Fast Scalable Approximate Nearest Neighbor Search for High-dimensional Data

Renga Bashyam K G
Department of Computational and Data Sciences
Indian Institute of Science
Bengaluru, India
rengak@iisc.ac.in

Sathish Vadhiyar
Department of Computational and Data Sciences
Indian Institute of Science
Bengaluru, India
vss@iisc.ac.in

***Abstract*—K-Nearest Neighbor (k-NN) search is one of the most commonly used approaches for similarity search. It finds extensive applications in machine learning and data mining. This era of big data warrants efficiently scaling k-NN search algorithms for billion-scale datasets with high dimensionality. In this paper, we propose a solution towards this end where we use vantage point trees for partitioning the dataset across multiple processes and exploit an existing graph-based sequential approximate k-NN search algorithm called HNSW (Hierarchical Navigable Small World) for searching locally within a process. Our hybrid MPI-OpenMP solution employs techniques including exploiting MPI one-sided communication for reducing communication times and partition replication for better load balancing across processes. We demonstrate computation of k-NN for 10,000 queries in the order of seconds using our approach on ∼8000 cores on a dataset with billion points in an 128-dimensional space. We also show 10X speedup over a completely k-d tree-based solution for the same dataset, thus demonstrating better suitability of our solution for high dimensional datasets. Our solution shows almost linear strong scaling.**

***Index Terms*—K-NN Search, Parallel Algorithms, Load Balancing, Vantage Point Tree, HNSW**

I. Introduction

In this era of big data, the amount of information at hand now is more than ever. Querying such large-scale information requires scalable distributed data structures for similarity search. One of the most commonly used approaches for similarity search is the k-Nearest Neighbor (k-NN) search. k-NN search returns K elements from a dataset that minimize a distance metric to a given query. More formally, given a dataset D from a metric space $\mathcal{S}$ with metric $\mathcal{M}$ and query $q \in \mathcal{S}$, a k-NN search algorithm returns k points from D which are closest to q with respect to $\mathcal{M}$. K-NN search finds extensive applications in machine learning and data mining as a classification and regression method. It is also used in scientific applications for prediction of protein interactions, in cosmology and particle and plasma physics [1]. Many popular implementations of different k-NN search methods like FLANN [2], FAISS [3], Annoy [4], NGT [5] and HNSW [6] are available. All of these are shared memory solutions that support multi-threaded executions using OpenMP. FAISS supports GPU executions using CUDA. For billion-scale datasets, the memory becomes a bottleneck when the entire dataset cannot be loaded in a single machine, and hence distributed systems need to be considered for such datasets. PANDA [1] builds a distributed KD tree to serve as a distributed index for the k-NN search. This is an exact search method that suffers considerable performance degradation as the dimension of the dataset increases.

Our work aims at developing a distributed graph-based index for k-NN search in order to facilitate effective querying on high-dimensional billion-scale datasets. Our approach achieves increased throughput in terms of the number of queries processed per unit time. This can be useful when queries need not be answered in real time and can be batched together like in recommender systems and when k-NN is used as a classifier. Our method involves partitioning the search space using a tree-based approach until the number of data points in each individual partition becomes small enough that indexing those points can be restricted to a single thread or process. We use vantage point (VP) trees to achieve this space partitioning as they generally offer better search pruning than KD trees irrespective of the metric employed [7]. The part of the data assigned to a process is then indexed for search using HNSW algorithm, an approximate k-NN search method, which scales well with dimension and is the current state-of-art sequential method [6]. When processing a batch of queries, each query can be localized to a subset of partitions and increased throughput is achieved by processing multiple queries on their respective partitions at the same time. Our solution, implemented as a hybrid MPI-OpenMP framework, also employs optimisations including load balancing by partition replication and MPI one-sided communication to reduce time spent on communication synchronisation.

Our experiments demonstrate computation of 10-NN for 10,000 queries in the order of seconds using our approach on ∼8000 cores on a dataset with billion points in a 128-dimensional space. We also show 10X speedup over a completely k-d tree-based solution for the same dataset, thus demonstrating better suitability of our solution for high dimensional datasets. Our solution shows almost linear strong scaling. Our results also show that load balancing by partition replication leads to reduced execution times. To our knowledge, ours is the first work showing almost linear scalability

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00040

for such large high-dimensional datasets.

The paper is organized as follows. Section II describes works in the literature relating to the parallel k-NN search. In section III, we provide background on HNSW graphs and VP trees, the two major data structures used in our approach. In section IV, we describe our approach in detail. Section V presents our experiments and results.

II. Related Work

Classic methods to solve k-NN search include constructing tree-based indexes [7] (KD-trees, VP-trees, ball trees, etc) and spatial approximation [8]. Exact solutions to the k-NN search do not scale well with the dimension of the dataset. To overcome this, approximate methods have been proposed. These methods include locality-sensitive hashing [9], product quantization [10] and proximity graph techniques [11]. Proximity graph techniques scale well with dimension and hence recent research efforts have been focused on them. One such proximity graph technique is Hierarchical Navigable Small World (HNSW) graph indexing [6].

There have been many efforts on the k-NN search for multi-core and GPU architectures. FLANN provides multi-threaded searching on hierarchical k-means trees and randomized KD trees [2]. FAISS provides GPU support for building and searching inverted file (IVF) index with and without product quantization [3]. HNSW supports multi-threaded index construction and searching [12]. NGT constructs k-NN graphs as indices for the k-NN search and provides multi-threaded graph search heuristics for answering k-NN queries [5]. These single-node implementations cannot handle BigData that arise in practical problems. Annoy provides for building compact indices that can be easily moved around as static files and shared across processes [4].

Endeavors towards building compressed search indices for billion-scale datasets that fit into single-node memory are also described in the literature. Reference [13] employs an inverted file index (IVF) with a large codebook of centroids and an HNSW index of those centroids to limit query assignment to a subset of centroids. Reference [14] introduces polysemous codes that are used together with an inverted multi-index (IMI) to index and query a billion-scale dataset in sub-millisecond time per core. Compressed indices suffer from poor search recall and recall usually plateaus after a while when increasing the quality of the compressed index.

Reference [15] proposes GRIP, a multi-store approximate k-NN search algorithm that builds a two-layer index spanning both the memory and the disk. The first layer uses HNSW and product quantization to build an index that can be fit into memory. This is used to fetch r nearest neighbors. The second layer consists of the actual dataset with full-precision vectors which is used to validate the r ($> k$) nearest neighbors fetched from the first layer using their true distances from the query. Thus, k nearest neighbors are picked from the r nearest neighbors returned from the first layer. GRIP achieves competitive search latencies with very low memory consumption and high recall but suffers from resource limitations presented by single-node systems.

Multiple approaches have been proposed for distributed k-NN search. The paper by Patwary et al. [1] implements PANDA which uses a distributed KD tree for efficient k-NN search on billion-scale datasets. Four levels of parallelism are exploited in this work. At the coarse level, a global KD tree is constructed by moving points across processors. Once a KD sub-tree can be fit into a processor, data parallelism is used within a processor to construct subsequent levels of the tree. Once there are enough tree-nodes in a level, thread-level parallelism is used wherein each thread has its own local KD sub-tree. The last level of parallelism is achieved by using SIMD optimised buckets to speed-up distance computations. Their implementation constructed a KD tree of 189 billion particles in 48 seconds by utilizing around 50,000 cores. They were also able to demonstrate the computation of k-NN for 19 billion queries in 12 seconds. This approach does not scale for high dimensional datasets. In higher dimensions, the number of tree-nodes and hence processors visited by the k-NN search routine explodes. The highest dimension experimented in the work was ten.

Reference [16] proposes a two-layer indexing scheme for building a distributed index based on neighborhood graphs for large scale k-NN search. A flat randomized partitioning scheme is used to partition the search space into subspaces by selecting *pivots* from the dataset which causes significant load imbalance across processes. Every data point is assigned to the subspace with its closest pivot. Neighborhood graphs are built locally within every subspace to answer to k-NN queries within that subspace. A neighborhood graph is also constructed for pivots. To answer a query globally, the pivot neighborhood graph is used to obtain the most promising subspaces on which local search is performed. On a dataset with billion points, their implementation takes $\sim$240 milliseconds per query using 64 cores. Our work has demonstrated 8X improvement over this method.

III. Background

A. Hierarchical Navigable Small World (HNSW) Graphs

Reference [6] proposed the Hierarchical Navigable Small World (HNSW) algorithm for approximate k-NN search. This is a graph-based approach to construct an index for the k-NN search wherein points of the dataset D are taken to be nodes of a multi-layered graph G and these nodes are connected in such a way that a k-NN search on D becomes a *greedy search* on G. A greedy search on a graph is one that is localized and where the next vertex to jump to is decided only by the neighbors of the current vertex.

HNSW builds on Navigable Small World (NSW) graphs [12]. HNSW graph consists of layers of NSW graphs. The bottom layer consists of all the points in D. Each point is "promoted" to the layer above independently with a fixed probability, very similar to how keys are promoted in the skip-list data structure [17]. An NSW graph is constructed on the promoted points on the above layer. This is repeated until no

points are promoted from the current layer. The current layer then forms the top layer of HNSW.

Search in an HNSW graph starts from the top layer. A greedy search is performed on the current layer and the results of this search are used to begin the search in the layer below. Results of the greedy search in the bottom layer are returned finally as the k nearest neighbors. Introducing this hierarchy in HNSW allows for a time complexity of $O(log|D|)$ time for the search whereas searching in NSW graphs takes $O(log^2|D|)$ time. HNSW is the current state-of-art for sequential k-NN search. In this project, we try to exploit its speed for local computation within a data partition after partitioning the dataset and distributing it efficiently to reduce the query time for large datasets. The theoretical foundations of the ability of the HNSW to give a good approximation of the exact search lies in the fact that the HSNW graph that is constructed provides an approximation of the graph that has both navigable (Delaunay graph) [18] and small-world [19] properties.

B. Vantage Point (VP) Trees

The work by Yianilos [7] introduces the Vantage Point (VP) tree data structure for nearest neighbor search in metric spaces and outlines its construction and search routines. VP trees are space partitioning tree data structures used to build indices for similarity search in metric spaces.

A VP tree shares similarities with the popular data structure KD tree. Similar to a KD tree, each node in a VP tree partitions the space. While KD trees use coordinate values, VP trees use distance from an arbitrary point called the "vantage point" to partition a space. A distance d is selected such that half the points in the dataset are within distance d from the vantage point and half the points are outside that distance. Then, the sphere of radius d around the vantage point is chosen as the left subspace forming the left child of the current node, and the complementary subspace is chosen as the right subspace forming the right child. This proceeds recursively forming a binary tree.

K-NN search in a VP tree starts at the root and proceeds recursively to the child nodes. If at a node N, the current nearest neighbor is at a distance d from query q, only the subspace in the sphere of radius d around q is searched. If the sphere lies entirely within the left or the right subspace corresponding to node N, the other subspace can be pruned out from searching. If the sphere intersects both the subspaces, the search has to proceed in both the child nodes and no pruning is done. Hence searching can reach more than one leaf of a VP tree.

Yianilos [7] proves that the average search complexity is logarithmic in the size of the dataset. Their experiments show that VP trees scale better with dimension than KD trees in terms of search pruning and that VP trees are metric-agnostic, whereas KD trees perform poorly for metrics other than L_2 and L_∞. We use VP trees to recursively partition the search space until data points within a partition can be accommodated in a single processor. The leaves of the VP tree we construct will be a set of data points rather than a single point as in a conventional VP tree.

IV. Methodology

Let $\mathcal{P} = \{p_1, p_2, ..., p_P\}$ be the P processing cores available. The dataset D is initially equi-partitioned into P equal partitions D_1, D_2, ..., D_P, so that the compute node with processor p_i gets the partition D_i. Let $L(D_i, q)$ represent the result of local k-NN search for the query q in the partition D_i. Let $\mathcal{F} : S \mapsto 2^D$ be a function that gives the subset of partitions for a given query such that local results from these partitions are sufficient to reconstruct the global nearest neighbors for that query.

We implement a master-worker strategy in which a master process computes $\mathcal{F}(q)$ for every query q and identifies the processor cores containing the partitions in $\mathcal{F}(q)$ and on which a query should be executed. The master then sends the query to the compute nodes with those processor cores. Any of the available processes in a compute node then work independently to find $L(D_i, q)$ for its queries simultaneously, thus achieving increased throughput. Here, D_i refers to the partition in the compute node. Any process in a compute node can be used for finding $L(D_i, q)$ because the partitions in a compute node reside in shared memory. We do not strongly couple a process core p_i with data partition D_i in its compute node. This way load balance is achieved between the processor cores in a node. Once a processor completes a query, it sends the results back to the master process.

Our hybrid MPI-OpenMP implementation uses the VP tree data-structure to calculate $\mathcal{F}(q)$ at the master process and HNSW algorithm in the individual processors to compute $L(D_i, q)$ locally. Figure 1 provides an architectural overview of the proposed solution. All communications are non-blocking and hence asynchronous to achieve maximum overlap with computations.

We also implemented a multiple-owner strategy where the VP tree built is shared by all the processes to compute $\mathcal{F}(q)$. The owner of a query is determined by a hash function. We saw a small improvement in search time over an optimized master-worker strategy but this improvement deteriorated as core count increased. This is because a multiple-owner strategy does not lend itself to be optimized for load balancing across data partitions that arises from VP tree partitioning. We present an optimization to counter load imbalance in the master-worker strategy in section IV-C2

A. Parallel VP Tree Construction

To construct the vantage point (VP) tree sequentially at the master process, we need to load the entire dataset which is not possible for billion-scale datasets. We have come up with a distributed implementation of VP-tree construction to overcome this.

In our distributed construction, all of the processes are involved in constructing the root of the VP tree. After the root is constructed, the left child is built by one-half of the processes, and the right child is built by the other half. This is

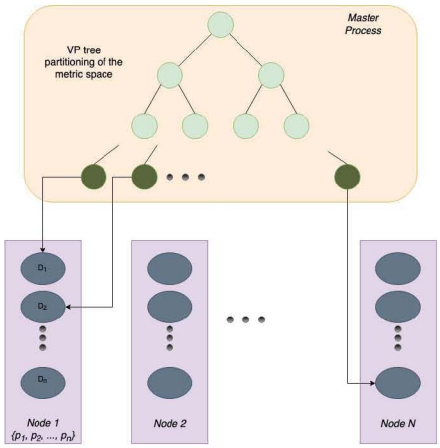


Fig. 1. Architectural overview of the proposed solution for increasing throughput. Every leaf node of the VP tree in master process corresponds to one partition D_i of the dataset. The data partition is sent to the compute node with processor core p_i. Hence, compute node 1 with processor cores $\{p_1, p_2, ..., p_n\}$ gets the data partitions $\{D_1, D_2, ..., D_n\}$.

repeated recursively for the child nodes. Construction of every tree node in a level is equally split among the processes. One of the processes involved in the construction of a root node of a subtree is designated as the master process during construction.

The algorithm for the sequential VP tree construction is defined in [7]. At every level, the algorithm selects a vantage point by a heuristic that maximizes the quality of pruning during the search. The heuristic works by randomly sampling a small subset of points D' from the dataset D as the candidate set and selecting that point $v \in D'$ which maximises a function $\mathcal{H}(v, D)$. This function computes the second moment of the distances of the points in D to v about the median of these distances. We will denote this heuristic as `SelectVantagePointSerial(`D'`, `D`)`.

For selecting the vantage point, each process runs `SelectVantagePointSerial()` routine on its subset of data after sampling the candidate set from the subset itself and sends its representative to the master. The master process runs the same routine again on its subset of data with the received representatives to select the vantage point. The assumption we make here is that each subset local to a process is representative of the global data distribution. The vantage point is then broadcast to all the processes. Now, all the processes involved in tree node construction agree on the vantage point. This distributed vantage point selection is presented in Algorithm 1.

The next step involves finding the radius μ of the sphere around the vantage point which equipartitions the dataset. Towards this end, processes compute the distance of each of the points in their subset of data with the vantage point. The median of these distances will give μ. This is found using a distributed version of the median of median algorithm. This is followed by the shuffling of data between processors using `MPI_AlltoAllv` routine. Half the number of processors handle the left sub-tree and the other half the right sub-tree.

Algorithm 1 Distributed Vantage Point Selection

procedure SELECTVANTAGEPOINT($D, M, P_1...P_N$)
  ▷ D is the dataset, M is the master, $P_1...P_N$ are worker processes
  ▷ D is randomly partitioned across $P_1...P_N$
  for $p \in P_1...P_N$ in parallel **do**
    $d \leftarrow$ Subset of dataset D with p
    $s \leftarrow$ Randomly sample 100 elements from d
    `Candidate` $\leftarrow$ SELECTVANTAGEPOINTSEARIAL(s, d)
    Send `Candidate` to master
  end for
  if master **then**
    Receive candidate set C from workers
    $d \leftarrow$ Subset of dataset D with master
    `Vantage point` $\leftarrow$ SELECTVANTAGEPOINTSEARIAL(C, d)
  end if
end procedure

This process continues recursively until every processor is assigned its partition. This process is presented in Algorithm 2.

Algorithm 2 Distributed VP Tree Construction

procedure CONSTRUCTNODE($D, P_1...P_N$)
  ▷ D is the dataset, M is the master, $P_1...P_N$ are worker processes
  ▷ D is randomly partitioned across $P_1...P_N$
  Initialize node N ▷ N is the node to be returned
  $N.vp \leftarrow$ SELECTVANTAGEPOINT($D, P_1...P_N$)
  $N.\mu \leftarrow Median_{p \in D}\mathcal{M}(N.vp, p)$
  ▷ Use median of medians algorithm
  $D_L \leftarrow$ Points in D within radius $N.\mu$ of $N.vp$
  $D_R \leftarrow D - D_L$
  Move D_L to $P_1...P_{N/2}$
  Move D_R to $P_{N/2}...P_N$
  ▷ Using `MPI_Alltoallv()`
  $N.left \leftarrow$ CONSTRUCTNODE($D_L, P_1...P_{N/2}$)
  $N.right \leftarrow$ CONSTRUCTNODE($D_R, P_{N/2}...P_N$)
  return N
end procedure

B. Searching

While searching, one worker process is spawned per compute node. A worker process spawns multiple OpenMP threads to handle local queries. The master process sends a query to the worker processes that contain the VP tree partitions where the search has to be performed for the query. Once it has dispatched every query as required, it sends a `End of Queries` command to every worker process and waits for the responses from the worker processes. When the master receives a local k-NN result from a worker for a query, it

updates the global k-NN result for the query. The search routine in the master process is shown in Algorithm 3.

Algorithm 3 Search Routine (Master)

```
1: procedure SEARCH(Q)                          ▷ Q is the query set
2:     for q in Q do
3:         D_q ← Partitions to be searched for q
4:                                              ▷ From VP tree
5:         for d in D_q do
6:             Send (q, d) to d's process
7:                                              ▷ Using MPI_Isend()
8:             Start listening for response from d's process
9:                                              ▷ Using MPI_Irecv()
10:        end for
11:    end for
12:    for every worker processor p do
13:        Send End of Queries command to p
14:    end for
15:    while Listening for response do
16:        Get Results of query q in partition d
17:        Update q's final results
18:    end while
19: end procedure
```

Algorithm 4 Search Routine (Worker)

```
1: procedure SEARCH
2:     Spawn a set T threads
3:     Done ← false
4:     for every thread t ∈ T in parallel do
5:         while True do
6:             Receive message M from master
7:                                              ▷ Using MPI_Irecv()
8:             while Message M not received do
9:                                              ▷ Using MPI_Test()
10:                if Done is true then
11:                    Terminate execution
12:                end if
13:            end while
14:            if M is End of Queries command then
15:                Done ← true
16:                Terminate execution
17:            end if
18:            Get query q and partition d from M
19:            Search partition for q
20:            Send results to master
21:                                             ▷ Using MPI_Send()
22:        end while
23:    end for
24: end procedure
```

The worker processes spawn a fixed number of threads at the start. Each thread waits for a message from the master. If it receives a message with a query, it proceeds to perform the HNSW search on the partition specified in the message. If it receives a `End of Queries` command, it signals other threads to terminate by setting a shared flag and terminates. While waiting for messages, threads check to see if they should terminate by accessing the shared flag. If they find it to be set, they cancel the message receive operations, and terminate. Algorithm 4 present the search routine in worker processes.

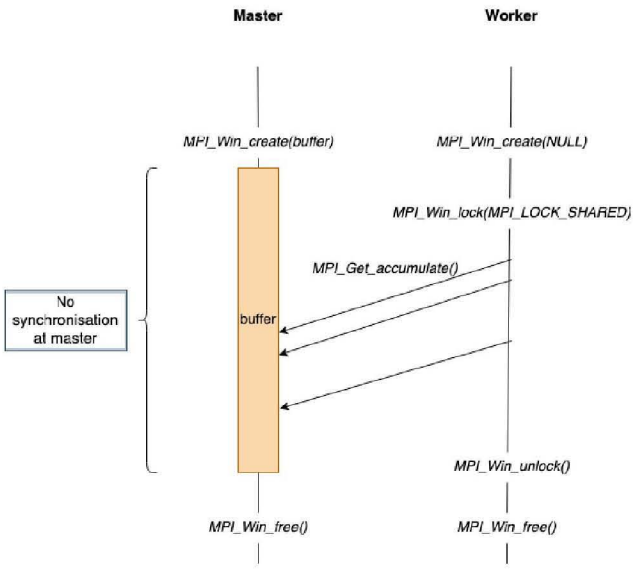


Fig. 2. MPI One-Sided Communication between the master and a worker.

Load imbalance in the number of queries run on a data partition arises across data partitions depending on the query set. The proposed search method achieves dynamic work assignment to threads in a compute node for answering queries and thus helps alleviate load imbalance within a compute node.

C. Optimisations

1) MPI One-Sided Communication: Preliminary results from baseline implementation showed that there was a scalability bottleneck when worker processes send their results back to the master. This is because the master spends considerable time receiving responses from the workers. Moreover, the communication pattern is highly irregular. MPI one-sided communication can be used to overcome these bottlenecks on systems supporting remote memory access over a specialised network interconnect.

The master process exposes a part of its memory for storing the global k-NN results for every query. The workers update the part of this memory corresponding to the query they have processed with their local results. The workers use a series of atomic remote memory read-update operations to achieve this. We exploit passive target synchronisation by using `MPI_Win_lock()` in shared mode on the worker processes. The worker processes use `MPI_Get_accumulate()` to write their results to the remote memory window of the master. This optimisation works better for small k values.

Figure 2 illustrates how the master process and worker processes exploit MPI one-sided communication primitives to aggregate results of local k-NN searches. All the processes call the MPI collective primitive `MPI_Win_create()`, but only the master specifies a buffer allocated in its memory. Then all the worker processes invoke `MPI_Win_lock()` to begin

an RMA (Remote Memory Access) access epoch at the target process in the shared mode which informs the MPI library that multiple processes are accessing the master process's memory simultaneously.

The worker processes then begin writing results of their local search to the master process's buffer as and when they have completed local searches for a query using `MPI_Get_accumulate()` calls, which are atomic. The memory location at which a worker process has to write a result is determined by the ID of the corresponding query. It should be noted that synchronization primitives need not be used at the master.

Algorithm 5 Search Routine (Master) with Load Balancing

```
1: procedure SEARCH(Q)                          ▷ Q is the query set
2:     for i in {1, 2, ..., P} do
3:         W_i ← {p_i, p_{(i+1)modP}, ..., p_{(i+r−1)modP}}
4:         W_i.next ← p_i
5:     end for
6:     for q in Q do
7:         D_q ← Partitions to be searched for q
8:                                              ▷ From VP tree
9:         for d_i in D_q do
10:            p ← W_i.next
11:            W_i.next ← Next element in W_i
12:            Send (q, d_i) to p's process
13:                                    ▷ Using MPI_Isend()
14:            Start listening for response from p's process
15:                                    ▷ Using MPI_Irecv()
16:        end for
17:    end for
18:    for every worker process p do
19:        Send End of Queries command to p
20:    end for
21:    while Listening for response do
22:        Get Results of query q in partition d
23:        Update q's final results
24:    end while
25: end procedure
```

2) Load Balancing: Though the baseline described so far offers load balancing within a compute node, load imbalance across compute nodes allows room for more improvement. To this end, the P processing cores are logically grouped into P workgroups. The number of processing cores in each workgroup is defined as the replication factor. For a replication factor r, the workgroup W_i has the processing cores $\{p_i, p_{(i+1)modP}, ..., p_{(i+r-1)modP}\}$. Each data partition is replicated on every processing core in its corresponding workgroup. A compute node loads all data partitions corresponding to workgroups its processing cores belong to.

The master process maintains a circular list for each workgroup, containing the processing cores of that workgroup along with a next pointer. Instead of dispatching a query to its processing core, the master process now dispatches a query to its workgroup in a round-robin fashion, i.e, it dispatches the query to the processing core pointed by the next pointer in the workgroup's circular list and updates the next pointer to point to the next processing core in the circular list. This modified search routine at the master process is depicted in Algorithm 5. This approach distributes the workload more evenly across processing cores at the cost of incurring more memory in each compute-node to store additional data partitions.

TABLE I
DATASETS USED IN OUR EXPERIMENTS

Dataset	No. of points	Dimension	No. of queries used	Ground truth available
ANN_SIFT1B	1 billion	128	10000	Yes
DEEP1B	1 billion	96	10000	Yes
ANN_GIST1M	1 million	960	1000	Yes
SYN_1M	1 million	512	10000	No
SYN_10M	10 million	256	10000	No

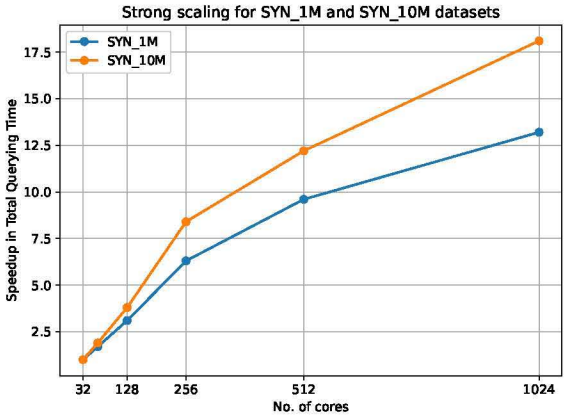
V. EXPERIMENTS AND RESULTS

A Cray XC40 system in our Institute was used to run our experiments. The system has 1376 compute nodes and each compute-node has two CPU sockets with 12 Intel Xeon Haswell (@2.5 GHz) cores each and 128 GB of memory. The compute nodes are connected using Cray Aries interconnect. Cray Clang compiler (version 9.0.2) was used to compile our code with Cray MPICH (version 7.7.10) library.
[1]

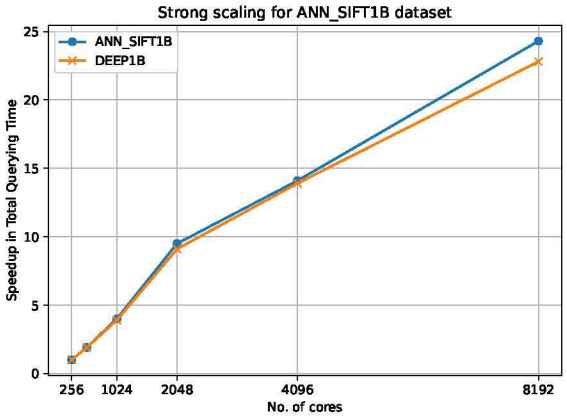
Table I lists the datasets we have used in our experiments. `ANN_SIFT1B` [20] and `ANN_GIST1M` [10] were generated by extracting SIFT and GIST image descriptors, respectively, from image datasets. These datasets also provide a query set and k-NN ground truth for that set that can be used for estimating the accuracy of approximate search methods. `DEEP1B` [21] is a similar dataset of image descriptors, but produced by a convolutional neural network (CNN). `SYN_1M` and `SYN_10M` were generated using MDCGen [22]. For both the `SYN_1M` and `SYN_10M` datasets, we use Gaussian and uniform distributions to generate points in 10 clusters. The number of outliers were set to 5000 and 50000 in `SYN_1M` and `SYN_10M`, respectively. The rest of the parameters were set to default values. Query sets for these datasets were generated using uniform distribution in a single cluster with a compactness factor of 0.01.

In the following subsections, we present the strong scaling characteristics of our method for querying times on different datasets, time taken for distributed index construction on `ANN_SIFT1B` dataset, the effect of replication factor for load balancing and comparison of our approach with a KD tree-based approach. In all of the experiments, the value of k used for the k-NN querying is 10 and the distance metric used is L_2 norm. Most of the results were collected by obtaining an average over five runs.

[1]Our code is available at https://github.com/renshyam/fast_ann



(a) Scalability results SYN_1M and SYN_10M Datasets



(b) Scalability results with ANN_SIFT1B Dataset

Fig. 3. Strong Scaling Results with different Datasets. Speedups are normalized to time taken on 32 cores for SYN_1M and SYN_10M datasets and to that on 256 cores for ANN_SIFT1B and DEEP1B datasets.

A. Strong Scaling

Figure 3 shows the strong scaling results for the total query time for the different number of cores. The graphs give speedups of the querying times when compared to the times taken on 32 cores for the 1 million and 10 million datasets, and when compared to the times taken on 256 cores for the 1 bilion dataset. The executions correspond to our hybrid MPI-OpenMP code with one-sided communications and without load balancing.

From Figure 3(a), we find that the speedups obtained for 1024 cores are about 13 and 18 for the SYN_1M (1 million) and SYN_10M (10 million) datasets, respectively when compared to the executions on 32 cores. From Figure 3(b), we find that we obtain a speedup of about 25 for both the 1-billion datasets of ANN_SIFT1B and DEEP1B on 8192 cores when compared to the execution on 256 cores. The results for the 1-billion datasets almost exhibits linear scalability. Thus our parallel algorithms are suitable for large and multi-dimensional datasets. Note that in Figure 3(b), speedup curves obtained for both the billion-scale datasets are similar.

B. Construction

Table II shows the times taken for our parallel algorithm for VP tree and local HSNW graph constructions for the different number of cores for the 1 billion SIFT dataset. We find that the total construction times show good scalability with the increasing number of cores. The table also shows that there is a large reduction in the time for HSNW construction, the primary core of the construction, for large parallelism.

TABLE II
CONSTRUCTION TIMES FOR ANN_SIFT1B

No. of cores	Total Time (minutes)	HNSW Construction (minutes)
256	21.5	17.6
512	20.1	14.8
1024	18.3	12.4
2048	16.5	9.8
4096	15.2	7.8
8192	14.7	4.3

C. Effect of Replication Factor on Load Balancing

Our load balancing optimized algorithm replicates the partitions to different processors and assigns the queries for a partition in a round-robin manner across the processors on which the data is replicated. The algorithm employs a replication factor to determine the number of processors on which a partition is replicated.

Figure 4 shows the effect of the replication factor on the load balancing and the execution times for the SIFT 1 billion datasets when executed on 8192 cores. The replication factor of 1 corresponds to the base algorithm without replication. Figure 4(a) shows the total query times while Figure 4(b) shows the distribution of the number of queries sent to the processors for the different replication factors.

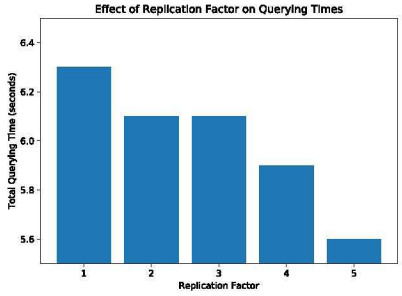
We find from Figure 4(a) that our replication-based load balancing gives a definite performance improvement over the baseline algorithm without replication. The performance improvement also increases with increasing replication factor achieving up to performance improvement of 11% for the replication factor of 5. The performance improvement is due to the balanced processing of the queries as shown by Figure 4(b). We find that as the replication factor increases, the range of the number of queries processed by the different processors becomes more compact.

D. Comparison with existing methods

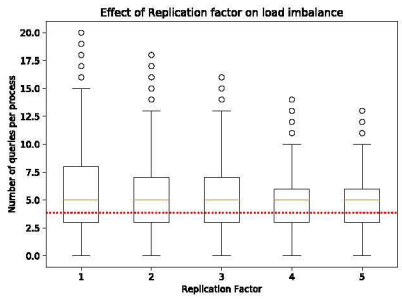
TABLE III
TOTAL SEARCH TIMES

Dataset	Total Query Time (seconds)	
	Our method	KD-Tree [1]
ANN_SIFT1B (8192 cores)	6.3 (13.6X faster)	85.6
DEEP1B (8192 cores)	7.1 (11.4X faster)	80.9
ANN_GIST1M (24 cores)	0.54 (8.5X faster)	4.6

Table III presents search times for our method and the method described in Patwary et al. [1] for different datasets.



(a) Total Querying Times for Different Replication Factors



(b) Distribution of the Number of Queries Processed by the Processes for Different Replication Factors

Fig. 4. Load Balancing For Replication Factors for ANN_SIFT1B dataset. The red dotted line in 4(b) indicates the number of queries per process for optimal load balance.

We implemented the method described in [1] and compared its performance with our approach on ANN_SIFT1B and DEEP1B for 8192 cores. To our knowledge, this is the only other work that builds an uncompressed index for billion-scale datasets. It should be noted here that distributed KD trees give exact results for k-NN queries. Hence we measured the accuracy of our method using the recall metric. Recall is defined as the ratio of the number of true k-nearest neighbors in the result of the approximate search to k. Our approach was 13.6 times faster in answering 10^4 queries with an average recall of 0.88 for ANN_SIFT1B and 11.4 time faster in answering the same number of queries with an average recall of 0.85 for DEEP1B. On ANN_GIST1M, our approach was 8.5 times faster in answering 10^3 queries with an average recall of 0.91 on 24 cores. These results clearly demonstrate the superiority of HNSW with VP tree partitioning over a complete KD tree approach for searching in a high-dimensional distributed index.

E. Search Time Breakdown

Figure 5 shows the breakdown of the total time for searching 10^4 queries on ANN_SIFT1B for the different number of cores. It can be seen that MPI-based communications occupy only a small percentage of the overall time and the computation-communication times are greater than 90% in many cases. The small communication times is due to the use of non-blocking communications by the master and the

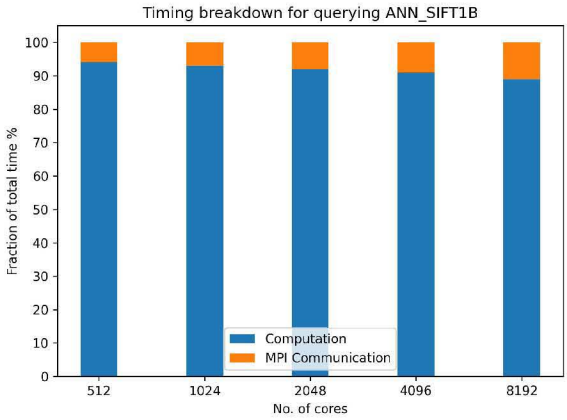


Fig. 5. Search Time Breakdown

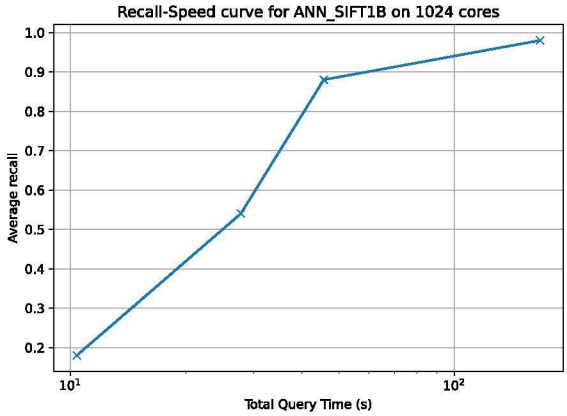


Fig. 6. Search Recall plotted against Total Query Time for ANN_SIFT1B on 1024 cores.

asynchronous MPI one-sided communications for accumulation of the results by the workers to the memory of the master process.

F. Search Recall

During the construction of the HNSW index, the parameter M is used to specify the number of neighbors of a newly inserted vertex in a layer. The trade-off between search time and recall in HNSW is controlled by the M parameter. Figure 6 presents search recall plotted against the total time taken for querying for ANN_SIFT1B on 1024 cores for the following values of M: $\{8, 16, 32, 64\}$ (the default value of M is 16). It should also be noted that higher value of M leads to more memory consumption. When M is 64, we achieve near perfect recall while answering 10^4 queries in 167 seconds. Compression methods [13] [14], even though capable of building an index for billion-scale datasets that can be fit into the memory of a single node and perform search faster, cannot achieve near perfect recalls.

VI. Conclusion and Future Work

We have presented a scalable distributed solution for performing an approximate k-NN search that is well-suited for high-dimensional data in general metric spaces. We have proposed two optimizations for better query performance, namely the use of MPI one-sided communications and load balancing using replication. Our experiments show that our approach scales almost linearly with the number of cores on two standard billion-scale high-dimensional datasets, `ANN_SIFT1B` and `DEEP1B`. We show around 10X speed-up over a completely KD tree-based approach on the same datasets. Our parallel index construction shows scalability and competitive build times. Optimizing for load imbalance by partition replication gives 11% performance improvement. We also demonstrate that it is possible to achieve near perfect recalls on billion-scale datasets.

Our approach is extensible in that any algorithm can be used for local indexing and searching instead of HNSW. We can utilise the parallelism offered by GPUs to perform local searching. Building a GPU kernel for a graph-based search method like HNSW is an interesting avenue that is yet to be explored. We can also exploit CPU and GPU cores simultaneously to answer local queries.

References

[1] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, P. Dubey *et al.*, "Panda: Extreme scale parallel k-nearest neighbor on distributed architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 494–503.

[2] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*, 2009.

[3] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[4] "ANNOY library," https://github.com/spotify/annoy, accessed: 2017-08-01.

[5] M. Iwasaki, "Proximity search in metric spaces using approximate k nearest neighbor graph," *IPSJ Trans. Database*, vol. 3, no. 1, pp. 18–28, 2010.

[6] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[7] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Soda*, vol. 93, no. 194, 1993, pp. 311–21.

[8] G. Navarro, "Searching in metric spaces by spatial approximation," *The VLDB Journal*, vol. 11, no. 1, pp. 28–46, 2002.

[9] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[10] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[11] E. Chávez and E. S. Tellez, "Navigating k-nearest neighbor graphs to solve nearest neighbor searches," in *Mexican Conference on Pattern Recognition*. Springer, 2010, pp. 270–280.

[12] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[13] D. Baranchuk, A. Babenko, and Y. Malkov, "Revisiting the inverted indices for billion-scale approximate nearest neighbors," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 202–216.

[14] M. Douze, H. Jégou, and F. Perronnin, "Polysemous codes," in *European Conference on Computer Vision*. Springer, 2016, pp. 785–801.

[15] M. Zhang and Y. He, "Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 1673–1682.

[16] W. Zhou, C. Yuan, R. Gu, and Y. Huang, "Large scale nearest neighbors search based on neighborhood graph," in *2013 International Conference on Advanced Cloud and Big Data*. IEEE, 2013, pp. 181–186.

[17] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, 1990.

[18] O. Beaumont, A.-M. Kermarrec, L. Marchal, and É. Rivière, "Voronet: A scalable object network based on voronoi tessellations," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–10.

[19] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," Cornell University, Tech. Rep., 1999.

[20] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2011, pp. 861–864.

[21] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.

[22] F. Iglesias, T. Zseby, D. Ferreira, and A. Zimek, "Mdcgen: Multidimensional dataset generator for clustering," *Journal of Classification*, vol. 36, no. 3, pp. 599–618, 2019.