LLNL-PROC-852700

Towards A Massive-Scale Distributed Neighborhood Graph Construction

K. Iwabuchi, T. Steil, B. Priest, R. Pearce, G. Sanders

August 3, 2023

Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis
Denver, CO, United States
November 12, 2023 through November 17, 2023

Towards A Massive-Scale Distributed Neighborhood Graph Construction

Keita Iwabuchi
kiwabuchi@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Trevor Steil
steil1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Benjamin Priest
priest2@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Roger Pearce
rpearce@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Geoffrey Sanders
sanders29@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

ABSTRACT

Graph-based approximate nearest neighbor algorithms have shown high performance and quality. However, such approaches require a large amount of memory and still take a long time to construct high-quality nearest neighbor graphs (NNGs). Using distributed memory systems is important when data is large or a shorter indexing time is desired.

We develop a distributed memory version of NN-Descent, a widely known graph-based ANN algorithm, closely following algorithmic advances by PyNN-Descent authors. Our distributed NN-Descent (DNND) is built on top of MPI and leverages two existing high-performance computing libraries: YGM (an asynchronous communication library) and Metall (a persistent memory allocator).

We evaluate the performance of DNND on an HPC system using billion-scale datasets, demonstrating that our approach shows high performance and strong scaling and has great potential for developing massive-scale NNG frameworks.

CCS CONCEPTS

• **Computing methodologies → Machine learning algorithms**; • **Theory of computation → Graph algorithms analysis**; **Distributed algorithms**.

KEYWORDS

approximate nearest neighbor, distributed computing

ACM Reference Format:
Keita Iwabuchi, Trevor Steil, Benjamin Priest, Roger Pearce, and Geoffrey Sanders. 2023. Towards A Massive-Scale Distributed Neighborhood Graph Construction. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3625132

1 INTRODUCTION

The k-nearest neighbor (k-NN) search is the task of finding the closest (most similar) k data points in a given dataset from a query point, and it has various application fields, such as recommendation systems, anomaly detection, and large language models (LLM). Each data point is represented as a vector with tens to sometimes thousands of dimensions, and the search dataset may contain millions to tens of billions of points, with the size of the data expected to increase in the future. Therefore, a brute-force approach, which compares the query point to every data point, is impractical. Moreover, many applications do not require an exact solution, and techniques called approximate nearest neighbor (ANN) search are widely used. These methods usually compute approximate nearest neighbor data/information out of the input datasets in advance that is used to conduct ANN searches. Popular ANN algorithm categories are: tree-based methods like k-d trees, hash function-based methods like Locality-Sensitive Hashing (LSH) [11], quantization-based methods that quantize the data and utilize that information (e.g., Product Quantization [16]), and graph-based methods. Graph-based methods construct approximate nearest neighbor graphs (ANNGs) and perform graph traversals to find the nearest neighbors. Neighbor relationships can be naturally embedded into graph structures, and constructing ANNGs is performed by directly applying the given distance functions to the original data points. Because of that, graph-based methods often offer high flexibility and high accuracy [8, 10] compared to the other methods. However, on the other hand, graph-based approaches consume a lot of memory, and constructing high-quality ANN data requires a large amount of time generally.

To address the drawbacks, we leverage distributed-memory systems with some high performance computing (HPC) libraries. This paper introduces Distributed NN-Descent (DNND), our novel distributed memory version of a well-known graph-based ANN algorithm, NN-Descent [6]. DNND employs two high-performance computing libraries: YGM [21, 25], an asynchronous message communication library that provides efficient distributed communication, and Metall [13], a persistent memory allocator that enables applications to allocate data in a file system transparently. We also propose communication-saving techniques to accelerate the NN-Descent algorithm on distributed memory systems.

We evaluate the performance of DNND using up to billion-scale datasets on an HPC system. We demonstrate that our communication-saving technique cut half the amount of distributed messages on the 1 billion datasets. Compared to Hnswlib [17], a state-of-the-art shared-memory graph-based ANN library, DNND was able to construct similar- or better-quality ANNGs up to 4.7 times faster using 16 compute nodes. DNND shows promising results for developing massive-scale NNG construction frameworks.

The following sections provide details of the NN-Descent algorithm, followed by our DNND's design and implementation. We also present comprehensive performance evaluation results, offering key insights into the effectiveness and potential of DNND in tackling the challenges of massive-scale ANN graph construction.

2 PRELIMINARIES

In this paper, we use the following notation, basically following the one used in the original NN-Descent paper [6]. Let V be a dataset that contains $N = |V|$ points/vertices/feature vectors (we use these terms interchangeably). We assume every feature vector has the same number of dimensions. Let G be a k-nearest neighbor graph (k-NNG) constructed from V. G_v denotes the neighbor list of vertex $v \in V$, and the number of neighbors in G_v is K ($K < N$). We use $\theta(v_1, v_2)$ to denote the distance between v_1 and v_2, where $v_1, v_2 \in V$ and the distance metric θ returns a value $d \in [0, \inf)$ — the smaller the value, the closer the distance. We assume θ is a symmetric function, i.e., $\theta(v_1, v_2) = \theta(v_2, v_1)$.

As for data size, a dataset V is $N \times dim \times E$ bytes, where N is the number of points in the dataset, dim is the dimension of the feature vectors, and E is the size of the feature vector element type (e.g., E is 4B for a single-precision floating-point). A k-NN graph G consumes $k \times N \times T$ bytes, where T is the size of the point ID type (e.g., 4B for unsigned int). Because datasets and k-NN graphs are dense, those numbers are actual memory/storage consumption if they are stored in the binary format.

3 NN-DESCENT

This section describes the details of the NN-Descent [6] algorithm and the reasons we use the algorithm for our distributed neighbor graph construction.

3.1 NN-Descent Algorithm

NN-Descent is a heuristic and iterative approach to construct an approximate k-nearest neighbor graph (k-NNG). Although its basic idea is simple, it can achieve high-quality k-NNGs efficiently in practice. NN-Descent's empirical cost is round $O(n^{1.14})$ whereas brute-force takes $O(n^2)$ [6]. The key concept of NN-Descent is that *my neighbors will likely also be close to each other.* Specifically, if vertices v_0 and v_1 are not neighbors yet but are in G_{v_2} (the nearest neighbor list of v_2) at the time, v_0 and v_1 will likely be located at a close distance. Therefore, v_2 suggests v_0 and v_1 check the distance between them and update their nearest neighbor lists if necessary. NN-Descent initializes a k-NNG randomly and repeats *neighbor checks* until the number of newly discovered close neighbors fall below a pre-defined value. NN-Descent needs to use a distance function only for neighbor checks and works on any data as long as the distance metric can calculate the distance between any vertex pair in the dataset.

We show its detailed algorithm in Algorithm 1. The algorithm takes a dataset V, a similarity function θ, and algorithm parameters, ρ, and δ. The output is a K-NNG G (every vertex has K approximated nearest neighbors). $G[v]$ represents G_v (the neighbor list of vertex v), and $G[v][k]$ represents the k-th closest neighbor of v.

The first parallel for-loop block from line 2 initializes G by randomly generating K initial neighbors for every vertex in V.

The for-loop block at line 7 generates two arrays (*new* and *old*) for every vertex in the k-NNG (G). To avoid generating duplicate neighbor checks from the same vertex, NN-Descent marks neighbors in G as either *old* or *new*. When a new neighbor entry is inserted in G, the entry is marked as *new*. NN-Descent picks up ρK new items at a time (ρ is the sample rate, e.g., 0.8), and those selected items are marked as *old*.

In line 11–12, NN-Descent first generates reversed (transposed) *old* and *new* matrices (old' and new'). Those matrices are generated hoping that close neighbor status stays true even in the other direction. Then, NN-Descent samples the entries from the reversed matrices (old' and new') and merges with the corresponding original matrix *old* and *new*, respectively.

The loop block from line 17 is the core of the NN-Descent algorithm. In the loop, it performs neighbor checks between the pairs generated from the *old* and *new* matrices.

If the number of newly discovered close neighbors is less than δKN (e.g., $\delta = 0.001$), NN-Descent terminates the algorithm. Otherwise, it repeats the same steps from line 6. δ controls the graph quality vs. speed trade-off — the higher the value, the higher the accuracy and computational cost are expected.

3.2 NN-Descent Analysis

Neighbor relationships can be naturally embedded into graph structures, and constructing ANNGs is performed by directly applying the given distance functions to the original data points. Because of that, graph-based methods often offer high flexibility and high accuracy [8, 10] compared to the other methods.

On the other hand, the biggest weakness of the graph-based approach is its memory usage. Specifically, it is necessary to retain the original dataset not only during the construction of the NNGs but also during subsequent NN searches. k-NNG tends to require large k values to achieve high quality NNGs. However, as k increases, the memory consumption and the computational cost of the k-NNG also increase.

Therefore, this study aims to investigate whether it is possible to develop a large-scale k-NNG construction framework that maintains high versatility and performance by utilizing distributed memory systems and HPC techniques.

Among the many graph-based approaches, we chose NN-Descent [6]. NN-Descent performs relatively simple graph processing while supporting arbitrary distance functions. Wang et al. conducted a comprehensive study of graph-based ANN. They claimed that NN-Descent (and its derived algorithms) achieved the highest graph accuracy and high graph construction performance compared to other graph-based algorithms [23]. The core operation of NN-Descent is checking the distance between two neighbor

```
Data: dataset $V$, similarity function $\theta, K, \rho, \delta$
Result: $K$-NNG $G$
begin
    parallel for $v \in V$
        for $k \leftarrow 1$ to $K$ do
            $u \leftarrow$ Sample($V$, 1)
            $G[v][k] \leftarrow (u, \theta(v, u), true)$
    while $true$ do
        parallel for $v \in V$
            $old[v] \leftarrow$ all items in $G[v]$ with $false$ flag
            $new[v] \leftarrow \rho K$ items in $G[v]$ with $true$ flag
            Mark sampled items in $G[v]$ as false
        $old' \leftarrow$ Reverse($old$)
        $new' \leftarrow$ Reverse($new$)
        $c \leftarrow 0$
        parallel for $v \in V$
            $old[v] \leftarrow old[v] \cup$ Sample($old'[v], \rho K$)
            $new[v] \leftarrow new[v] \cup$ Sample($new'[v], \rho K$)
            foreach $u_1, u_2 \in new[v], u_1 < u_2$ or
              $u_1 \in new[v], u_2 \in old[v]$ do
                $d \leftarrow \theta(u_1, u_2)$
                /* $c$ and $G$ are atomically updated */
                $c \leftarrow c+$ Update($G[u_1]$, ($u_2, d, true$))
                $c \leftarrow c+$ Update($G[u_2]$, ($u_1, d, true$))
        if $c < \delta K|V|$ then return $G$
Function Reverse(A)
    $A' \leftarrow$ Transpose matrix $A$
    return $A'$
Function Sample($S, n$)
    $s \leftarrow$ Sample $n$ items from set $S$
    return $s$
Function Update($H, (v, d, f)$)
    $md \leftarrow$ Farthest neighbor distance in $H$
    if $v \notin H$ and $d < md$ then
        Pop farthest neighbor from $H$
        Push $(v, d, f)$ to $H$
        return 1
    return 0
```

Algorithm 1: NN-Descent algorithm [6]

vertices of a vertex. Similar operations can be seen in general graph processing, such as triangle counting, and there have been many prior studies in those algorithms [20]. Another advantage of NN-Descent is that the final output is a simple graph data structure where each vertex has k nearest neighbors. This provides high convenience for other applications that want to utilize k-NNGs.

3.3 Approximate Nearest Neighbor Search

Although the focus of this study is the graph construction step, we explain how to perform an ANN search on a k-NNG to provide readers with a better understanding of the background information.

By performing graph traversals on a k-NNG, it is possible to efficiently find approximate nearest neighbors for a given point with high accuracy in practice. To be clear, the query point used in the search does not need to exist in the dataset used to construct the graph, and the number of nearest neighbors to search for can be larger than k.

Here, we describe a naïve yet efficient search algorithm. Let's consider the case where we search l nearest neighbors from a query point (feature vector) q on a k-NNG G. First, l points are chosen randomly from G. Then, the distances from q to the chosen points are calculated, and the results are stored in two heaps: *frontier* heap and *l-NN* heap. The frontier heap $heap_f$ holds the point IDs to visit in the future and places the nearest element at the top, and the l-NN heap $heap_l$ holds q's up to l nearest neighbors and places the farthest point at the top. Next, the nearest point p is popped from the $heap_f$, and for every neighbor point w of p (i.e., $w \in G_p$), the distance $\theta(q, w)$ is calculated if w has not yet been visited. If $\theta(q, w)$ is smaller than the top element in $heap_l$, $heap_l$ is updated, and the element is also added to the $heap_f$. This process is repeated until A) the $heap_f$ becomes empty or B) $heap_f.top() > heap_l.top()$ is satisfied (i.e., the closest point in the frontier is already farther than the most distant point in the l-NN heap). When the algorithm finishes, the $heap_l$ contains the l nearest neighbors of q.

This simple greedy algorithm visits and finishes searching for far fewer points than N (the number of points in G) in practice. While this allows for rapid search completion, the search could fall into a local minimum and fail to find the optimal nearest neighbors. Therefore, PyNNDescent [18] introduces a parameter *epsilon* (*epsilon* ≥ 0) to relax the conditions for adding points to the $heap_f$, thereby expanding the search range. Specifically, when the distance between the query point q and a point p in G is calculated, p is added to the $heap_f$ if the condition $(epsilon + 1)dmax > \theta(q, p)$ is met, where $dmax$ is the distance to the farthest point in $heap_l$ at the time. This allows the search space to be expanded. Thus, there is a trade-off between search quality and search time, and using a large *epsilon* value can lead to a significant reduction in search performance.

4 DNND: DISTRIBUTED NN-DESCENT

Our DNND implementation closely follows the approach taken by PyNNDescent [18]. PyNNDescent is a shared-memory NN-Descent implementation written in Python. The library employs some optimization techniques to improve the performance and quality of NN-Descent. Our contributions consist of building their algorithms utilizing YGM (asynchronous communication library) and Metall (persistent memory allocator) and reorganizing communication patterns to reduce off-node communication.

DNND distributes a k-NNG G and an input dataset V equally among all MPI ranks based on the hash values of the vertex IDs. Each vertex (feature vector) $v \in V$ and the corresponding neighbor list G_v are located in the same MPI rank.

There are three large phases that require distributed-memory communication in NN-Descent in Algorithm 1. The first one is k-NNG initialization (for-loop block starts on line 2). The second one is the generation of reversed *old* and *new* matrices (line 11–12). The third one is the neighbor check step (for-loop block starts on line 17). We describe how we implement these phases in the following subsections.

4.1 Asynchronous Communication

DNND employs an asynchronous distributed-memory communication model. To implement an asynchronous computing model, DNND utilizes YGM [21, 25]. YGM is an asynchronous irregular communication library built on top of MPI (message passing interface) and is designed for irregular communication patterns. It employs fire-and-forget remote procedure call semantics. Specifically, a sender provides a function and function arguments for execution on a specified destination rank through an async call. This function will be invoked on the destination rank at an unspecified time in the future, but YGM does not explicitly make the sender aware of this completion to avoid synchronization costs. YGM buffers messages internally to increase communication throughput. When YGM's barrier function is called, all ranks wait until all messages are processed.

For example, to calculate the distance between v_a and v_b from the owner rank of v_a, DNND first sends the feature vector of v_a and a user-level function to the owner rank of v_b asynchronously. When the message is executed at the destination rank, YGM invokes the user-specified function, which A) calculates the distance between v_a and v_b and B) asynchronously sends back the distance value to the original rank where v_a exists. This communication pattern is used to initialize the k-NNG.

Because distributed NN-Descent performs various types of irregular communication patterns, YGM is a suitable library for implementing DNND. In the NN-Descent algorithm, the key descent operation is the neighbor checks, as described in lines 19– 20 in NN-Descent Algotimthm 1. This pattern is similar to *wedge queries* performed in triangle enumerations, for which YGM has demonstrated it was highly performant in the distributed setting [20]. We explain how the neighbor check step is conducted in DNND efficiency in Section 4.3.

4.2 Generating Reverse Neighbors

Generating the *old* and the *new* matrices does not require communication between MPI ranks, whereas generating reversed (transposed) *old* and *new* matrices does. DNND first makes reversed *old* and *new* matrices locally, then sends them to the corresponding ranks. When sending the reversed matrices, DNND randomly shuffles the order of the destination vertices to avoid sending data to the same rank from multiple ranks simultaneously and causing communication congestion at the destination rank.

4.3 Neighbor Checks with Communication Saving

In the distributed setting, it is possible to nearly halve the amount of data that are sent off nodes in the neighbor check step.

In lines 19–20 in Algorithm 1, neighbor checks are performed for both directions between u_1 and u_2. In the distributed setting, both u_1 and u_2 send their feature vectors to each other during the neighbor check. Since the size of a feature vector is the dimensions of the dataset, the communication cost is high. Figure 1a shows an *unoptimized* distributed NN-Descent neighbor check pattern. For

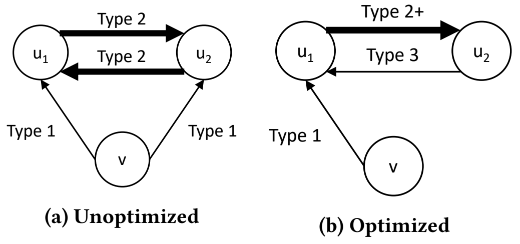


Figure 1: NN-Descent neighbor checking communication patterns (unoptimized vs. optimized). The line thickness represents the message volume.

convenience, hereafter, we call the neighbor check message from a center vertex as *Type 1* message and the feature vector message as *Type 2* message.

We propose three communication-saving techniques and describe the optimized communication in Figure 1b.

4.3.1 One-sided Communication. We can reduce the communication cost by employing a one-sided communication pattern. Instead of the central vertex v sending a Type 1 message to u_1 and u_2, v sends a Type 1 message to only u_1. Then u_1 subsequently sends its feature vector to the other vertex u_2. After receiving the feature vector, u_2 calculates the distance between itself and u_1 and returns the distance to u_1. This distance returner message is called *Type 3* message in Figure 1b.

4.3.2 Redundant Neighbor Check Reduction. The second optimization we propose is the reduction of redundant neighbor checks. After u_1 receives a neighbor check message (Type 1) from v, it checks if u_2 is already in its neighbor (i.e., $u_2 \in G_{u_1}$). If that is true, sending a Type 2 message to u_2 is a waste, obviously; therefore, u_1 does not send a Type 2 message to u_2. DNND also invokes the same check before sending the Type 3 messages from u_2 to u_1.

4.3.3 Pruning Long Distance Messages. The third method is another way to reduce the number of Type 3 messages. If u_2 is already farther than u_1's most distant neighbor ($G[u_1][k]$), sending back a calculated distance value from u_2 to u_1 is a waste. Therefore, u_1 attaches the distance to its most distant neighbor $\theta(u_1, G[u_1][k])$ to Type 2 messages. After calculating $\theta(u_1, u_2)$ at u_2, u_2 sends back the distance only when $\theta(u_1, u_2) < \theta(u_1, G[u_1][k])$. In this proposed method, the distance information attached to a Type 2 message is very small compared to the feature vector. Therefore, by sending data that is negligible in size, DNND can stop some unnecessary messages from u_2. This modified Type 2 message is called *Type 2+* message in Figure 1b.

4.4 Batched Communication

Although YGM is designed for irregular communication patterns and automatically sends messages when its internal buffer exceeds a certain threshold, we found that periodically performing global synchronization to send off buffered messages in the application layer brings great benefits when performing massive volume communication.

Because YGM does not have real-time global knowledge of the number of messages in all processes' buffers, a large number of messages are sent from many ranks simultaneously, potentially causing a kind of traffic congestion in YGM or lower-level communication libraries like MPI.

Therefore, when DNND needs to send a large number of messages at a time, it calls YGM's barrier function every time after passing a certain number of requests to YGM (e.g., 2^{25}–2^{30} requests globally).

4.5 Nearest Neighbor Graph Optimizations

In addition to making NN-Descent work on distributed memory systems, we also implemented two k-NNG optimization techniques implemented in PyNNDescent to increase ANN search performance and accuracy.

The first one is generating reverse neighbors. After constructing a k-NNG G, one of the simple yet effective optimizations is adding edges for the opposite directions (in other words, merging a transposed G to the original G). This optimization makes a graph more densely connected. We remove duplicate edges after merging reverse edges.

The second one is limiting the neighborhood size. Although the NN-Descent step creates a k-NNG, where each vertex contains k nearest neighbors, applying the first graph optimization could produce vertices that contain large numbers of neighbors. DNND prunes such high-degree vertices' neighborhood sizes up to $k \times m$, where m is a constant number and $m >= 1$ (e.g., $m = 1.5$).

4.6 Leveraging Persistent Memory Allocator

We also employ a persistent memory allocator, Metall [13], to increase the usability of our DNND at massive-scale ANN computations. Constructing high-quality k-NNGs for massive-scale data requires significantly more time than performing ANN searches on the constructed graphs. Therefore, the ability to store the constructed graph data in some form of persistent storage is highly beneficial for large-scale ANN.

Metall is built on top of the file-backed memory-mapped mechanism, i.e., mmap(2) system call, to enable applications to allocate data in files transparently. Metall has a C++ memory allocator interface compatible with the C++ Standard Template Library (STL). We can store STL Container-based data structures without writing file I/O code or dedicated data structures that work with only a particular library.

5 EVALUATION

We evaluate our DNND's k-NNG construction performance and quality on a large-scale distributed memory system using up to billion-scale datasets.

5.1 Setup

Here we describe the dataset, the evaluation environment, and the configuration of DNND.

5.1.1 Dataset. We use 8 datasets, including non-L_p distances, from the ANN-Benchmarks [1] and the Big ANN Benchmarks [19]. We list the datasets in Table 1.

Table 1: Datasets used in the evaluation.

Dataset	Dimensions	Entries	Similarity Metric
Fashion-MNIST	784	60,000	L_2
GloVe 25	25	1,183,514	Cosine
Kosarak	27,983	74,962	Jaccard
MNIST	784	60,000	L_2
NYTimes	256	290,000	Cosine
Last.fm	65	292,385	Cosine
Yandex DEEP 1B	96	1 billion	L_2
BigANN	128	1 billion	L_2

5.1.2 Evaluation Environment. Experiments were run on the *Mammoth* cluster at Lawrence Livermore National Laboratory. There are around 50 compute nodes, and each node has dual 64-core AMD EPYC 7742 processors (2.25 GHz) and 2048 GiB of memory. Nodes are connected with a Cornelis Networks Omni-Path interconnect. As for the MPI library, we used MVAPICH2.

5.1.3 DNND Configuration. Unless specifically mentioned, we used the following algorithm parameters in DNND during all evaluations. During the NN-Descent construction, we set the early termination parameter δ to 0.001 and the sample rate ρ to 0.8. In the graph optimization phase, we set the neighborhood size limit parameter m to 1.5. We set the communication batch size to 2^{28} requests for all top 6 datasets except Kosarak (2^{25}) and 2^{29} for the bottom two datasets.

There are two DNND execution files: one for k-NNG construction and the other for graph optimization. The k-NNG construction program stores the constructed k-NNG and the corresponding dataset V (matrix data) in Metall, and the optimization program reads the data from Metall and performs the optimizations described in Section 4.5. As for the filesystem location, where Metall allocates a k−NNG and a dataset, we used a tmpfs filesystem (/dev/shm) to avoid storage I/O overhead.

5.2 Preliminary NN Graph Quality Evaluation

Our first evaluation criterion is the quality of k-NNGs. To confirm DNND achieves reasonable k-NNG accuracy, we ran DNND on the top 6 small graphs in TABLE 1 and compared the achieved k-NNGs to the ones computed by a brute-force approach. The brute-force approach performs similarity comparisons between all pairs in the datasets.

We calculate a recall score for every point in a k-NNG and report the average per graph. The recall score is the ratio of the neighbor IDs that exist in the corresponding ground truth data.

DNND was able to construct k-NNGs ($k = 100$) with 0.93 and 0.98 recall scores for NYTimes and Last.fm, respectively. The rest of the datasets' scores are equal to or more than 0.99. Those results indicate that DNND can construct high-quality k-NNGs.

5.3 Billion-scale k-NNG Construction

Using the datasets that contain one billion points in Table 1, we evaluated DNND regarding the following two aspects: A) the scalability, and B) how quickly it can construct a graph of similar quality compared to a state-of-the-art ANN library (Hnswlib [17]).

As for k-NNG quality checking, it is impractical to construct ground truth k-NNGs using a brute-force method. Therefore, we evaluate the quality of constructed graphs in terms of the trade-off between query time and recall score using the query and ground truth data contained in the dataset.

The Yandex DEEP 1B and the BigANN datasets use float32 and uint8 data types, respectively, and we used those types in DNND and Hnswlib. We also used uint32 to represent point IDs.

5.3.1 DNND Configuration for 1 billion Datasets. We constructed k-NNGs with $k = 10$, $k = 20$, and $k = 30$, where k represents the number of neighbors each vertex has after construction. We varied the number of compute nodes up to 32 (128 processes per node). We used at least 4, 8, and 16 nodes for $k = 10$, $k = 20$, and $k = 30$, respectively, because the jobs did not finish within a reasonable time limit or encountered an out-of-memory (OOM) error with fewer compute nodes. DNND's other parameters are described in Section 5.1.3.

Query Program: We implemented a shared memory query program using C++ and OpenMP. Our query program employs the same search algorithm as PyNNDescent, which is described in Section 3.3. In a preliminary evaluation, we confirmed that our query program could achieve identical recall scores and efficiency (in terms of the number of performed distance calculations) compared to PyNNDescent. As for the $epsilon$ parameter, we used 0 and varied the value from 0.1 to 0.4 by 0.025 in this billion-scale evaluation.

5.3.2 Hnswlib Configuration. We used Hnswlib [17], a state-of-the-art shared-memory ANN library, for performance comparison. Hnswlib does not construct a general structure k-NNG, like NN-Descent and DNND do — achieving a portable k-NNG from Hnswlib requires additional processing. However, Hnswlib and DNND are both graph-based ANN libraries that support various distance metrics. It also showed better performance over a product quantization-based library (Faiss [15]) in terms of index construction time and query quality vs. performance trade-off [17]. Because of those reasons, we compare their performance to DNND. Hnswlib's core code is written in C++, and we used the C++ interface. We used 256 threads on Mammoth and set job time limits to 24 hours.

Hnswlib uses two parameters when building the graph, M and $ef\_construction$ (referred to as efc from here). There is also an ef parameter for nearest neighbor search. Generally, using larger values improves the qualities of graphs and queries but also increases the run time.

On the other hand, there is no obvious relationship between NN-Descent (DNND) and Hnswlib parameters for a fair comparison. Therefore, we conducted a wide range of parameter surveys for Hnswlib and carefully selected the Hnswlib results to compare against DNND. More specifically, to select reasonable parameters for Hnswlib for performance comparison, we first constructed graphs using Hnswlib varying their parameters, followed by performing nearest neighbor searches on the constructed graphs using their query function. Second, we did the same for DNND using the parameters described in Section 5.3.1. Third, for each graph constructed by DNND, we looked for the Hnswlib-constructed graphs that show almost the same or better query recall scores with almost the same or shorter time. In case multiple graphs were found, we picked the one with the shortest graph construction time.

We show the selected Hnswlib graph's parameters in Table 2.

Table 2: Hnswlib parameters.

Dataset	Yandex DEEP 1B		BigAnn	
Label	Hnsw A	Hnsw B	Hnsw C	Hnsw D
M	64	64	32	64
efc	50	200	25	200
ef	20–1200		20–1000	

5.3.3 Query Recall-score vs. Performacne Trade-off. Since we chose the Hnswlib graphs based on query recall scores and performance, we first show the recall@10 vs. query performance trade off results in Figure 2. The x-axis represents the recall score, and the y-axis represents the query throughput per second (qps score). Each data point on the same line represents a different query parameter combination.

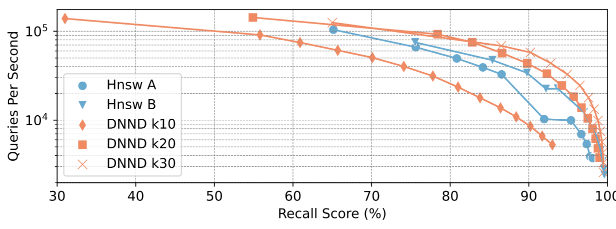
We submit all queries at once and process them in parallel (256 threads) for both implementations after loading all queries from a file. The query and ground truth datasets were obtained from the Big ANN Benchmarks website [19], Each query and ground truth dataset contains 10,000 queries and 10 ground truth nearest neighbors for each query point. The recall score is the ratio of computed nearest neighbor IDs that also exist in the corresponding ground truth. We report the mean recall score of all queries per data point.

Hnswlib results. We show two Hnswlib results for each DEEP 1B and BigANN dataset. Hnsw A and Hnsw C are the graphs that achieved similar or better graph quality than DNND k10 graphs with the minimum graph construction time. Hnsw B and Hnsw D are the best graphs we could achieve.

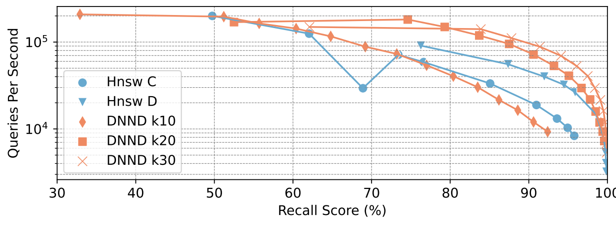
DNND results. We show each dataset's results of $k = 10$, $k = 20$, and $k = 30$. DNND was able to produce the same quality graphs regardless of the number of compute nodes used; thus, we do not show the results of changing the number of compute nodes. DNND achieved similar quality graphs to Hnswlib's best ones with $k20$. With $k30$, DNND produced better-quality graphs than Hnswlib.

5.3.4 k-NNG Construction Performacne. Finally, we show the k-NNG construction time in Figure 3 (thre raw numbers are in Table 3). The x-axis represents the number of compute nodes, and the y-axis represents the time to construct a k-NNG in hours. All axes are in the log scale. The graph construction step does not include the time to load the dataset from the files.

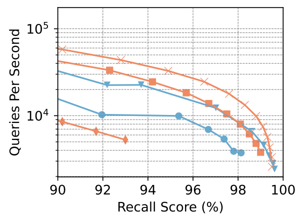
Hnswlib results. Hnsw A and Hnsw C are the cases that showed similar or higher quality than the DNND k10 graphs with the minimum construction time. Hnswlib took 5.9 hours and 1.7 hours,



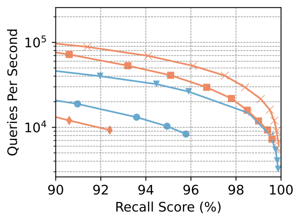
(a) Yandex DEEP 1B



(b) BigAnn



(c) Yandex DEEP 1B (recall score ≥ 90%)



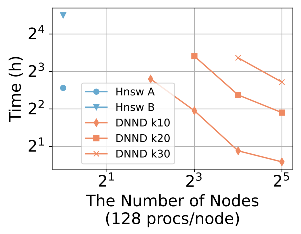
(d) BigAnn (recall score ≥ 90%)

Figure 2: Recall@10-query time trade-off on the k-NN graphs constructed in Figure 3. Figure 2c and Figure 2d are enlargements of the parts where the corresponding figures' recall scores are ≥ 90%.

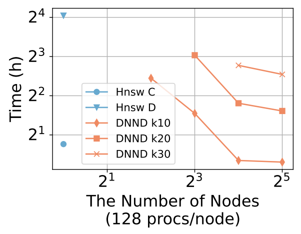
respectively, demonstrating very high k-NNG construction performance in those cases. On the other hand, Hnsw B and Hnsw D, which are similar to or less than DNND k20 cases, took 22.6 hours and 16.5 hours, respectively. Hnswlib could not construct graphs of higher quality than DNND k30 within 24 hours.

DNND results. Each result includes both k-NNG construction and optimization (generating reverse edges) steps. The optimization step accounted for less than 0.2% of the corresponding construction time in all cases. Opening and closing Metall data stores took negligible time; therefore, we did not include the time in the figures.

As for the DEEP 1B dataset with $k = 10$, DNND took 6.96 hours to construct a k-NNG on 4 compute nodes and showed high scalability until 16 nodes — taking 1.84 hours on 16 nodes, yielding a scaling factor of 3.8X. With $k = 30$, DNND was able to construct a k-NNG in 6.58 hours by using 32 nodes. The same trend was shown with BigAnn data, DNND took 1.24h, 3.05h, and 5.83h, respectively, to construct k-NNGs with k=10, 20, and 30 when using 32 nodes. By using 16 nodes, DNND (k20) was able to construct graphs at least similar quality to Hnsw B (DEEP 1B) and Hnsw D (BigAnn) at speeds 4.4 times and 4.7 times faster, respectively.



(a) Yandex DEEP 1B



(b) BigAnn

Figure 3: k-NNG construction time. Hnswlib (shared-memory) results were run on a single node using 256 threads.

Given the results, it is suggested that NN-Descent excels in constructing high-quality graphs, and DNND can address the increase in execution time and memory usage by utilizing distributed memory machines.

Table 3: The raw numbers of the data points in Figure 3

Number of nodes	1	4	8	16	32
Hnsw A	5.90	-	-	-	-
Hnsw B	22.60	-	-	-	-
DNND k10	-	6.96	3.87	1.84	1.50
DNND k20	-	-	10.62	5.18	3.74
DNND k30	-	-	-	10.29	6.58

(a) Yandex DEEP 1B

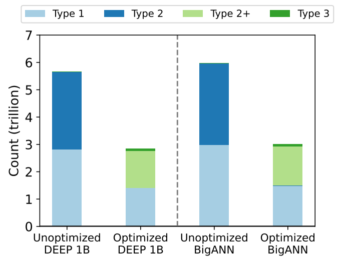
Number of nodes	1	4	8	16	32
Hnsw C	1.70	-	-	-	-
Hnsw D	16.50	-	-	-	-
DNND k10	-	5.45	2.92	1.27	1.24
DNND k20	-	-	8.19	3.50	3.05
DNND k30	-	-	-	6.84	5.83

(b) BigAnn

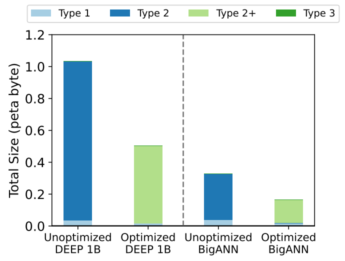
5.3.5 *Effects of Neighbor Checking Communication Saving Techniques.*

We collected the number of sent messages during the neighbor checks to evaluate the effects of the neighbor checking communication-saving techniques described in Section 4.3.

We show the number of messages that were sent when constructing k-NNGs for the two billion-scale datasets with $k = 10$ in Figure 4a. We also show the same results converted to the total message sizes in Figure 4b. Because we used uint8_t for BigAnn data's feature vector, BigAnn's message size is smaller than DEEP 1B's. For both datasets, the number of messages and the total message sizes were reduced by about 50%. Those results show that our proposed communication saving techniques are very effective in terms of both the number of messages and the communication data volume.



(a) The numbers of neighbor check messages.



(b) The total sizes of the neighbor check messages.

Figure 4: The effectiveness of our proposed communication-saving techniques described in Section 4.3. We corrected the data with $k = 10$ on 16 nodes. The unoptimized pattern sends Type 1 and Type 2 messages. The optimized pattern sends Type 1, Type 2+, and Type 3 messages.

6 RELATED WORK

To implement distributed-memory NN-Descent, the original NN-Descent paper [6] employed MapReduce [4]. Also, Warashina et al. [24] presented a performance-optimized version of that. Due to the design and implementation of MapReduce-type frameworks, we believe that implementing algorithm-specialized software, as DNND, produces much better performance. GNND [22] is a GPU-based NN-Descent (no distributed memory support). Leveraging GPU for local NN-Descent computing could be one of our future work directions.

A Hierarchical Navigable Small World (HNSW) graph is a hierarchical nearest neighbor index structure in which successive layers contain fewer points from the original dataset. Searches begin in the sparsest layer and end in a layer containing the full dataset, making finer adjustments to the nearest neighbors in each layer. This data structure is implemented by its creators in the Hnswlib library [17]. Pyramid extends the HNSW structure to distributed memory by partitioning data across processes and building a meta-HNSW on the partitions for use in distributing queries [5].

ELPIS [2] is a shared-memory ANNG library that utilizes Hercules [7] to split the given dataset into multiple groups using a tree structure, and, at the leaf level, ELPIS employs Hnswlib [17] to construct a graph for each divided dataset group. It demonstrates very fast index construction time and query performance. ELPIS is specialized for L_2 distance.

EFANNA [9] utilizes a dive-and-conquer technique to generate an initial k-NNG and performs the NN-Descent algorithm. Although EFANNA does not show better performance during the graph construction and the query over the original NN-Descent algorithm in another study [23], exploring their strategy for distributed-memory systems could be one of our future works.

As a method to speed up the query time of graph-based ANN, PyNNDescent divides data points using a random projection tree and selects the search's starting point based on this information. FINGER [3] proposed to speed up queries by introducing an approximate distance function calculation method.

7 FUTURE WORK

First, further performance profiling is required to identify bottlenecks, such as finding how much the computation or communication is heavier than the other and understanding communication patterns deeply. Based on the obtained information, leveraging GPU for accelerating local NN-Descent computing and specialized hardware for network communication are our interests.

Another future work would be exploring the further utilization of the persistent memory allocator. Employing Metall will facilitate rapid graph updates in the face of several real-world situations. For example, new data points may be added/deleted, followed by a short graph refinement phase, which will fit NN-Descent's iterative nature well. Also, several studies have shown that out-of-core processing could reduce memory usage while minimizing the performance overhead for large-scale graph processing and graph-based NN, for example [12, 14].

8 CONCLUSION

We developed a novel distributed memory implementation of NN-Descent leveraging HPC libraries and proposed a communication-saving technique. We adopted YGM, an asynchronous message library for distributed memory communication, and also utilized Metall, a persistent memory allocator, to enhance the usability of the constructed k-NN data.

When using the one billion-size datasets, our proposed k-NN graph construction communication-saving method reduced message counts and volume by approximately 50%. In contrast to Hnswlib, a state-of-the-art graph-based shared memory ANN library, DNND was able to construct a graph of similar or higher quality up to 4.7 times faster using 16 compute nodes. Furthermore, DNND demonstrated to construct even better quality graphs (k=30) in 6.6 and 5.8 hours using 32 nodes on Yandex DEEP 1B and BigAnn graphs, respectively. Given that Hnswlib does not produce a generic k-NNG different from DNND (NN-Descent), the outcomes of this study indicate that DNND can construct k-NNGs with high performance while maintaining the high accuracy and versatility of k-NNG, which is a strength of NN-Descent, by leveraging distributed memory systems. We achieved promising results toward developing a massive-scale k-NNG framework.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-PROC-852700).

REFERENCES

[1] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374. https://doi.org/10.1016/j.is.2019.02.006

[2] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1548–1559. https://doi.org/10.14778/3583140.3583166

[3] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2023. FINGER: Fast Inference for Graph-Based Approximate Nearest Neighbor Search. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 3225–3235. https://doi.org/10.1145/3543507.3583318

[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[5] Shiyuan Deng, Xiao Yan, KW Ng Kelvin, Chenyu Jiang, and James Cheng. 2019. Pyramid: A general framework for distributed similarity search on large-scale datasets. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 1066–1071.

[6] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 577–586. https://doi.org/10.1145/1963405.1963487

[7] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules against Data Series Similarity Search. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2005–2018. https://doi.org/10.14778/3547305.3547308

[8] Carlos Eiras-Franco, David Martínez-Rego, Leslie Kanthan, César Piñeiro, Antonio Bahamonde, Bertha Guijarro-Berdiñas, and Amparo Alonso-Betanzos. 2020. Fast Distributed KNN Graph Construction Using Auto-Tuned Locality-Sensitive Hashing. *ACM Trans. Intell. Syst. Technol.* 11, 6, Article 71 (oct 2020), 18 pages. https://doi.org/10.1145/3408889

[9] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. arXiv:1609.07228 [cs.CV]

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

[11] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 518–529.

[12] Keita Iwabuchi, Scott Sallinen, Roger Pearce, Brian Van Essen, Maya Gokhale, and Satoshi Matsuoka. 2016. Towards a Distributed Large-Scale Dynamic Graph Data Store. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 892–901. https://doi.org/10.1109/IPDPSW.2016.189

[13] Keita Iwabuchi, Karim Youssef, Kaushik Velusamy, Maya Gokhale, and Roger Pearce. 2022. Metall: A persistent memory allocator for data-centric analytics. *Parallel Comput.* 111 (2022), 102905. https://doi.org/10.1016/j.parco.2022.102905

[14] 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. https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[15] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[16] Herve 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

[17] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[18] PyNNDescent. [n. d.]. GitHub - lmcinnes/pynndescent: A Python nearest neighbor descent for approximate nearest neighbors — github.com. https://github.com/lmcinnes/pynndescent. [Accessed 24-Jun-2023].

[19] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Artem Babenko, Dmitry Baranchuk, Qi Chen, Matthijs Douze, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. [n. d.]. Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. http://big-ann-benchmarks.com/neurips21.html. [Accessed 30-Jun-2023].

[20] Trevor Steil, Tahsin Reza, Keita Iwabuchi, Benjamin W. Priest, Geoffrey Sanders, and Roger Pearce. 2021. TriPoll: Computing Surveys of Triangles in Massive-Scale Temporal Graphs with Metadata. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 67, 12 pages. https://doi.org/10.1145/3458817.3476200

[21] Trevor Steil, Tahsin Reza, Benjamin W. Priest, and Roger Pearce. 2023 (to appear). Embracing Irregular Parallelism in HPC with YGM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '23)*. Association for Computing Machinery, New York, NY, USA.

[22] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. *Fast K-NN Graph Construction by GPU Based NN-Descent.* Association for Computing Machinery, New York, NY, USA, 1929–1938. https://doi.org/10.1145/3459637.3482344

[23] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1964–1978. https://doi.org/10.14778/3476249.3476255

[24] Tomohiro WARASHINA, Kazuo AOYAMA, Hiroshi SAWADA, and Takashi HATTORI. 2014. Efficient K-Nearest Neighbor Graph Construction Using MapReduce for Large-Scale Data Sets. *IEICE Transactions on Information and Systems* E97.D, 12 (2014), 3142–3154. https://doi.org/10.1587/transinf.2014EDP7108

[25] YGM. [n. d.]. GitHub - LLNL/ygm — github.com. https://github.com/LLNL/ygm. [Accessed 28-Jun-2023].