designs (e.g., GraphChi [31] and X-Stream [50]) and process large graphs with disk support. Many graph frameworks are also on GPUs [42, 63], such as CuSha [30], Gunrock [65], GraphReduce [51], Graphie [23], Multigraph [24], GraphBLAST [69], and Ascetic [59]. These graph systems are either based on a vertex-centric model [36, 70] or its variants (e.g., edge-centric [50]). Many of these parallel graph frameworks are designed primarily for generic parallel graph analytics instead of vector-based similarity search. Enabling ANN search in these frameworks, which have matured compilers and optimization technologies, is possible but requires addressing non-trivial portability challenges. First, vector-based similarity search uses special data formats for the index (e.g., hierarchical graphs or hybrid KD-tree plus graph structuresl [39]), which is not supported by existing graph frameworks. Second, vector-based similarity search requires well-designed parallelism and synchronization mechanisms (even targeted for heterogeneous memory [26, 48]), while the general optimization provided by existing graph frameworks (e.g., delta-stepping) cannot provide enough performance benefits. Therefore, porting those changes, to existing frameworks, beyond the search algorithm itself, requires changes across the entire system stack.

3 Preliminaries

Similarity graph. A similarity graph is a directed graph $G = (V, E)$, where each vertex $v_i \in V$ corresponds to one of the vectors v_i in a set of N d-dimensional embedding vectors $v = \{v_1, ..., v_N\}$. In practice, embedding vectors are generated by entities in a problem domain (e.g., a video or image in a recommendation system), which carry semantic meanings. The vertices v_i and v_j are connected by an edge if v_j belongs to the set of M relative nearest neighbors of v_i, determined by the similarity graph construction algorithm (e.g., NSG [19]). There are no self-edge or duplicate edges in the graph.

Algorithm 1: Best-First Search (*BFiS*)

```
Input: graph $G$, starting point $P$, query $Q$, queue capacity $L$
Output: $K$ nearest neighbors of $Q$
1  priority queue $S \leftarrow \emptyset$ /* sorted based on distance */
2  index $i \leftarrow 0$
3  compute $dist(P, Q)$
4  add $P$ into $S$
5  while $S$ has unchecked vertices do /* stop condition */
6      $i \leftarrow$ the index of the 1st unchecked vertex in $S$
7      mark $v_i$ as checked
8      foreach neighbor $u$ of $v_i$ in $G$ do
9          if $u$ is not visited then
10             mark $u$ as visited
11             compute $dist(u, Q)$
12             add $u$ into $S$          /* $u$ is unchecked */
13     if $S$.size() > $L$ then $S$.resize($L$)
14 return the first $K$ vertices in $S$
```

Top-K search. The search in a similarity graph is performed via the Best-First Search (*BFiS*) [19], which aims to search only a small subset of the graph nodes to find the top-K nearest neighbors based on their closeness (e.g., Euclidean distance) to the query. BFiS starts at a chosen (e.g., medoid or random) point of the graph and greedily traverses the graph's edges by getting closer to the nearest neighbors at each step until it converges to a local optimum (i.e., found top-K near neighbors). Algorithm 1 shows its basic idea. The search algorithm maintains a priority queue of size L with graph nodes ($L \geq K$), indicating which neighbors should be visited by the search process. In the beginning, all nodes are initially in an unchecked state. During graph traversal, the algorithm first selects the closest unchecked node v_i from the queue, called an active node (Line 6), and performs a node expansion. A node expansion computes the pair-wise distance of all neighbors of v_i to the query (Line 8-12). After the node expansion, the search inserts promising neighbors into the priority queue as new unchecked candidates for future expansion. The candidates in the priority queue are sorted according to their distance to the query, so less promising candidates will be popped out as new ones are added (Line 13). The search iteratively expands unchecked nodes based on their closeness (e.g., Euclidean distance) to the query. The search ***converges*** when the priority queue has at least K candidates and there are no unchecked nodes in it, indicating that it has reached a local optimum.

Metric. In practice, finding the exact top-K can be very time-consuming. As a result, the search process only examines a subset of vectors in the similarity graph, leading to an *accuracy-vs-latency* trade-off. The accuracy is often measured by the *recall*, which is the fraction of true nearest neighbors (R) in retrieved top-K candidates (R'), defined as follows [18]:

$$Recall(R') = \frac{|R' \cap R|}{|R'|} = \frac{|R' \cap R|}{K} \tag{1}$$

A high *recall* is desired as low accurate results degrade user satisfaction. On the other hand, the latency measures the time spent to find the top-K nearest neighbors. Low latency is crucial, especially to enable ANN search for online interactive applications.

Given the preliminaries, we now define the exact problem we are tackling in this paper:

Problem definition. Considering a similarity graph and a multi-core architecture with P processors, our goal is to design a parallel search algorithm such that the search latency to reach a given recall target is minimized.

4 Challenges in Graph-based ANN Search

We first discuss a straightforward parallelism implementation and analyze its cost on multi-core CPUs. The analysis later guides the design in the design section.

Edge-wise parallelism (EP). Given that the pair-wise distance computations (Line 8-12 in Algorithm 1) have no dependency on each other within an iteration, it is a natural idea to parallelize the neighbor expansion step by splitting the distance computation across multiple threads. We denote this scheme as *edge-wise parallelism*. Edge-wise parallelism allows neighbor expansion to run in parallel while performing the same computations on each neighbor as the sequential algorithm. Another benefit of edge-wise parallelism is that it returns the same result on each execution regardless of how many threads are used. Despite its benefits in simplicity, this natural idea cannot lead to good speedups. In fact, due to the well-tuned sequential baseline, the edge-wise parallelism often achieves sub-optimal performance. Fig. 1 shows that to reach the recall target of 0.999 on DEEP100M dataset (the detailed setup can be found in Section 6), the multi-threading search with edge-wise parallelism performs poorly, i.e., no speedups from 1 to up to 64 threads. What causes the poor scalability of graph-based search on multi-core architecture?

Cause 1: Edge-wise parallelism leads to a high synchronization cost. One major challenge in scaling edge-wise parallelism is that a large number of node expansions need to be performed to achieve high accuracy on large graphs, resulting in hundreds or sometimes even thousands of expansion rounds. Since each round requires at least one global synchronization to maintain the order of all candidates according to their distances to the queue point, this frequent global synchronization adds significant synchronization overhead to the search process. Fig. 2 shows that as we increase the number of threads, the synchronization overhead accounts for more than 50% of the total search time, becoming a dominating factor in the overall search latency. In principle, it is possible to mitigate this synchronization overhead by adopting a concurrent priority queue or lock-free algorithms during insertions. However, we found that there are additional challenges that severely limit the parallel search speed, as described below.

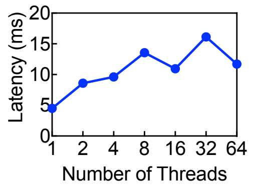


Figure 1. EP's latency on Deep100M.

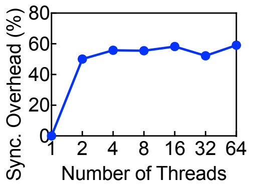


Figure 2. EP adds high sync. overhead.

Cause 2: Edge-wise parallelism leads to low computational intensity, making the search process hard to fully utilize memory bandwidth. We use Intel Processor Counter Monitor [15] to measure the memory bandwidth utilization under various datasets and graphs. The data movement mostly comes from loading vectors during node expansion. Table 1 shows that execution with a single thread use only less than 3.4% of the peak hardware memory bandwidth consumption (~80 GB/s) on an Intel Xeon Phi processor, indicating that utilizing more bandwidth with multi-threading should lead to higher performance. However, with 32-way edge-wise parallelism, the memory bandwidth utilization only modestly increases to up to 4.2%. Some of the executions (e.g., SIFT1M) even observe decreased bandwidth consumption, implying that the edge-wise parallelism has a low computational intensity that makes it challenging to use all of the raw bandwidth available. The reason that edge-wise parallelism has low compute intensity lies in two aspects: (1) unlike matrix multiplication, the point-wise Euclidean distance computation is an operator with low compute intensity, and (2) the number of neighbors to be expanded in one step is limited, given that similarity graphs naturally have low out-degree to avoid the *out-degree explosion problem* [19].

Table 1. Memory bandwidth (GB/s) measurement.

Benchmarks	SIFT1M	GIST1M	DEEP10M	SIFT100M	DEEP100M
single thd.	2.1	2.7	1.6	1.2	0.8
edge-wise 64 thds.	2.0	3.4	2.0	2.7	1.6

Cause 3: Edge-wise parallelism still requires many iterations to converge, resulting in long sequential dependencies between steps. In Algorithm 1, the search performs a series of sequential iterations (Line 5-13), where each iteration performs a node expansion. Which node to expand depends on the priority queue updated by the previous steps. Moreover, the number of iterations depends on the recall target and the graph size. For example, Fig. 3 shows that as the recall target increases, the number of iterations to find the top-100 nearest neighbors on a hundred million scale dataset DEEP100M grows dramatically as the recall target becomes

higher (e.g., a 34.6-time increase from 0.9 to 0.999 recall). Fig. 4 shows that as the dataset size increases, the number of iterations to find the results for recall target 0.999 also grows (e.g., 7.3 times from 1M-vector dataset to 100M-vector dataset). This long sequential dependency makes achieving low latency with high accuracy especially challenging.

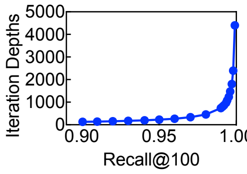


Figure 3. Iteration depths change along with recall.

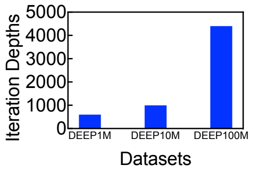


Figure 4. Iteration depths change along with data sizes.

5 Design of *iQAN*

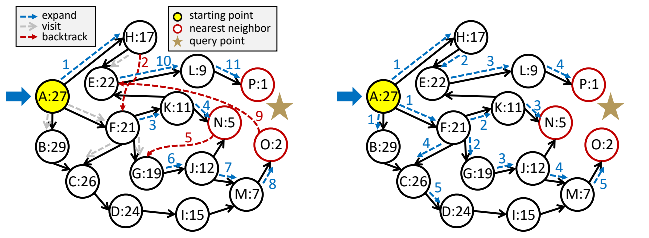
To address the aforementioned challenges, we introduce *iQAN*, a parallel search algorithm to accelerate graph-based ANNS on multi-core architectures.

5.1 Reduce Iteration Depth by Path-Wise Parallelism

In each search iteration, *BFiS* performs node expansion to the most promising unchecked candidate. In *iQAN*, we make a small modification to this process by relaxing the priority order and letting each thread expand a few more nodes (e.g., top W unchecked candidates) in every step as active nodes for expansion. We also relax the synchronization such that a global synchronization is only performed after a few expansion steps. We call this new way of expanding nodes *path-wise parallelism (PP)*. This small change in algorithm results in a significant reduction in iteration depths for queries, e.g., from a few thousands to tens in some cases.

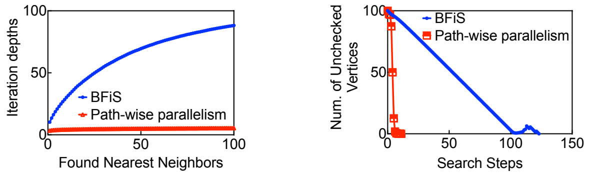
Why would this change reduce the iteration depth? The multi-node expansion and relaxed synchronizations are equivalent to letting each thread explore paths in a local region instead of a single node's neighbor list before doing a global synchronization. By doing so, it increases the likelihood of finding nearest neighbors in less number of iterations. Conceptually, this can be illustrated in Fig. 5a. It shows the search path of *BFiS* expands only one node in every search step marked as dashed arrows. After the 1st step, H is the active node. However, it cannot lead to a closer candidate, so it *backtracks* to the unchecked candidate F. The same kind of backtracking happens from N to G and O to E. Consequently, *BFiS* has a long search path to find all nearest neighbors. In contrast, Fig. 5b shows an example that by letting a thread expand top-3 promising candidates (e.g.,$W = 3$), after the 1st step, H and F are both active nodes; thus, the search paths to nearest neighbors start immediately without waiting for its turn as in *BFiS*. As a result, while *BFiS* has an iteration depth of 11, the path-wise parallelism only has 5.

We have conducted experimental studies to verify the effectiveness of this change. Fig. 6 shows the comparison results of iteration depths between *BFiS* and *PP* on dataset SIFT1M using 10K queries with a 0.90 recall target. We set W to 64. Overall, while *BFiS* takes 10.1, 69.4, and 88.1 steps



(a) *BFiS*'s long iteration depths. **(b)** PP's shorter iteration depths.

Figure 5. BFiS (11 steps) vs. path-wise parallelism (5 steps).



(a) Iteration depths to find the K-th nearest neighbor (x-axis). **(b)** The number of steps for a search to converge.

Figure 6. Path-wise parallelism converges to local optimum with a much smaller iteration steps than *BFiS*.

to find the top-1, top-50, and top-100 near neighbor, *PP* only takes 3.4, 5.0, and 5.4 steps on average, respectively, a significant reduction. From the unchecked node's perspective, Fig. 6b shows that *PP* also takes much fewer steps to converge to a local optimum (i.e., finish examining all the unchecked vertices) than *BFiS*.

We remark that the *BFiS* expansion process is similar to the tree expansion in the classical DFS/BFS. *BFiS* naturally introduces an expansion tree: the root node T_r of the tree is the starting vertex P in graph G; the children of a tree node Q_i (corresponding to a graph vertex v_i) are the unvisited neighbors of v_i. The expansion of *BFiS* bears many similarities to DFS, as each time, it will expand only one leaf node. Our path-wise parallelism is inspired by parallel expansion for DFS/BFS. However, different from DFS, which expands the one with the most depth, PP expands the W leaves simultaneously of the tree, which are W nearest neighbors of query Q among all the leaves of the *current* expansion tree. Thus, PP can potentially reduce the total number of iteration depths of *BFiS* by a factor of W times. As for T steps, PP can expand $T \times W$ graph nodes. We also note that the edge-wise parallelism becomes a special case of PP where $W = 1$ and $T = 1$, and both parallelization are the under Bulk Synchronous Parallel (BSP) model [61] though *BFiS* has rather limited parallelism to explore.

We also remark that in the path-wise parallelism, the visiting map is shared by all workers to indicate if a vertex has been visited. Since multiple threads may access the shared visiting map concurrently, locking or lock-free algorithms are required to ensure a vertex is visited only once. Fortunately, the ANNS algorithm remains correct even if a vertex is calculated multiple times because the local candidates are guaranteed to be merged back to the global priority queue. Therefore, multiple threads do not need to exclusively update the visiting map and a *loosely synchronized visiting map*

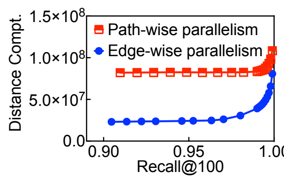


Figure 7. Aggregated distance computations of *BFiS* w/ EP and PP, where $W = 64$.

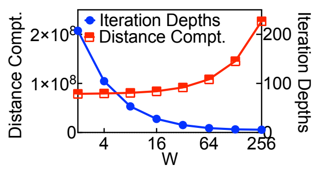
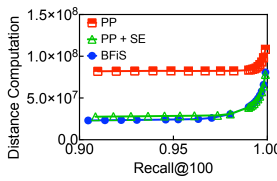
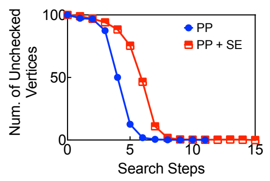


Figure 8. Dist. compt. increases as iter. depths decrease for PP increasing W.



(a) Dist. computation of *BFiS*, PP w/o and w/ staged expansion.



(b) Number of unchecked candidates after each search step.

Figure 9. Comparison between path-wise parallelism with and without staged expansion. $W = 64$.

can reduce communication overhead. We implement it using a bit vector for fast access.

5.2 Reduce Redundant Computation by Staged Expansion

Although reducing the iteration depth significantly, does it mean the search process will now get desired speedups on multi-core architectures? The answer is no. The path-wise parallelism reduces iteration depths but at the same time introduces a considerate amount of additional distance computations, especially when the number of parallel workers is large. This is because path-wise parallelism increases the likelihood for a thread to expand unpromising nodes that could have been avoided in edge-wise parallelism: some active vertices out of the expansion width W might not lead to the final nearest neighbors from a future perspective. Fig. 7 shows that to reach the same recall (0.9–0.999), the path-wise parallelism often needs to perform significantly more distance computations than *BFiS* (1.3–3.5 times). Moreover, we also observe that although the iteration depths continue to decrease by increasing the concurrent expansion width W, the number of distance computations inversely increases, as shown in Fig. 8. The huge amount of redundant computations adversely affects the search efficiency as many threads are loading vectors for unnecessary computations, wasting memory bandwidth and compute resources.

To mitigate it, we investigate the usefulness of path-wise parallelism at different search stages: at which stage does the path-wise parallelism reduce the iteration depths the most? We found that overall, in the beginning, since all candidates are far from the query, those early expanded candidates are likely to be discarded by closer ones that are visited later. In other words, candidates expanded and checked at an earlier stage have a high likelihood of becoming unnecessary from a future perspective. As the search moves forward towards the region that has the near neighbors, a larger expansion width that covers more search paths can effectively prevent the search from getting stuck at a local minimum.

Based on these observations, we propose a *staged expansion (SE)* scheme by gradually increasing the expansion width W and the number of workers every t steps during the search procedure. In practice, we set the starting value of W to 1 and the maximum value as the number of available hardware threads. Then for every t steps (e.g., $t = 1$) we double the value of W until W reaches its maximum. Fig. 9a shows the comparison results of path-wise parallelism without and with staged expansion. The staged expansion reduces the number of redundant distance computations significantly, leading to distance computations comparable to *BFiS*. On the other hand, staged expansion is able to preserve the benefits of path-wise parallelism in terms of obtaining reduced iteration depths, as shown in Fig. 9b. These results indicate that by performing path-wise parallelism at where they are most effective (i.e., the later phase of the search), the parallel search process can effectively converge with reduced iteration depths and very minimal addition of redundant computations among multiple workers.

5.3 Reduce Synchronization Overhead by Redundancy-Aware Synchronization

The remaining performance challenge in parallel search resides in the synchronization, as we still need to decide when to do synchronization. However, reducing the synchronization overhead for graph-based ANNS is non-trivial. Fig. 10 shows that as we skip synchronizations in between search iterations (i.e., increasing the interval between two synchronizations), the synchronization overhead (shown as the ratio to the total time) decreases significantly. However, decreasing synchronization increases distance computations, especially when the synchronization intervals become large. This is because as we increase the synchronization interval, it increases the likelihood that individual workers would search their local but unpromising areas without switching to newly identified promising regions found by other workers. As such, one cannot infinitely delay synchronization, and a small set but useful synchronizations are desired to achieve the overall high search efficiency without incurring too many redundant computations.

Finding such intervals turns out to be non-trivial since the relative distance of a query to its near neighbors changes all the time at different stages. It is also hard to find one fixed synchronization interval for all queries. To mitigate the synchronization overhead, *iQAN* performs *redundancy-aware synchronization (RAS)*, which allows workers to perform a search with low redundant computations by adding a minimal set of global synchronizations.

Measuring redundant expansion via update positions. We introduce a metric — *update positions* — to capture the redundancy during expansion. When a worker thread expands an unchecked candidate, its unchecked neighbors are then inserted into the worker's local queue, and we define

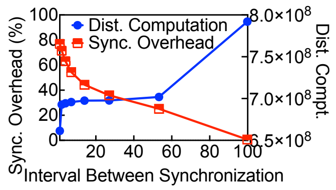


Figure 10. Sync. overhead and distance compt. as the sync. interval increases.

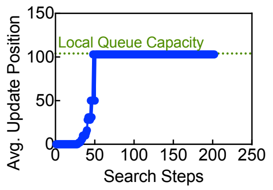


Figure 11. A query's average update positions during searching.

the update position as the *lowest (best)* position of all newly inserted candidates. Thus, *the average update position* (AUP) is the mean of all update positions of workers. Fig. 11 demonstrates how an example query's AUP changes during the search process without doing any global synchronizations. We observe that the AUP increases gradually to be equal to the local queue capacity and remains flat to the end. When the AUP is close to the queue capacity, it indicates that a majority of workers are searching areas that cannot find promising candidates to update their local results. Therefore, a high AUP indicates that most workers are doing redundant computations, and it would benefit from a global synchronization such that all workers can focus on searching for more promising areas that have a higher probability of including closer near neighbors. Due to the space limitation, the detailed algorithm of *iQAN* is described in the supplementary material. It describes how to use AUP as a metric to decide when to perform a lazy synchronization. Given the queue capacity L and a position ratio R $(0 < R \leq 1.0)$, the threshold of AUP to do sync is set as $L \cdot R$. If the *checker* finds that AUP is equal to or larger than the threshold, it triggers a global sync. Empirically, we found that setting the ratio R to 0.9 or 0.8 works well in most cases.

5.4 Cost Analysis

In this part, we examine the cost of *iQAN* executed on multi-core processors. The running time of search using BFiS is:

$$T_{seq} = I_{seq} \times C_{seq} \tag{2}$$

where I_{seq} represents the number of iterations to converge to near neighbors, and C_{seq} represents the cost of node expansion within one step. Now consider the search with the edge-wise parallelism with P cores. This would lead to an execution time:

$$T_{EP} = I_{EP} \times (\frac{C_{EP}}{P} + C_{sync}) \tag{3}$$

Since edge-wise parallelism only partitions the distance computations during node expansion across P workers while still doing global synchronization after each step, $I_{EP} = I_{seq}$ and $C_{EP} = C_{seq}$. The scalability of edge-wise parallelism is limited because the iteration depth I_{EP} is large, and the synchronization cost C_{sync} is still high. To reduce the iteration depth, *iQAN* introduces path-wise parallelism, which reduces I_{EP} to I_{PP}, where $I_{PP} << I_{EP}$.

$$T_{PP} = I_{PP}(\downarrow\downarrow) \times (\frac{C_{PP}(\uparrow\uparrow)}{P} + C_{sync}) \tag{4}$$

However, by doing so, the node expansion cost increases from C_{EP} to C_{PP}, where $C_{PP} >> C_{EP}$ because path-wise parallelism introduces redundant computations. To avoid redundant computations and synchronization overhead, *iQAN* further introduces staged expansion and redundancy-aware synchronization, which reduces the node expansion cost from C_{PP} to C_{SE} with minimal impact on the iteration depth I_{PP} while reducing the amortized synchronization overhead from C_{sync} to C_{RAS}, leading to a search cost:

$$T_{iQAN} = I_{PP}(\downarrow\downarrow) \times (\frac{C_{SE}(\uparrow)}{P} + C_{RAS}(\downarrow)) \tag{5}$$

5.5 Discussion: Why not use Δ-stepping?

Some parallel graph processing frameworks such as Galios [45] and GraphIt [72] support various parallel processing algorithms. Therefore, the careful reader may think why not apply existing parallel graph processing algorithms, such as Δ-stepping [55], to the vector search problem. To answer this question, we apply Δ-stepping to graph-based ANNS with a few modifications to make it adapt to vector search. First, instead of keep increasing the number of buckets, which leads to unbounded search latency, we keep a fixed length of the priority queue and drop a vertex if it has a larger distance than the last vertex in the queue. This change allows us to control the search budget via adjusting the queue length and avoid expanding candidates that are not promising for refining top-K. Second, we choose the first B unchecked nodes in the priority queue as the bucket. We do not use a distance range because the range of pair-wise vector distances is very dynamic due to the curse of dimensionality phenomenon. Third, we find that using a single bucket size hurts the performance significantly, for a similar reason as described in Section 5.2. Therefore, we dynamically increase the bucket size. Finally, we let Δ-stepping stop when the candidates in the priority queue remain the same for two consecutive iterations, same as *iQAN*. We provide comparison results with Δ-stepping in Section 6.4.

6 Evaluation

6.1 Evaluation Methodology

Baselines. We compare *iQAN* with two state-of-the-art graph-based ANNS, NSG [19, 20] and HNSW [38, 40]. NSG employs *BFiS*, and HNSW uses a similar but slightly modified implementation adapted to its hierarchical index. The hyperparameters used for building their indices are set based on the suggested values from their code repo or papers if provided by the authors. Otherwise, several values are tested, and the best performance is reported.

Metrics. We use latency and recall to measure the performance as described in Section 3. We measure *Recall@100* (R@100), which measures the accuracy of finding the top-100 nearest neighbors for every query, according to Eqn. 1 with $K = 100$.

Datasets. This evaluation uses five datasets, which are standard benchmarks when evaluating NSG and HNSW. SIFT1M

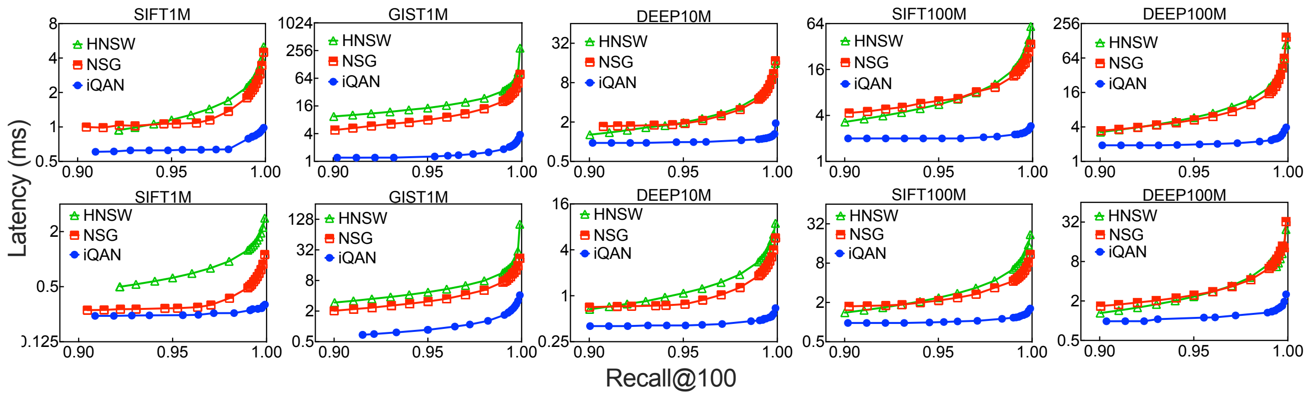


Figure 12. The first and second rows of figures show latency comparison among HNSW, NSG, and *iQAN* on KNL (32T) and Skylake (16T), respectively.

(128D) and GIST1M (960D) are from the datasets introduced by Jégou et al. [1, 28]; SIFT100M (128D) is sampled randomly from the SIFT1B (bigann) introduced by Jégou et al. [29]; DEEP10M (96D) and DEEP100M (96D) are subsets from DEEP1B that is released by Babenko and Lempitsky [7, 8]. Beyond these datasets, we also tried to evaluate the two billion-scale datasets SIFT1B (128D) and DEEP1B (96D). However, they easily run out of memory on our testbeds with 128GB memory (e.g., DEEP1B would require >384GB memory to load just the embedding vectors). This is probably also why prior works did not evaluate these two datasets. Therefore, we ran the billion-scale datasets on a specialized machine with 1.5TB DRAM. We include the details of the datasets in the appendix.

Implementation. Our implementation of *iQAN* is based on the state-of-the-art NSG using C++ [19]. For a given dataset (e.g., 1M–100M), we construct an NSG graph using the base set. We use the hyperparameters suggested by the NSG code repository to construct the graph due to their excellent performance in practice. The original NSG paper did not evaluate the billion-scale datasets due to their huge memory footprints. For billion-scale datasets, we use a machine that has DRAM that is sufficiently large to build the NSG graph. We then use the query vectors provided by the benchmarks for final evaluation. We set the average update position ratio as 0.8 for SIFT1M, GIST1M, and SIFT100M, and 0.9 for DEEP10M and DEEP100M.

Platform and settings. We conduct our experiments on a workstation with Xeon Phi 7210 (1.30 GHz) with 64 cores and 109 GB DRAM (*KNL* for short) and a workstation with Xeon Gold 6138 (2.00 GHz) with 20 cores and 128 GB DRAM (*Skylake* for short). To further evaluate two billion-scale datasets, we use a machine with Xeon Gold 6254 (3.10 GHz) with 72 cores and 1.5 TB memory.

6.2 Evaluation of Latency Speedups

Fig. 12 compares the latency of HNSW, NSG, and *iQAN* on KNL and Skylake. NSG and HNSW use their sequential search algorithm, whereas *iQAN* uses 32 threads and 16 threads, respectively, on KNL and Skylake. We make the following observations.

First, across all five datasets, iQAN *consistently provides latency speedups over existing sequential-based approaches NSG and HNSW over a wide range of recall targets.* In particular, the speedups from *iQAN* increase as the recall target moves to the high accuracy regime (e.g., from 0.90 to 0.999). On KNL, for R@100 targets 0.9, 0.99, and 0.999, *iQAN* achieves 2.2×, 5.9×, and 16.7× speedups over NSG on average over five datasets, and 2.8×, 8.1×, and 27.6× over HNSW. *iQAN* achieves even more latency speedups on large graphs. Notably, *iQAN* achieves up to 37.7× and 12.9× speedups over NSG on DEEP100M on KNL and Skylake, obtaining an incredibly low latency of <5ms or <3ms at the recall target 0.999 by leveraging aggregated multi-core computation and memory bandwidth resources. This enables vector search with very high accuracy on large-scale graphs, even in extremely interactive online applications.

iQAN achieves significant latency speedups mainly because of three reasons. First, *iQAN*'s path-wise parallelism effectively reduces the iteration depths, making the sequential dependencies no longer a major bottleneck. This is particularly critical for a large graph (e.g., DEEP100M) and high recall (e.g., 0.999) as seen in Section 4 that the iteration depths increase significantly as we either scale the graph size or increase the recall targets. Second, the reduced iteration depths do not come at the cost of many redundant computations as *iQAN* leverages staged expansion to effectively avoid redundant computations from doing path-wise parallelism. Third, *iQAN* significantly reduces the synchronization overhead through redundancy-aware synchronization.

Second, the results also show that *iQAN* exhibits latency speedups across different multi-core architectures. On Skylake, for R@100 targets 0.9, 0.99, and 0.999, *iQAN* also achieves 1.8×, 3.6×, and 7.3× speedups on average over NSG, and 2.1×, 5.6×, and 14.0× over HNSW. These results indicate that the optimizations in *iQAN* carry good portability across different multi-core architectures.

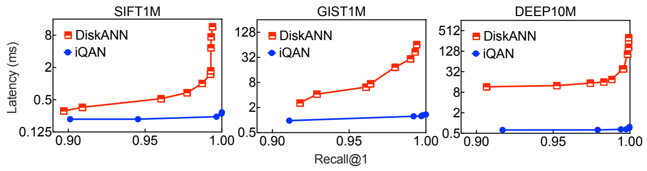


Figure 13. Recall@1 latency of DiskANN and *iQAN* on KNL.

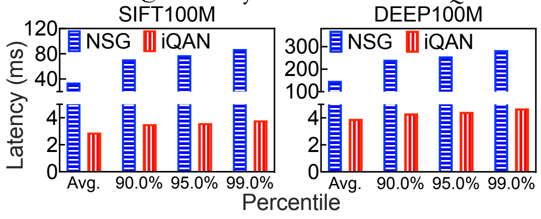


Figure 14. Percentile latency on KNL of *iQAN* & NSG.

Third, it is also worth mentioning that *iQAN* achieves excellent speedups as we increase the dimensionality of the embedding vectors. *iQAN* achieves up to 76.6× and 24.9× speedups over HNSW on GIST1M on KNL and Skylake, respectively. This is higher than the speedups we get on a dataset with a similar scale but much smaller dimensionality (e.g., SIFT1M). *iQAN* is able to achieve better speedups on higher dimensional vectors because as the vector dimension increases, the amount of computation workload for the pair-wise distance computation also increases, which allows *iQAN* to benefit more from parallel computing.

Moreover, Fig. 13 compares the latency of DiskANN [26] (using 1 thread with its in-memory index) and *iQAN* (using 32 threads) on KNL for *Recall@1* targets. For building its indices of datasets SIFT1M and GIST1M, DiskANN uses $L = 125$, $R = 70$, $\alpha = 2$, which are the same setting as shown in its paper. For DEEP10M, DiskANN uses $L = 100$, $R = 100$, $\alpha = 1.2$. Fig. 13 shows that *iQAN* also achieves latency speedups over DiskANN, especially for the high recall regime. For example, for recall target 0.999, *iQAN* has about 180.5× average speedup on DiskANN among these three datasets. For SIFT100M and DEEP100M, DiskANN requires larger memory than KNL has for searching, which causes Segmentation Fault.

Reducing tail latency. For online inference, tail latency is as important, if not more, as the mean latency. To see if *iQAN* provides steady speedups, we collect the 90th percentile (90%tile), 95th percentile (95%tile), and 99th percentile (99%tile) latency of NSG and *iQAN* on KNL on SIFT100M and DEEP100M at the recall target 0.999 in Fig. 14. The results show that while NSG's 99%tile compared to mean increases significantly by 154% and 91% for SIFT100M and DEEP100M, respectively, the *iQAN*'s 99%tile increases only by 31% and 19%. *iQAN* leads to a relatively smaller increase in tail latency, presumably because intra-query parallel search is particularly effective in reducing latency on long queries.

6.3 Evaluation of Scalability

In this part, we evaluate the scalability of *iQAN* with respect of the number of threads and graph scales.

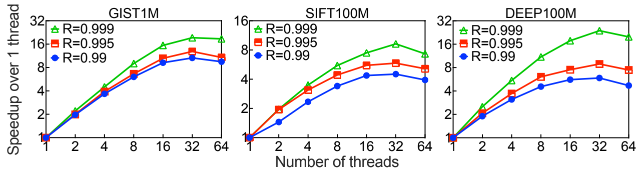


Figure 15. Speedups of *iQAN* over the seq. baseline on KNL.

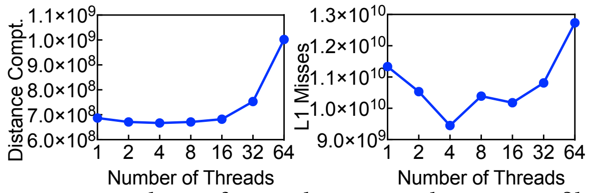


Figure 16. Analysis of super-linear speedups via profiling distance computations and L1 misses of DEEP100M on KNL.

Improving scalability with more threads. Fig. 15 reports the scalability of *iQAN* varying threads on KNL at different recalls (0.99, 0.995, and 0.999). The results show that the speedups increase as the target recall grows because the increased distance computations to reach a higher recall offers more parallelism opportunities. The average speedup of the three datasets at recall 0.999 is 13.5×, 17.4×, and 15.3× for 16-, 32-, and 64-thread, respectively. *iQAN* starts to observe diminishing return at 64 threads. For GIST1M that has high dimensional vectors, it is because 32-thread *iQAN* has saturated memory bandwidth. For other datasets, extra threads introduce unnecessary expansions, so that adding additional threads does not improve the latency further.

Exhibiting super-linear speedups. Fig. 15 also shows an interesting phenomenon: *iQAN* demonstrates super-linear speedup (up to 16 threads) for some high recall targets (e.g., 0.999). For example, *iQAN* on KNL achieves 9.0 and 17.7 speedups on GIST1M and DEEP100M with 8 and 16 threads, respectively. How is it possible for *iQAN* to achieve $> P$ times speedups with only P threads? To investigate this issue, we perform profiling to examine the executed operations and the cache behavior. Fig. 16 shows the profiled results on DEEP100M at recall 0.999. The super-linear speedup appears to come from two aspects: (1) Path-wise parallelism, together with the staged expansion, allows *iQAN* to reduce both the iteration depths and the redundant computations, which ends up letting some range of processors (i.e. 2 to 16 threads) to have a smaller number of distance computations. (2) *iQAN* obtains super-linear speedups also due to utilizing more cache memory effectively. As shown in Fig. 16, as we increase the number of threads, there is a decrease in the aggregated L1 cache misses since more cache memory is used in parallel search, while the sequential search cannot. The L1 cache misses increase from $P = 16$ to $P = 64$ because the total amount of distance computations tends to increase in those cases, where we stop seeing super-linear speedups.

Influence on inter-query. Fig. 17 shows *iQAN*'s intra-query latency and inter-query throughput on DEEP100M with different thread settings for recall targets 0.995, 0.997, and 0.999. We remark that although the goal of this paper is

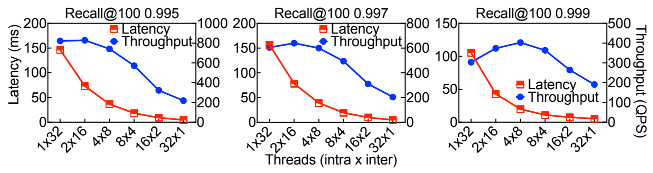


Figure 17. Intra/Inter-query of DEEP100M on KNL.

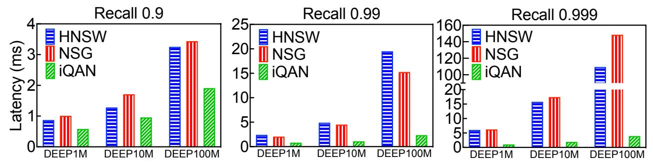


Figure 18. Scalability with varied graph sizes for *iQAN*, NSG, and HNSW on KNL. *iQAN* uses 32 threads.

to accelerate the latency, with a proper inter-/intra-query setting, *iQAN* can improve **both query latency and throughput simultaneously** for high recall targets. For example, for recall 0.999 on DEEP100M, *iQAN* using 4-way inter-query + 8-way intra-query parallelism outperforms NSG using 32-way inter-query parallelism by 1.7× throughput (queries-per-second or QPS) and 13.7× query latency improvement.

Varied inter-/intra-query configurations result in different query throughputs and latencies. On the one hand, inter-query parallelism increases the peak memory bandwidth roughly proportionally to the number of parallel queries, whereas its latency might be bounded by some long-latency queries. On the other hand, intra-query parallelism using multiple cores together can shorten the iteration depths and increase the cache capacity but may incur computation and synchronization overhead. As Fig. 17 shows, when using all 32 threads only for inter-query parallelism, the performance is bounded by the memory bandwidth and some long-latency queries. When the thread count of intra-query parallelism increases from 1 to 4, latency shows a super-linear speedup. This observation is consistent with the results in this subsection before that intra-query parallelism with a small number of threads can reduce the iteration depths, the redundant computations, and the cache misses. Moreover, the thread count of inter-query parallelism decreasing from 32 to 8 also reduces the memory bandwidth pressure. Therefore, there is a performance improvement both in latency and throughput. In Fig. 17, the improvement of throughput is more obvious for recall 0.999 than 0.997 and 0.995, as 0.999 requires more computation. When the thread count of intra-query parallelism continues to increase from 8 to 32, latency shows a sub-linear speedup because of the increased distance computation and synchronization overhead. Therefore, there is a decrease in throughput when the threads of inter-query parallelism decrease from 4 to 1. In conclusion, with a proper combination, inter- and intra-query together can achieve a good throughput performance.

Improving scalability on larger graph sizes. Fig. 18 reports the latency results of NSG, HNSW, and *iQAN* for the recall of 0.9, 0.99, and 0.999, varying the data sizes from 1M to 100M. As the graph size increases, the speedup of *iQAN* over NSG and HNSW increases. For example, for recall 0.999, the speedup of *iQAN* over NSG grows from 5.9× to 37.7× when the dataset size changes from 1M to 100M. These results confirm that *iQAN* is scalable and preferable to existing search methods when the graph size increases.

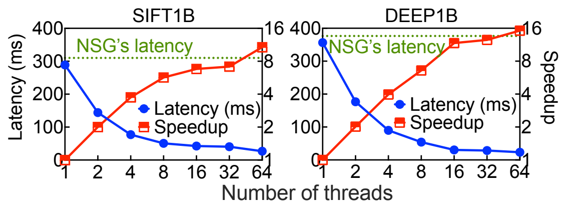


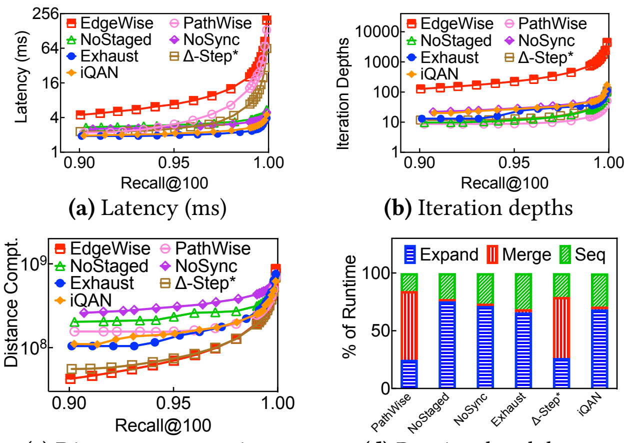
Figure 19. Performance comparison of *iQAN* and NSG on two billion-scale datasets SIFT1B (bigann) and DEEP1B.

Scaling to billion points. This experiment is conducted on a machine with a 1.5 TB memory. It is worth mentioning that even 1.5 TB of memory is not enough to build a 100-NN graph with one billion data vectors. Therefore, we limit the out-degree of NSG when generating the corresponding NSG index so that the index construction can finish in a reasonable amount of time (e.g.,<10 days). We also note that this is the first time to evaluate an NSG graph at a billion scale as the maximum graph prior work such as NSG evaluated contained less than 100M data points. Fig. 19 compares the latency of *iQAN* and NSG. *iQAN* uses up to 64 threads, and the recall target is 0.9. When using 64 threads, *iQAN* follows the trend of scalability we observed as we increase the graph size and outperform NSG with 11.5× and 16.0× speedup for SIFT1B and DEEP1B, respectively, confirming the excellent scalability of *iQAN* on large-scale graphs again.

6.4 Performance Breakdown and Analysis

This section performs a series of experiments to show where *iQAN*'s improvements come from. It first compares *iQAN*'s performance with several alternative parallel search schemes. (i) *EdgeWise*: NSG with edge-wise parallelism. (ii) *PathWise*: path-wise parallelism without the staged expansion nor redundancy-aware synchronization. (iii) *NoStaged*: *iQAN* without the staged expansion. (iv) *NoSync*: it performs path-wise parallelism but never synchronizes among workers until the very end. (v) *Exhaust*: it uses an exhaustive search to preprocess the dataset and obtain the proper synchronization settings. It should have the best latency performance, although requiring more than ten hours of tuning for the given dataset. (vi) *iQAN*: the full version with path-wise parallelism, staged expansion, and redundancy-aware synchronization. Apart from the above analysis, we also include a comparison with **Δ-stepping**. We apply the modifications described in Section 5.5 and denote the config as Δ-step*. We report results on the DEEP100M dataset with 32 threads on KNL in Fig. 20. Other datasets and threads show the same trend and thus are omitted due to the space constraint.

Latency improvements breakdown. Fig. 20a first reports the latency results of all versions. Compared with *EdgeWise*,



(a) Latency (ms) **(b)** Iteration depths **(c)** Distance computation **(d)** Runtime breakdown

Figure 20. Performance w/ 32 threads (KNL) on DEEP100M.

PathWise has a 1.7× speedup on average for all recall cases because of the iteration depths reduction from path-wise parallelism, and *NoStaged* has an extra 4.9× speedup on average because the redundancy-aware synchronization still reduces the overhead from frequent global synchronizations. *Exhaust* has a 1.5× speedup over *NoStaged* and a 1.3× speedup over *NoSync* mainly due to its reduction in redundant computations from using staged expansion and its infrequent synchronizations. *Exhaust* achieves slightly better performance than *iQAN* (i.e., a 1.1× speedup). However, *iQAN* does not require the expensive offline tuning process as *Exhaust*.

Effects on iteration depths. Fig. 20b profiles the iteration depths, where *EdgeWise* and *PathWise* result in the most and the fewest, respectively. All other five settings result in comparable iteration depths to *PathWise*. This is because *PathWise* employs a fixed and relatively large number of multiple paths throughout search, resulting in the most aggressive exploring. Its high synchronization frequency makes its iteration depth smaller than *NoStaged*. Meanwhile, *iQAN* and *Exhaust* adopt staged expansion that slightly increases the iteration depths but significantly reduces distance computations, and *NoSync* suffers more divergence than them.

Effects on distance computation. Fig. 20c profiles the number of distance computations. *NoStaged* leads to more distance computations than others except *NoSync* to achieve the same recall (especially for low recall cases). *PathWise* has smaller distance computations than *NoStaged*, thanks to its frequent synchronization. While completely removing synchronization, *NoSync* has the most distance computations. However, as shown in Fig. 20a, it still achieves lower latency than *NoStaged* because synchronization overhead can dominate the total search time when the number of parallel workers is large. Although *iQAN* and *Exhaust* do not synchronize as frequent as *PathWise*, they have lower distance computation thanks to staged expansion.

Effects on synchronization overhead. Fig. 20d reports the execution time breakdown of six approaches that apply our design. It splits the execution time into three parts: the Expanding (*Expand*), the Merging (*Merge*), and the Sequential (*Seq*). *Expand* denotes the parallel phase of a query that workers expand their unchecked candidates. It consists of computing distances and inserting visited neighbors into their queues. *Merge* denotes the phase in which workers merge their local queues into a global queue after they complete expansion. It reflects the major synchronization overhead. And sequential execution of a search is included in *Seq*. All results are for recall 0.999. Fig. 20d shows that the redundancy-aware synchronization strategy effectively mitigates the synchronization overhead, allowing *iQAN* to achieve a similar portion of synchronization overhead as *Exhaust* (∼2%) and a much smaller portion than *PathWise*.

Comparison with Δ-stepping. We observe that our improved version of Δ-stepping (Δ-step*) reaches a comparable latency as *iQAN* at the low recall regime (e.g.,0.90). However, as we increase the recall target (e.g., to 0.95 and 0.99), Δ-step*'s latency increases more significantly than *iQAN*, e.g., *iQAN* achieves 14.1× speedup over Δ-step* at recall target 0.999 and on average 3.3× speedup across a range of recall targets. By further analyzing the results, we find that although Δ-step* decreases the iteration depth and the amount of redundant distance computations, synchronization overhead becomes a crucial factor that affects the overall search efficiency. In contrast, *iQAN*'s update-position-based redundancy-aware strategy helps significantly reduce the synchronization overhead, especially at the high recall range, without hurting accuracy, leading to faster search speed.

7 Conclusion

This work looks into the problem of accelerating graph-based ANNS latency on multi-core systems, performing several studies to reveal multiple challenges in exploiting intra-query parallelism for speeding up ANNS. Based on the studies, we propose *iQAN*, a parallel graph search algorithm that takes advantage of multi-core CPUs to significantly accelerate the search without compromising search accuracy. Evaluation results show that *iQAN* is able to outperform two state-of-the-art methods NSG and HNSW on a wide range of real-world datasets while enjoying excellent scalability as the graph size and accuracy target increases, enabling vector search with very high accuracy on large-scale graphs even in extremely interactive online applications.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments and helpful suggestions. This work was supported in part by National Science Foundation (NSF) under the awards of CCF-2047516 (CAREER), CCF-2146873, IIS-2142681, III-2008557, and IIS-2142675. Any errors and opinions are not those of the NSF and are attributable solely to the author(s). The authors also acknowledge William & Mary Research Computing for providing computational resources.

References

[1] Laurent Amsaleg and Hervé Jégou. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/. [Online; accessed 25-April-2022].

[2] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*. IEEE, 459–468. https://doi.org/10.1109/FOCS.2006.49

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS)* (Montreal, Canada) *(NIPS'15, Vol. 28)*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.). MIT Press, Cambridge, MA, USA, 1225–1233. https://proceedings.neurips.cc/paper/2015/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2017. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. In *International Conference on Similarity Search and Applications (SISAP)*. Springer, 34–49.

[5] Mohsen Aznaveh, Jinhao Chen, Timothy A Davis, Bálint Hegyi, Scott P Kolodziej, Timothy G Mattson, and Gábor Szárnyas. 2020. Parallel GraphBLAS with OpenMP*. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 138–148.

[6] Artem Babenko and Victor Lempitsky. 2014. The Inverted Multi-Index. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37, 6 (2014), 1247–1260.

[7] Artem Babenko and Victor Lempitsky. 2016. Deep billion-scale indexing. https://sites.skoltech.ru/compvision/noimi/. [Online; accessed 25-April-2022].

[8] Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] KG Renga Bashyam and Sathish Vadhiyar. 2020. Fast Scalable Approximate Nearest Neighbor Search for High-dimensional Data. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 294–302. https://doi.org/10.1109/CLUSTER49012.2020.00040

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[11] 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.*

[12] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells (Eds.). ACM, 191–198.

[13] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (Brooklyn, New York, USA) *(SCG '04)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/997817.997857

[14] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2019. Gluon-async: A Bulk-Asynchronous System for Distributed And Heterogeneous Graph Analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 15–28.

[15] Roman Dementiev. 2017. Processor Counter Monitor. https://github.com/opcm/pcm. [Online; accessed 26-June-2022].

[16] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proceedings of the VLDB Endowment* 13, 3 (2019), 403–420.

[17] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. 159–170.

[18] Cong Fu and Deng Cai. 2016. Efanna: An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on Knn Graph. *arXiv preprint arXiv:1609.07228* (2016).

[19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment (VLDB)* 12, 5 (Jan. 2019), 461–474. https://doi.org/10.14778/3303753.3303754

[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. NSG. https://github.com/ZJULearning/nsg. [Online; accessed 25-April-2022].

[21] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2946–2953.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[23] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 233–245.

[24] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. Multigraph: Efficient Graph Processing on Gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 27–40.

[25] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[26] 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. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 13771–13781. https://proceedings.neurips.cc/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[27] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2008. Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search. In *European conference on computer vision*. Springer, 304–317.

[28] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[29] 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.

[30] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel*

and Distributed Computing (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[31] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[32] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1188–1196.

[33] D. T. Lee and C. K. Wong. 1977. Worst-case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees. *Acta Informatica* 9, 1 (March 1977), 23–29.

[34] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data – Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204

[35] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[37] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate Nearest Neighbor Algorithm Based on Navigable Small World Graphs. *Information Systems* 45 (2014), 61–68. https://doi.org/10.1016/j.is.2013.10.006

[38] Yury A Malkov and Dmitry A Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[39] Yury A. Malkov and D. 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. https://doi.org/10.1109/TPAMI.2018.2889473

[40] Yury A Malkov and Dmitry A Yashunin. 2020. Hnswlib. https://github.com/nmslib/hnswlib. [Online; accessed 25-April-2022].

[41] Clara Meister, Tim Vieira, and Ryan Cotterell. 2020. Best-First Beam Search. *Transactions of the Association for Computational Linguistics* 8 (2020), 795–809.

[42] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 201–213. https://doi.org/10.1145/3293883.3295716

[43] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[44] Maxim Naumov, Alysson Vrielink, and Michael Garland. 2017. Parallel Depth-First Search for Directed Acyclic Graphs. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[45] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[46] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Allen Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. 2019. Semantic Product Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 2876–2885.

[47] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. 2018. Graphphi: Efficient Parallel Graph Processing on Emerging Throughput-Oriented Architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/3243176.3243205

[48] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[49] Sergio Rivas-Gomez, Roberto Gioiosa, Ivy Bo Peng, Gokcen Kestor, Sai Narasimhamurthy, Erwin Laure, and Stefano Markidis. 2018. MPI windows on storage for HPC applications. *Parallel Comput.* 77 (2018), 38–56.

[50] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[51] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing Large-Scale Graphs on Accelerator-Based Systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[52] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[53] 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 (CVPR)*. IEEE, 1–8.

[54] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[55] Upasana Sridhar, Mark Blanco, Rahul Mayuranath, Daniele G Spampinato, Tze Meng Low, and Scott McMillan. 2019. Delta-stepping SSSP: From Vertices and Edges to GraphBLAS Implementations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 241–250.

[56] Danny Sullivan. 2018. FAQ: All about the Google RankBrain algorithm. https://searchengineland.com/faq-all-about-the-new-google-rankbrain-algorithm-234440.

[57] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery,

New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/3079079.3079097

[58] Nishil Talati, Di Jin, Haojie Yet, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. 2021. A Deep Dive Into Understanding The Random Walk-Based Temporal Graph Learning. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 87–100.

[59] Ruiqi Tang, Ziyi Zhao, Kailun Wang, Xiaoli Gong, Jin Zhang, Wenwen Wang, and Pen-Chung Yew. 2021. Ascetic: Enhancing Cross-Iterations Data Efficiency in Out-of-Memory Graph Processing on GPUs. In *50th International Conference on Parallel Processing*. 1–10.

[60] Alexandru Uta, Ana Lucia Varbanescu, Ahmed Musaafir, Chris Lemaire, and Alexandru Iosup. 2018. Exploring Hpc and Big Data Convergence: A Graph Processing Study on Intel Knights Landing. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 66–77.

[61] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111.

[62] Hans Vandierendonck. 2020. Graptor: Efficient Pull and Push Style Vectorized Graph Processing. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3392717.3392753

[63] Qihan Wang, Zhen Peng, Bin Ren, Jie Chen, and Robert G. Edwards. 2022. MemHC: An Optimized GPU Memory Management Framework for Accelerating Many-Body Correlation. *ACM Transactions on Architecture and Code Optimization* 19, 2, Article 24 (mar 2022), 26 pages. https://doi.org/10.1145/3506705

[64] 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.

[65] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[66] 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.

[67] 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. In *Advances in Neural Information Processing Systems (NIPS)*. 5745–5755.

[68] Yubao Wu, Ruoming Jin, and Xiang Zhang. 2014. Fast and Unified Local Search for Random Walk Based K-Nearest-Neighbor Query in Large Graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*. ACM, 1139–1150.

[69] Carl Yang, Aydın Buluç, and John D Owens. 2021. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Trans. Math. Software* (2021). https://arxiv.org/abs/1908.01407

[70] Eddy Z Zhang. 2018. A Simple Yet Effective Graph Partition Model for GPU Computing. In *Proceedings of the 47th International Conference on Parallel Processing Companion (ICPP)*. 1–2.

[71] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 183–193. https://doi.org/10.1145/2688500.2688507

[72] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO)* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 158–170.

A Artifact Evaluation

A.1 Getting Started Guide

A.1.1 Availability. The artifact is available on Zenodo (https://doi.org/10.5281/zenodo.7322631) and GitHub (https://github.com/johnpzh/iQAN_AE). Please note the artifact on GitHub does not contain the SIFT1M dataset because it exceeds the platform's file size limit. If using GitHub, please use the command to clone the repository into the local directory `iQAN_AE`.

```
git clone https://github.com/johnpzh/iQAN_AE.git
```

A.1.2 Hardware Environment. The artifact is supposed to be run on an Intel CPU with at least 32 cores and support AVX2 intrinsics. It does not use hyperthreading.

To run the quick script, the artifact requires 2 GB disk space and 4 GB memory. To run all full scripts, it needs 300 GB disk space and 80 GB memory. The particular number of required space are shown in Table **??**.

A.1.3 Software Environment. Some libraries and programs are required to build and run the artifact.

Libraries:

- Boost C++. The artifact is tested on version 1.53. To install Boost C++ on Ubuntu, please use the command

```
sudo apt install libboost-all-dev
```

- OpenMP. The artifact is tested on version 5.0.
- Matplotlib for plotting figures. The artifact is tested on version 3.1.2. To install Matplotlab in Python, please use the command

```
python -m pip install -U matplotlib
```

Programs:

- CMake for building the program (>= 3.9). The artifact is tested on 3.17.
- C++ Compiler, recommending Intel C++ Compiler (>= 19.0.1). The artifact is tested on GCC 4.8.5 and ICC 2021.4.0.
- Bash or Zsh for shell scripts.
- Python3 for Python scripts. The artifact is tested on 3.7.11.

A.1.4 Build. A script is provided to build the artifact.

Under the project directory, please run this command to build the artifact:

```
bash ./build.sh
```

If using the Intel ICC compiler, please run this command with the option `icc`:

```
bash ./build.sh icc
```

This command creates a directory `cmake-build-release` and builds the program under it.

Table 2. Dataset sizes and running time.

Datasets	SIFT1M	GIST1M	DEEP1M	DEEP10M	SIFT100M	DEEP100M
base file (GB)	0.5	3.6	0.3	3.7	49	37
NSG index (GB)	0.1	0.08	0.1	1.7	19	14
HNSW index (GB)	0.6	3.8	0.5	5	62	50
query file (MB)	5	3.7	3.8	3.8	5	3.8
groundtruth file (MB)	7.7	0.7	7.7	7.7	7.7	7.7
running time (hr)	3.5	12	5	15	24	30

A.1.5 Quick Run. A quick script can run the artifact using the included dataset SIFT1M. Please note the script should be run under the build directory.

```
cd cmake-build-release
bash ../scripts/run.quick.sh
```

This script runs *iQAN*, NSG, and HNSW for six recall targets (Recall@100) 0.9, 0.95, 0.99, 0.995, 0.997, and 0.999 on the dataset SIFT1M. It takes about 30 minutes to finish depending on the machines (e.g., 1 hour on KNL). The script can generate a figure `fig.quick.png` under the current directory `cmake-build-release`.

The generated figure should reflect the same trend as Fig. 13 in the paper, although the quick script only collects results for six recall targets. The figure provides a glimpse showing that *iQAN* can achieve latency speedup over NSG and HNSW for given recall targets.

A.2 Step by Step Instructions

A.2.1 Prepare Datasets. (If you use remote access, you do not need to prepare datasets as are ready to use.) The artifact is delivered with the included dataset SIFT1M, which is relatively smaller than other datasets used in the paper. Table 2 shows the sizes of all five datasets. Besides SIFT1M, the artifact provides scripts to download five other datasets. To download a particular dataset, please run the commands under the directory `data/`:

```
cd data
bash ../scripts/get.xxx.sh
```

Here xxx can be replaced by either `gist1m`, `deep1m`, `deep10m`, `sift100m`, or `deep100m`.

For a given dataset, *iQAN* uses a base file containing all data vectors, a query file containing all query vectors, a ground-truth file containing the real 100 nearest neighbors of all queries, and an NSG index file generated by using NSG implementation (https://github.com/ZJULearning/nsg). The NSG index can also be used by NSG searching method. Additionally, the HNSW index is generated by using HNSWLib (https://github.com/nmslib/hnswlib), which is used by HNSW searching method.

A.2.2 Run Scripts for Latency-vs-Recall Curves. After a particular dataset is ready, one can run *iQAN*, NSG, and HNSW upon the dataset by using the dataset's corresponding script provided by the artifact.

```
cd cmake-build-release
bash ../scripts/run.xxx.sh
```

Here xxx can be replaced by either `sift1m`, `gist1m`, `deep10m`, `sift100m`, or `deep100m`.

This script runs *iQAN*, NSG, and HNSW for nineteen recall targets from 0.9 to 0.999 on the corresponding dataset. The estimated running time of datasets is shown in Table ??. The script can generate a figure `fig.xxx.png`, under the build directory `cmake-build-release`. The figure shows the latency-vs-recall curves for *iQAN*, NSG, and HNSW, which are the main results of the evaluation in the paper (Fig. 13). The shorter latency is better.

A.2.3 Run Scripts for Speedup.

To get the speedup results, please run

```
cd cmake-build-release
bash ../scripts/run.speedup_xxx.sh
```

Here `xxx` can be replaced by `gist1m`, `deep10m`, `sift100m`, or `deep100m`.

This script runs *iQAN* recall targets 0.99, 0.995, and 0.999 on the corresponding dataset using 1 to 64 threads. The script can generate a figure `fig.speedup.xxx.png` under the build directory `cmake-build-release`. The figure shows *iQAN*'s speedup over its 1-thread latency (Fig. 15). The larger speedup is better.

A.2.4 Run Scripts for Graph-Size Scalability.

To get the graph-size scalability results, please run

```
cd cmake-build-release
# First, download dataset DEEP1M if not yet
bash ../scripts/get.deep1m.sh
bash ../scripts/run.graph_scale.sh
```

This script runs HNSW, NSG, and *iQAN* for recall targets 0.9, 0.99, and 0.999 on the datasets DEEP1M, DEEP10M, and DEEP100M. A figure `fig.graph_scale.png` will be generated under the build directory `cmake-build-release`. The figure shows *iQAN*'s scalability over large graphs (Fig. 18).

A.2.5 Programs, Parameters, and Scripts.

The iQAN mainly uses the following files for its implementation for relatively small datasets such as SIFT1M, GIST1M, and DEEP10M,

```
app/PSS_v5_distance_threshold_profiling.cpp
core/Searching.202102022027.PSS_v5.dist_thresh.
    profiling.cpp
```

For large datasets such as SIFT100M and DEEP100M, iQAN mainly uses:

```
app/PSS_v5_LG_distance_threshold_profiling.cpp
core/Searching.202102031939.PSS_v5.large_graph.
    dist_thresh.profiling.cpp
```

The two versions use the same search algorithm, while the small-dataset version uses a flatted graph format to improve the data locality. Correspondingly, NSG also use the same format for small dataset. For large datasets, the flatted format causes too large memory footprint. In that case, iQAN and NSG uses the standard graph format.

The iQAN takes the following parameters:

```
PSS_v5_distance_threshold_profiling
    <data_file> <query_file> <nsg_file>
    <K> <place_holder> <true_NN_file> <num_threads
    >
    <L_low> <L_up> <L_step>
    <place_holder> <place_holder> <place_holder>
    <X_low> <X_up> <X_step>
    <place_holder> <place_holder> <place_holder>
```

- `data_file`: the input file containing all vectors, such as `sift_base.fvecs`
- `query_file`: the input query file containing all query vectors, such as `sift_query.fvecs`
- `nsg_file`: the input NSG index file, such as `sift.nsg`
- `K`: the value `K` as in Top-K, which is set as 100 for current implementation
- `true_NN_file`: the input file contains the real top-100 neighbors' IDs for all queries, used for computing the recall
- `num_threads`: the number of threads
- `L_low`, `L_up`, and `L_step`: the settings for the capacity of queues (the value `L`). A larger `L` uses larger queues, which can improve the search accuracy but also increase the search latency. Here the program will run multiple times with different values of `L` from `L_low` to `L_up` (inclusive) with step `L_step`. For example, a setting of (100, 102, 1) lets the program run with `L` as 100, 101, and 102. A user can then choose the expected output that satisfies the accuracy target and also achieves the shortest latency.
- `X_low`, `X_up`, and `X_step`: the settings for the synchronization frequency (the value `X`). A larger `X` has less frequent synchronization (merging local queues) among threads, which can reduce the synchronization overhead, improve the search accuracy but also increase the distance computation overhead. Similar with `L`, here the program will run multiple times with different values of `X` from `X_low` to `X_up` (inclusive) with step `X_step`.

Under the directory `scripts/`, the shell script `sh.iqan_xxx.sh` drives the program. First, it uses the python script `test51.PSS_v5_dt_profiling_ranged_L.py` to provide input and format the output. Second, it uses the python script `output_find_runtime_above_presicion.py` to select the output of best performance. The final output can then be used to generate figures by the script `run.xxx.sh`.