A concurrent *k*-NN search algorithm for R-tree

Jagat Sesh Challa, Poonam Goyal, Nikhil S., Sundar Balasubramaniam, Navneet Goyal
Advanced Data Analytics and Parallel Technologies Laboratory
Department of Computer Science & Information Systems
Birla Institute of Technology & Science - Pilani, Pilani Campus, Pilani, India
{jagatsesh, poonam, h2014105, sundarb, goel}@pilani.bits-pilani.ac.in

ABSTRACT
k-nearest neighbor (*k*-NN) search is one of the commonly used query in database systems. It has its application in various domains like data mining, decision support systems, information retrieval, multimedia and spatial databases, etc. When *k*-NN search is performed over large data sets, spatial data indexing structures such as *R-trees* are commonly used to improve query efficiency. The *best-first* *k*-NN (BF-*k*NN) algorithm is the fastest known *k*-NN over *R-trees*. We present CBF-*k*NN, a concurrent BF-*k*NN for *R-trees,* which is the first concurrent version of *k*-NN we know of for R-trees. CBF-*k*NN uses one of the most efficient concurrent priority queues known as *mound*. CBF-*k*NN overcomes the concurrency limitations of priority queues by using a tree-parallel mode of execution. CBF-*k*NN has an estimated speedup of $O(p/k)$ for p threads. Experimental results on various real datasets show that the speedup in practice is close to this estimate.

CCS Concepts
• Information systems→Nearest-neighbor search • Computing methodologies→Shared memory algorithms.

Keywords
Data mining, *k*-nearest neighbor search, R-tree, concurrent data structures, priority queues, mounds, best first search.

1. INTRODUCTION
k-nearest neighbor search (*k*-NN) [1] is a query that retrieves *k* closest objects to a point q, from a given dataset. The distance of any of these objects from q is less than or equal to the distance of k^{th} farthest object from q. *k*-NN query is very commonly used in data mining, decision support systems, information retrieval, etc.

Algorithms for *k*-NN search have been extensively explored in literature. The approaches performing *k*-NN search can be broadly classified into two – Brute Force *k*-NN and *k*-NN using specialized data indexing structures [2]. Brute Force *k*-NN scans the entire dataset to compute nearest neighbors without using any spatial information [1, 2]. Whereas, *k*-NN approaches using specialized data indexing structures use information about spatial locality to improve query performance [2].

Compute 2015, October 29-31, 2015, Ghaziabad, India

DOI: http://dx.doi.org/10.1145/2835043.2835050

The spatial indexing structures that are commonly used are: *R-tree* [3] and its variants (*R*-tree*, *Hilbert-R-tree*, *Priority R-tree*, etc.) [4], *k-d-tree* [5], etc. These structures are used to improve the efficiency of neighborhood queries and nearest neighbor queries. *k*-NN search algorithms for *R-tree* and its variants have been extensively reported in literature [6, 7, 8]. All of these are branch-and-bound or greedy algorithms based on either depth-first approach or best-first approach. The best among these is the BF-*k*NN algorithm which is based on best-first approach [8]. The nearest neighbor search on *k-d-tree* has also been reported in literature [9, 10]. These are also branch-and-bound algorithms based on depth-first approach.

Both Brute Force *k*-NN and specialized data indexing structure based *k*-NN search have been parallelized and presented in literature. Brute Force *k*-NN search has been parallelized using GPUs [11]. Parallel *k*-NN search for *k-d-tree* has also been reported in literature using GPUs [12]. Very recently, a concurrent *k*-NN search algorithm for *k-d-trees* [13] on shared memory architectures has been reported. Parallel *k*-NN search for *R-tree* has also been reported in literature for distributed memory architectures [14, 15]. They essentially use *parallel-R-trees* [4] for indexing their data in a cluster of machines. However, no concurrent *k*-NN search algorithm for *R-tree* has been reported so far for shared memory architectures.

k-NN search is a very popular query used in domains like data mining, multimedia and spatial databases, decision support systems, information retrieval, etc. [16]. R-tree is commonly used for efficient query processing in these systems. Also, when these applications are applied to large datasets, *k-d-tree* is not suitable, because its height becomes very large and thus deteriorating the query performance. Today, with increase in number of processor cores in compute nodes, there is a need for an efficient concurrent *k*-NN search algorithm for *R-tree* that works on shared memory (multi-core) architectures using multithreading. To the best of our knowledge, there is no such variant reported.

We propose a concurrent *k*-NN search algorithm based on BF-*k*NN, using the most efficient concurrent priority queue called *mound* [17]. It is a well-known fact that concurrent priority queues do not scale beyond a few threads [18]. To address this issue, we have structured our solution such that only a few threads contend to update a priority queue. This approach results in a tree-parallel mode of execution: *one priority queue is associated with a group of* c *threads in one level of execution and* c *of those priority queues are merged in the next level*, where c is a small constant determined experimentally. This results in a scalable concurrent BF-*k*NN algorithm, namely CBF-*k*NN that has a speedup of $O(p/k)$ for p threads. We have performed experiments to measure the average query response time of CBF-*k*NN for various real datasets. The experiments show that the speedup achieved in practice is close to the theoretical estimate. We also demonstrate that the performance of CBF-*k*NN is maintained with increase in size and dimensionality of the dataset.

The rest of the paper is organized as follows. Section 2 gives literature survey of *k*-NN search and concurrent priority queues. Section 3 gives background of *R-tree*, BF-*k*NN search and *mound* data structure. Section 4 presents our algorithm (CBF-*k*NN). Section 5 presents the experimental results, followed by conclusions and future work in Section 6.

2. RELATED WORK

In this section we present a brief literature survey on *k*-NN search using specialized data indexing structures and a brief survey on concurrent priority queues.

2.1 K-NN and data indexing structures

k-NN search using specialized data indexing structures have been reported extensively in literature. The most commonly used data indexing structures are *k-d-tree* [5] and *R-tree* [3] or its variants [4]. *k*-NN search on *k-d-tree* has been well explored and presented in [9, 10]. They essentially use the branch-and-bound technique in depth-first fashion to compute nearest neighbors. *k*-NN search on *R-tree* and its variants has also been reported extensively in literature [6, 7, 8]. Of these, [6, 7] follow depth-first search. These are branch-and-bound algorithms that use pruning heuristics like *minDist*, *minmaxDist*, etc. to compute the nearest neighbors [6]. On the other hand, [8] is a greedy algorithm based on best-first search, known as BF-*k*NN. This is the most efficient among all these. It is this version that forms the basis of our concurrent version of *k*-NN. BF-*k*NN uses *minDist* as the pruning criterion. *k*-NN search over *R-tree* is more efficient for spatial databases than that over *k-d-tree*. This is mainly because, the height of *k-d-tree* becomes large when compared to *R-tree* while indexing large data sets, thus giving lesser query performance.

Parallel variants of *k*-NN search algorithm on these data indexing structures are also reported in literature. Parallel *k*-NN over *k-d-tree* has been reported in [12]. They use a data parallel model for execution on GPUs. Recently, a concurrent *k*-NN search algorithm for *k-d-tree* has been designed for shared memory architectures [13]. This algorithm uses queues to store *k-d-tree* nodes to enhance parallelism. Their limited experimental results show that the mean response time of their algorithm is better than its sequential counterpart for datasets of dimensionality up to 12.

Parallel variants of *k*-NN search for *R-tree* have also been reported in literature for distributed memory architectures [14, 15]. They essentially employ *parallel-R-tree* [4] for indexing their data in a cluster of compute nodes. These approaches are also based on the branch-and-bound technique and usually follow depth-first search. These algorithms also employ various pruning heuristics like *minDist*, *maxDist*, *minmaxDist*, *kdist*, etc [6]. These algorithms are primarily designed to answer queries efficiently in a multi-disk setup.

Although a few parallel variants of *k*-NN search algorithm for *R-tree* exist in literature, there has been no attempt reported specifically for shared memory architectures (multi-threaded environment). In our paper we present a concurrent *k*-NN algorithm for *R-tree* called CBF-*k*NN.

2.2 Concurrent Priority Queues

Priority Queue is a commonly used data structure. It has its applicability in various domains such as Operating Systems (for task scheduling), implementations of greedy algorithms, etc. Implementations of Priority Queues are based on arrays, trees, heaps, skip-lists or mounds [17, 19]. Concurrent versions of these variants have also been reported in literature. Concurrent priority queues support concurrent *insert()* and *removeMin()* operations without losing data consistency. They use synchronization mechanisms like coarse-grained locking, fine- grained locking and non-blocking (lock-free) synchronization to maintain data consistency [19]. Both *linearizable* and *quiescently consistent* versions of concurrent priority queues are reported in literature [17, 19]. One of the early implementations is the *Hunt's heap* [19] which is based on heap. It is linearizable and uses fine-grained locking. A better and faster version has been implemented using concurrent *skiplist* [19]. This is either lock-based or lock-free. It is very fast and is quiescently consistent. Linearizable version of the same has also been reported which is lock-free [19]. More recently a *mound* based concurrent priority queue has been reported which includes both lock-free and fine-grained lock based versions [17]. This is linearizable and works better than *skiplist* based priority queues in mixed workloads. We adopt the fine grained lock based version of concurrent *mound* in this paper for implementing our algorithm. Refer to section 3.3 for details.

3. BACKGROUND

In this section, we give a brief background on *R-tree* and its structure, the sequential best-first *k*-NN search algorithm for *R-tree* (BF-*k*NN) and the fine-grained concurrent *mound*.

3.1 R-tree

R-tree [3] is a commonly used multidimensional indexing structure in databases for indexing d-dimensional spatial objects. It supports efficient execution of *point, window, neighborhood*, and *nearest neighbor* queries in logarithmic time. *R-tree* consists of two kinds of nodes – internal nodes and external nodes (see Figure 1). Internal nodes store *d*-dimensional minimum bounding rectangles (*mbrs*) which further point to other internal or external nodes. An *mbr* stores the region of the bounding rectangle that contains all the regions of the nodes stored in the sub-tree rooted at it. External nodes store entries indexing d-dimensional data points. Each node (both internal and external) has a minimum of *m* entries and maximum of *M* entries stored in it (fan-out). For more details of R-tree and its algorithms, please refer to [3].

3.2 BF-*k*NN Search in R-tree

BF-*k*NN search algorithm over *R-tree* is proposed in [8]. It is a greedy algorithm based on best-first approach. It uses only one distance metric called *minDist* for its greedy choice and uses a priority queue to exercise greedy choice property. *minDist* is the minimum possible distance between a query point *q* and an object *N* (*N* is an *mbr* of a node of *R-tree*). It serves as the lower bound for the distance of any object lying in *mbr N* to the query point *q* (see Figure 2). Algorithm 1 describes the pseudo code for BF-*k*NN algorithm. For further details of BF-*k*NN please refer to [8].

3.3 Mound – Concurrent Priority Queue

Mound is one of the most efficient implementations of concurrent priority queue. Figure 3 presents the basic organization of *mound*. It is an array based implementation of a complete rooted binary tree. Every node in this tree consists of sorted lists of data points. *Mound* structure resembles that of a *heap* and all its properties apply to it. Mound is height balanced and uses randomization for insertion, giving asymptotic guarantees. Randomization also helps in improving disjoint access parallelism, which benefits concurrency of any data structure. Like any other priority queue implementations, *mound* also supports basic operations like *insert()*, *extractMin()*, *top()*, etc. Since *mound* is similar to *heap*, it supports a function called *moundify()* that is similar to *heapify()* in heap. If any of the *mound* properties gets violated during an *insert()* or *removeMin()* operation, *moundify()* is called to rectify the violation. The insertion in mound is a randomized operation

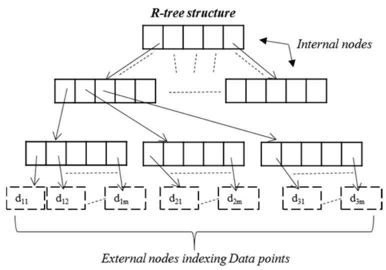

Figure 1. R-tree structure

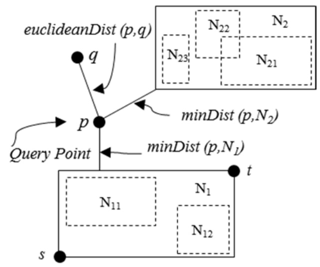

Figure 2. Illustrating *minDist* and *euclideanDist* calculation

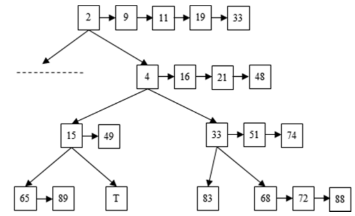

Figure 3. Mound data structure

with complexity $O(log(log\ N))$ where N is number of data points. It essentially employs binary search to find appropriate position to insert the new element. The complexity of *extractMin()* operation is $O(log(N))$. This operation has almost the same overheads as that of a normal heap.

Concurrent *mound* [17] has been implemented in two ways – 1) *fine-grained lock* based approach and 2) *lock-free* approach using atomic instructions supported by the underlying architecture. In this paper, we use the *fine-grained lock* based mound. This scales better than the lock-free one in mixed workloads (concurrent *inserts* and *removeMins*). Concurrent *mound* is linearizable and works better than the hunt's heap. For more details on the operations of concurrent mound, please refer to [17].

4. CONCURRENT *k*-NN: CBF-*k*NN

In this section we present the proposed concurrent *k*-NN algorithm (CBF-*k*NN). We first present a restricted version of our algorithm called Pre-CBF-*k*NN in subsection 4.1, followed by the complete algorithm in subsection 4.2. Complexity analysis of CBF-kNN is presented in subsection 4.3.

4.1 Pre-CBF-*k*NN Algorithm

Algorithm 2 presents the restricted version of our algorithm called Pre-CBF-*k*NN. This algorithm follows the same steps as that of sequential BF-*k*NN algorithm except that multiple threads operate on a single concurrent priority queue. A concurrent priority queue *CPQ* is used to store data points and *mbrs* of *R-tree*. *RQ* is another concurrent priority queue that is used to store the intermediate results computed by each thread before the program returns actual

ALGORITHM 1. BF-*k*NN Search in R-tree

Input: Data point q, R-tree R, k
Output: k nearest neighbors of q
Initialize Empty Priority Queue PQ;
Add root node of the R-tree R into PQ with its *minDist* as key;
int i = 1;
repeat
  element *ele* = *removeMin*(PQ);
  if (*ele.type* == *internal node*) **then**
    add all its entries to PQ with their respective *minDist* (from q) as keys;
  else if (*ele.type* == *external node*) **then**
    add all its entries to PQ with their respective *euclideanDist (from* q*)* as keys;
  else if (*ele.type* == *datapoint*) **then**
    report *ele* as i^{th} nearest neighbor;   *i*++;
    if (*i*>*k*) **then**
      break;
    end
  end
until *false*;

nearest neighbors. *RQ* is passed as an argument to be updated inside Pre-CBF-*k*NN. The root node of the *R-tree* is inserted into *CPQ* with its *minDist* from query point *q* as the key see section 3.2 for *minDist* computation). Then *t* threads are spawned. Each thread executes the following steps in a loop until *k* neighbors have been computed. Each thread performs a *removeMin()* operation on *CPQ*. If the element returned is NULL, then the thread proceeds for next iteration. If the element returned is an internal node of *R-tree*, the thread inserts all the entries of the node into *CPQ* with their respective *minDist* as keys. If the element returned is an external node of *R-tree*, the thread inserts all its entries into *CPQ* with their respective *euclideanDist* as keys. If the element returned is of type data point, the thread inserts it into *RQ*. Finally when all the threads are done, the top *k* elements in *RQ* are returned as the *k* nearest neighbors of q.

In the above algorithm, the number of threads *t* is to be set such that each thread will have at least *k* data points. We can also observe that the number of elements entering the priority queue in Pre-CBF-*k*NN can be as much as *t* times as that of BF-*k*NN. This could increase the query response time for the concurrent algorithm. To overcome this, we have used *mound* data structure

ALGORITHM 2. Preliminary Concurrent *k*-NN (Pre-CBF-*k*NN)

Input: Data point q, R-tree R, k, Number of threads t, Concurrent Priority Queue RQ
Output: k nearest neighbors of q stored in RQ
Initialize Empty Concurrent Priority Queue CPQ*; // shared by all* t *threads*
Add root of the R-tree R into CPQ along with its *minDist* as key;
for each thread t_i **|** $i = 0\ ..\ t-1$ **in parallel**
  int *i* = 1;
  repeat
    element *ele* = *removeMin*(CPQ);
    if (*ele* == **NULL**) **then**
      continue;
    end
    if (*ele.type* == *internal node*) **then**
      add all its entries to CPQ with their respective *minDist* from q as key;
    else if *(ele.type == external node)* **then**
      add all its entries to CPQ with their respective *euclideanDist* from q as key;
    else if *(ele.type == datapoint)* **then**
      add *ele* to RQ; *i*++;
      if *(i>k)* **then**
        break;
      end
    end
  until *false;*
end parallel for
Report the top k elements in RQ as k nearest neighbors;

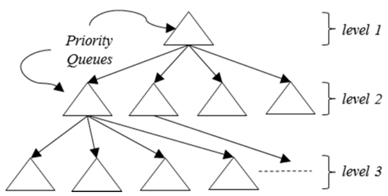

Figure 4. Tree parallel mode of execution of CBF-*k*NN

as a priority queue which stores lists of elements at its nodes, thus keeping the height of the priority queue almost the same as the height of the priority queue in sequential BF-*k*NN. Moreover, *mound* gives $O\ (log\ (log\ N))$ asymptotic complexity of insertion which is better than any other version of concurrent priority queue [17]. *Mound* is a linearizable variant of concurrent priority queue. Linearizability is essential for correctness & consistency of our algorithm.

4.2 CBF-*k*NN Algorithm

In this subsection we present the complete CBF-*k*NN algorithm. The motivation behind the design of CBF-*k*NN comes from the limitations of Pre-CBF-*k*NN. The performance analysis presented in Section 5 clearly show how Pre-CBF-*k*NN performs better than its sequential counterpart, but only up to 6 threads (See Figure 5). This confirms the claim in [18] that priority queues do not scale beyond a few threads at mixed workloads. Our design of CBF-*k*NN overcomes this problem by using tree-parallel mode of execution. CBF-*k*NN uses multiple priority queues in a tree parallel fashion wherein only a fixed number of threads *c* contend at every priority queue. Figure 4 illustrates this for *c* = 4 and three levels. Every priority queue at the bottom most level (3rd level in this case) is associated with *c* threads and the results of each of these are merged at a priority queue at its parent level in parallel. In this way the results are merged hierarchically wherein, exactly c threads contend at every priority queue. This flow can be extended to any number of levels depending on the data size and availability of processor cores.

Algorithm 3 presents the pseudo code of CBF-*k*NN. Let the number of threads be *p*. Let, the number of threads contending at every priority queue be *c*. Initially the data is partitioned into *p/c* partitions $\{PART_i \mid i = 0 .. (p/c - 1)\}$. For every partition $PART_i$ in parallel, an empty concurrent priority queue $\{RQ_i \mid i = 0 .. (p/c - 1)\}$ and an empty *R-tree* $\{RT_i \mid i = 0 .. (p/c - 1)\}$ are initialized. The data points that belong to each partition are then inserted into their respective *R-trees*. Now for each partition $PART_i$, c threads are assigned and Pre-CBF-kNN (Algorithm 2) is executed. *k*-nearest neighbors for each partition are computed by this and the results are stored in their respective result priority queues RQ_i. Then all the result priority queues are merged by parallel reduction with *p/c* threads. The top *k* elements in the topmost priority queue are returned as *k* nearest neighbors.

ALGORITHM 3. Concurrent *k*-NN (CBF-*k*NN)

Input: Data point q, k, constant c, no of threads p
Output: k nearest neighbors of q
Divide the data into p/c partitions: $\{PART_i \mid i = 0 .. (p/c - 1)\}$;
// Now initializing empty concurrent priority queues and constructing R-trees for every partition…
for each thread $P_i \mid i = 0 .. (p/c - 1)$ **in parallel** for p/c data partitions
  Initialize an empty concurrent priority queue RQ_i for this data partition $\{RQ_i \mid i = 0 .. (p/c - 1)\}$; // *(global)*
  Initialize an empty *R-tree* RT_i for this data partition $\{RT_i \mid i = 0 .. (p/c - 1)\}$; // (global)
  Insert all the data points lying in this partition into RT_i;
end parallel for

for each data partition $PART_i \mid i = 0 .. (p/c - 1)\}$ **in parallel**
  // assign c threads to each partition and execute Pre-CBF-kNN
  Pre-CBF-KNN(q, RT_i, k, c, RQ_i);
end parallel

merge all RQ_i for $i = 0 .. p/c - 1$ by **parallel reduction** with p/c threads;
Report the top k elements in the topmost priority queue PQ_0 as k nearest neighbors;

4.3 Complexity Analysis

The time complexity of sequential BF-*k*NN [8] is:-

$$T\_seq = N * log\,N$$

where N is the data size.

In Algorithm 2, time taken by each thread is dominated by operations on the two priority queues. Each thread inserts/deletes *N/p* elements into *CPQ* (of total size *N*) and *k* elements into *RQ* (of total size *p*k*). Thus each thread takes *(N/p) * log N* steps for constructing *CPQ* and *k * log (p*k)* for constructing *RQ* when *p* threads are running. Here it is assumed that concurrent access does not slow down insertion/deletion by more than a constant factor. Our assumption is justified in the context of Algorithm 3 which controls concurrent access by limiting the number of threads to *c* (when calling Algorithm 2) to ensure that priority queue operations are scalable. Thus, the time complexity of Algorithm 2 is:-

$$T_2 = (N/p) * log\,N + k * log\,(p * k)$$

The time taken by Algorithm 3 is the sum of time taken for constructing an *R-tree* with *N/p* elements, the time taken for running Algorithm 2, and the parallel reduction with *p/c* threads. Each step in the reduction inserts *k* elements into a priority queue of total size *k*c*. So, the time taken by Algorithm 3 is:-

$$T_{par} = (N/p) * log(N/p) + T2 + log(p/c) * k * log(k * c)$$

Therefore the speedup achieved is (T_{seq}/T_{par}) which when simplified gives:

$$T_{seq}/\,T_{par} = O(p/k)$$

5. RESULTS & ANALYSIS

In this section, we report our experimental results. All experiments were conducted over HP Proliant DL 580 gen8 server that has four Intel Xeon 12 core 2.29 GHz Hyper Threaded Processors, 192 GB RAM and 600 GB HDD. All programs have been implemented in C with Posix threads for multithreading. The performance has been measured using Vampir Trace profiler [20]. All experiments are conducted on real datasets of varying size and dimensions. Their details are provided in Table 1. 3DSRN data set is taken from UCI repository and contains geographical information (latitude, longitude and altitude) of road networks in Denmark [21]. MPAGD5M, MPAGD18M and SFONT1M datasets are taken from Millennium data repository that contains

Table 1. Details of Datasets used for Experimentation

Dataset	Size	Dimensions	Reference
3DSRN	0.34 M	3	[31]
MPAGD5M	5 M	3	[32]
MPAGD18M	18 M	3	[32]
SFONT1M	1 M	11	[32]
SBUS6M	6 M	2	[33]

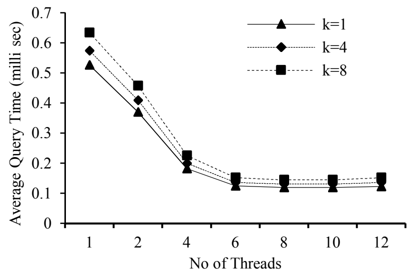

Figure 5. Average Query Execution time of Pre-CBF-KNN vs No of threads for 3DSRN dataset

astronomical data of galaxies in the sky [22]. SBUS6M dataset contains a sample of GPS traces of buses in Shanghai [23]. In all experiments, the average query response time has been measured by taking average of query response time of 10% data sample on each data set.

Figure 5 presents the performance results of Pre-CBF-*k*NN algorithm when run on 3DSRN dataset with variation in number of threads from 1 to 12 for k = 1, 4, and 8. The figure clearly indicate that the average query response time improves with increase in number of threads up to 6 for all values of k. The performance does not improve beyond 6 threads because of memory contention at the priority queue in mixed workloads (insertion and deletion). Similar behavior was observed in other data sets as well. This conforms to the scalability limitations of concurrent priority queues reported in [18]. As explained in Section 4, we have addressed this issue by designing the CBF-*k*NN algorithm which executes in a tree parallel mode. The results of this version are presented in Figure 6. Figures 6a, 6b, and 6c show average query response time of CBF-*k*NN with tree parallel configurations up to 2 levels for 3DSRN dataset with values of c = 2, 4, and 6 respectively. Figures 6d, 6e, and 6f show the same for SBUS6M. Figures 6g and 6h shows the same with tree parallel configurations up to 3 levels for MPAGD18M dataset with c = 2 and c = 4 respectively and Figure 6i shows the same up to 2 levels with c = 6. The results clearly indicate that the improvement in the average query response time is much better than that of Pre-CBF-*k*NN. Also CBF-*k*NN scales for larger number of threads. Figures 6g and 6h also indicate that the query response time improves with increase in number of levels in the tree parallel execution.

Figure 7 presents the results of an experiment conducted to measure the query performance with variation in k. The experiments are conducted for 3DSRN dataset with 1 thread, 4 threads, and 4*4 threads in a 2-level tree parallel configuration. The results clearly indicate that the improvement in query performance is maintained with increase in value of k, although the rate of improvement reduces for higher values of k. This conforms to our theoretical estimate of $O(p/k)$ speedup.

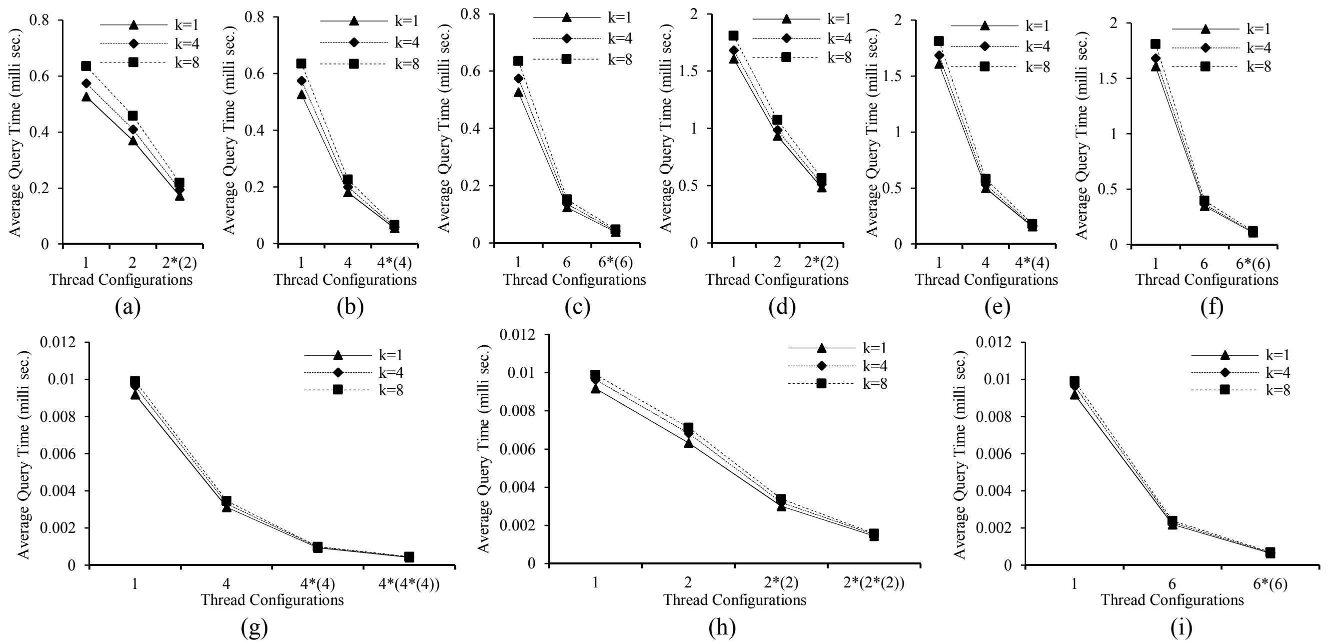

Figure 6. Avg. Query Exec. time of CBF-KNN for: (a), (b)&(c) 3DSRNwith c=2, 4 & 6 respectively; (d), (e)&(f) SBU6Mwith c=2, 4 & 6 respectively; (g),(h)&(i) MPAGD18M with c=2, 4 & 6 respectively.

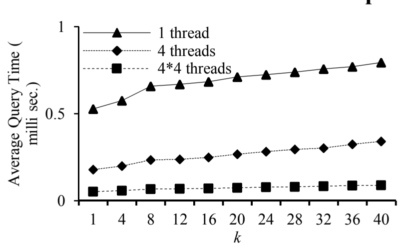

Figure 7. Avg. Query Exec. Time of CBF-KNN for 3DSRN with varying k

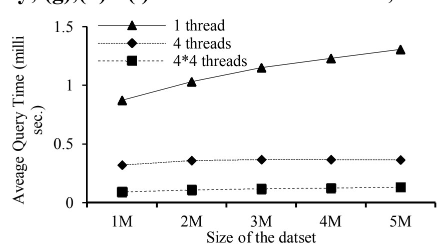

Figure 8. Avg. Query Exec. Time of CBF-KNN with varying size of samples for MPAGD5M for *k*=4

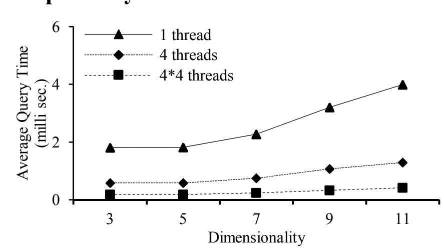

Figure 9. Avg. Query Exec. Time of CBF-KNN with varying dimensions for SFONT1M samples for *k*=4

Table 2. Speed up of CBF-*k*NN

Dataset	p	k	Estimated Speedup - p/k	Measured Speed up
MPAGD18M	64	4	16	20
MPAGD18M	36	4	9	15

Figure 8 presents results of an experiment conducted to measure the query response time with variation in size of the dataset. For this experiment, MPAGD5M data set has been sampled for 1M, 2M, 3M, 4M and 5M data points. CBF-*k*NN has been executed with 1 thread, 4 threads and 4*4 threads in a 2-level tree parallel configuration on all these samples. The results indicate that the performance improvement in average query response time is maintained with increase in data size.

Figure 9 presents the results of an experiment conducted to measure the query response time with variation in dimensionality of the dataset. For this experiment, SFONT1M data set that has originally 11 dimensions has been sampled for 3, 5, 7, and 9 dimensions randomly. CBF-*k*NN has been executed with one thread, 4 threads and 4*4 in a 2-level tree parallel configurations on all these samples. The results indicate that the performance improvement in average query response time is maintained with varying dimensionality. The rate of increase in average query time decreases with increase in number of threads.

Table 2 presents the speed up of CBF-*k*NN algorithm when run on MPAGD18M data set when run on various configurations. It can be observed that the measured speed up is comparable to the asymptotic estimate of $O(p/k)$. The difference observed is due to the constant factor.

6. CONCLUSIONS & FUTURE WORK

We have presented a novel concurrent algorithm (CBF-*k*NN) for *k*-nearest neighbor search on *R-trees* that uses multiple concurrent priority queues and executes in a tree parallel mode. To the best of our knowledge this is the first work reported of its kind. CBF-*k*NN gives a speed up of $O(p\ /\ k)$, where p is the number of threads. The results presented in Section 5 clearly show that CBF-*k*NN scales well with increase in processor cores and gives speed up close to the estimate. The results also indicate that the speedup is maintained with increase in size and dimensionality of the data set.

The CBF-*k*NN algorithm can be further optimized by including some other pruning heuristics on the elements entering the priority queue. A hybrid algorithm can be designed for *k*-NN to run on hybrid of distributed and shared memory architectures.

7. ACKNOWLEDGEMENT

This work has been partially supported by a research grant from Department of Electronics and Information Technology, Ministry of Communications & IT, Government of India.

8. REFERENCES

[1] T. Cover and P. Hart. 1967. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theo.* 13,1(Sep 1967), 21-27.

[2] N. Bhatia and Vandana. 2010. Survey of Nearest Neighbor Techniques. *International Journal of Computer Science & Information Security* (IJCSIS'10) 8, 2 (2010), 302-305.

[3] A. Guttman. 1984. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*14, 2 (June 1984), 47-57.

[4] Y. Manolopoulos, et al. 2005. R-Trees: Theory and Applications (Advanced Information and Knowledge Processing). Springer-Verlag New York, Inc., NJ, USA.

[5] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (Sep 1975), 509-517.

[6] N. Roussopoulos, S. Kelley, and F. Vincent. 1995. Nearest neighbor queries. *SIGMOD Rec.* 24, 2 (May 1995), 71-79.

[7] K. L. Cheung and A. W. Fu. 1998. Enhanced nearest neighbour search on the R-tree. *SIGMOD Rec.* 27, 3 (Sep 1998), 16-21.

[8] G. R. Hjaltason and H. Samet. 1999. Distance browsing in spatial databases. *ACM Trans. Database Syst.* 24, 2 (June 1999), 265-318

[9] J. H. Friedman, J. L. Bentley, and R. A. Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Soft.* 3, 3 (Sep 1977), 209-226.

[10] R. F. Sproull. 1991. Refinements to Nearest-Neighbor Search in *k*-Dimensional Trees. *Algorithmica* 6, (1991), 579-589.

[11] N. Sismanis, N. Pitsianis, and X. Sun. 2012. Parallel search of k-nearest neighbors with synchronous operations. In *Proceedings of 2012 IEEE Conference on High Performance Extreme Computing* (HPEC), IEEE Computer Society, Washington D.C., USA, 1–6.

[12] F. Gieseke, et al. 2014. Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs. In *Proc. of 31st International Conference on Machine Learning*, Beijing, China, 2014, 1-9.

[13] T. Hering. 2013. Parallel Execution of kNN-Queries on in-memory K-D Trees. In *Proc. of 15th GI Symposium on Business, Technology & Web* (BTW'13), Magdeburg, Germany, 257-266.

[14] A. N. Papadopoulos and Y. Manolopoulos. 1998. Similarity query processing using disk arrays. In *Proc. of the 1998 ACM SIGMOD international conference on Management of data* (SIGMOD '98), ACM, New York, NY, USA, 225-236.

[15] Y. Gao, et al. 2006. Efficient Parallel Processing for K-Nearest-Neighbor Search in Spatial Databases. *Lect. Notes in Comp. Science* 3984 (2006),39-48.

[16] C. Bohm and F. Krebs. 2002. High Performance Data Mining Using the Nearest Neighbor Join. In *Proc. of IEEE International Conf. on Data Mining* (ICDM), Japan, 43-50.

[17] Y. Liu and M. Spear. 2012. Mounds: Array-Based Concurrent Priority Queues. In *Proc. of 41st International Conference on Parallel Processing* (ICPP '12). IEEE Computer Society, Washington, DC, USA, 1-10.

[18] D. Alistarh, et al. 2015. The SprayList: a scalable relaxed priority queue. In *Proc. of 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP 2015). ACM, NY, 11-20.

[19] M. Herlihy and N. Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] VampirTrace Library, http://www.tudresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace

[21] V. Springel, et al. 2005. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature* 435, 7042, 629–636.

[22] SUVnet-Trace data, http://wirelesslab.sjtu.edu.cn.

[23] M. Kaul, B. Yang, and C. S. Jensen. 2013. Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems. In *Proc. of 14th International Conference on Mobile Data Management* (IEEE MDM'13), Milan, Italy, 137-14.