Best-First Based Parallel Nearest Neighbor Queries

Yunjun Gao, Gencai Chen, Ling Chen, Chun Chen

College of Computer Science, Zhejiang University
Hangzhou, 310027, P. R. China
{gaoyj, chengc, lingchen, chenc}@cs.zju.edu.cn

Abstract

Given a query point *q*, a nearest neighbor (NN) query retrieves the closest data point with the minimum distance to *q* in space (e.g., "*find the nearest hotel to the airport*"). It is one of the most important operations in spatial databases and spatio-temporal databases. However, most of existing methods for NN search only aim at a single disk to find the NN of *q*, which incur significant query cost (involving CPU time and I/O overhead) and huge number of accessed nodes with the increasing volume of data points. Motivated by these problems, in this paper, we present the first Best-First based Parallel NN (BFPNN) algorithm and Full BFPNN (FBFPNN) algorithm for effective processing of NN retrieval by means of parallelism (i.e., fetching more nodes or data points from multiple disks simultaneously) in multi-disk setting. Furthermore, extensive experiments verify that the proposed algorithms are correct, and also outperform the existing ones (e.g., FPSS and CRSS algorithms) under most cases in terms of effectiveness and scalability, by using various real and synthetic datasets. The goal is to reduce the query cost and alleviate I/O overhead, so as to facilitate the execution of the NN retrieval.

1. Introduction

Increasing applications involve intensive both data and computation, and require the large storage and manipulation of numerous traditional and non-traditional datasets (e.g., images, moving objects, etc.). These applications mainly include GIS, CAD/CAM, OLAP, and so forth. They impose various requirements with respect to the information and the operations that need to be supported. Thereinto, nearest neighbor (NN) search is one of the most important operations from the database perspective. Recently, it has attracted considerable attention in the database community. This paper just focuses on this theme.

Given a query point *q* and a dataset *s*, a NN query retrieves the closest data point *p* with smaller distance (in this paper we use Euclidean distance) to *q* than the one from any other data point contained in *s* to *q* in space. Formally, NN (*q*) = $\{p \in s \mid \neg\exists p' \in s'$ such that *dist* (*p'*, *q*) < *dist* (*p*, *q*)$\}$, where $s' = \{s - p\}$ and *dist* is a distance metric. For instance, in GIS, a tourist wants to find the nearest hotel to the airport when he/she attains a new city for the first time.

In the past decade, there have been attracted great interests and advancements in the area of the NN retrieval. Most methods of the NN queries have been presented by the database researchers. In particular, we categorize them into following four types: (i) *conventional* (or *stationary*) NN queries [1, 2, 3, 4], (ii) *continuous* NN (CNN) queries [5, 6, 7, 8, 9, 10], (iii) *aggregated* NN (ANN) queries [11, 12], and (iv) *reverse* NN (RNN) queries [13, 14, 15, 16, 17, 18], according to whether the query point and data points (dataset) are fixed or moving.

Most of the previous approaches for NN search can efficiently perform in the sequential process environment that only uses a single disk. In particular, various NN queries based on Best-First (BF) algorithm are the optimal ones with respect to query cost (including CPU time and I/O overhead) and the number of accessed nodes. However, with the emergence of several applications involving huge amounts of data (e.g., GIS, CAD/CAM, OLAP, etc.) that do not fit in one disk, those NN queries mainly aiming at a single-disk setting are poor for dealing with NN search in a multi-disk setting. Specifically, the query cost (time) of the NN search increases, such that it does not satisfy the requirement of the users. At the same

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005

time, the I/O cost is also large, which is easy to produce the bottleneck of I/O accesses, especially for NN queries in spatio-temporal databases. To address these problems, in this paper, we present the first Best-First based Parallel NN (BFPNN) query algorithm and Full BFPNN (FBFPNN) algorithm for effective processing of the NN retrieval by means of parallelism (i.e., fetching more nodes or data points from multiple disks simultaneously) in multi-disk setting. Furthermore, considerable experiments verify that the proposed algorithms are correct and outperform the existing ones (e.g., FPSS and CRSS algorithms [19]) under most cases in terms of effectiveness and scalability, by using various real and synthetic datasets. The goal is to improve the performance of the NN queries processing through parallel operation in multi-disk setting, so as to reduce the query cost and alleviate the bottleneck of the I/O accesses (mainly involving the number of accessed nodes).

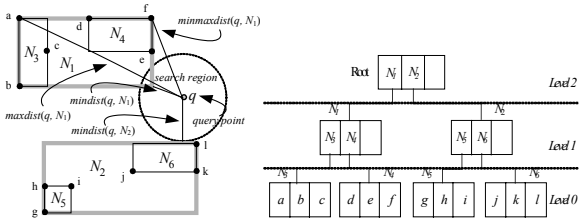
The rest of the paper is organized as follows. Section 2 surveys related work on NN search involving BF algorithm, parallel R-tree and existing parallel algorithms for NN search (i.e., FPSS and CRSS algorithms). Some definitions and problem characteristics are studied in Section 3. Section 4 presents two parallel algorithms (i.e., BFPNN and FBFPNN) for single NN search. Considerable experimental evaluations of the proposed algorithms are discussed in Section 5. Finally, Section 6 concludes the paper with directions for future work.

2. Related work

In the sequel, we assume that the dataset is indexed by a parallel R-tree due to the effectiveness of this structure in the case of parallel processing with one processor and several disks attached to it. Section 2.1 briefly surveys the R-tree and the best-first (BF) algorithm for NN search. An overview of the parallel R-tree is presented in Section 2.2. Section 2.3 describes the previous studies on parallel algorithms for NN queries.

2.1 BF algorithm for NN search using R-trees

Among various spatial indexing structures in the literature, the R-tree [20] and its variants (e.g., the R^+-tree [21], the R*-tree [22], etc.) are the most widely accepted and used ones. They can be thought of as extensions of B-trees (e.g., B^+-tree) in multi-dimensional space. Figure 2.1 (a) shows a set of points $\{a, b, \ldots, l\}$ indexed by an R-tree (Figure 2.1 (b)) assuming a capacity of three entries per node (i.e., the fanout of each node is three). In this example, according to the spatial proximity, 12 points are clustered into 4 leaf nodes $\{N_3, N_4, N_5, N_6\}$ which are then recursively grouped into nodes N_1, N_2 that become the entries of a single root node. Each node of the tree corresponds to one disk page. Intermediate nodes (e.g., N_3, N_4) contain entries of the form (R, *child_ptr*), where R is the *Minimum Bounding Rectangle* (MBR) that encloses all the MBRs of its descendants and *child_ptr* is the pointer to the page where the specific child node is stored. Leaf entries (e.g., a, b, c) store the coordinates of data points and (optionally) pointers to the corresponding data records.



(a) Points and node extents (b) The R-tree

Figure 2.1: Example of a NN query and an R-tree

Given a query point q and a d-dimensional dataset s, a NN query retrieves the point $p \in s$ that is closest to q. In particular, The NN algorithms on R-trees utilize three bounds to prune the search space, i.e., (i) *mindist* (q, N), (ii) *maxdist* (q, N) and (iii) *minmaxdist* (q, N), where N denote any node in R-trees. Specifically, *mindist* (q, N) corresponds to the minimum possible distance between q and any point in (the subtree of) node N (e.g., *mindist* (q, N_1) in Figure 2.1 (a)). Similarly, *maxdist* (q, N) specifies the maximum possible distance among all distances from any point in the subtree of node N to q (e.g., *maxdist* (q, N_1)). And *minmaxdist* (q, N) gives an upper bound of the distance between q and its closest point in N (e.g., *minmaxdist* (q, N_1)). In particular, notice that the derivation of *minmaxdist* (q, N) is based on the fact that each edge of the MBR of N contains at least one data point [2]. Hence, it also equals the smallest of the maximum distances from all edges of N to q. As shown in Figure 2.1 (a).

Existing approaches of NN search are based on either Depth-First (DF) algorithm [2, 3] or Best-First (BF) algorithm [1, 4]. However, here only overviews the BF algorithm for NN retrieval due to the proposed algorithms in this paper based on it.

The BF algorithm maintains a priority query (e.g., heap) Q containing the entries visited so far, sorted in ascending order of their *mindist*(s). BF starts from the root and inserts all its entries into Q together with their *mindist*(s). In Figure 2.1, for instance, BF starts by inserting the root entries into $Q = \{(N_1, \textit{mindist}\ (q, N_1)), (N_2, \textit{mindist}\ (q, N_2))\}$. Then, at each step, it visits the node in Q with the smallest *mindist*. Continuing the above example, the algorithm retrieves the content of N_1 and inserts its entries in Q, after which $Q = \{N_2, N_4, N_3\}$. Similarly, the next two nodes accessed are N_2 and N_6 (inserted in Q after visiting N_2) in turn, and the Q changes $Q = \{N_6, N_4, N_5, N_3\}$ and $Q = Q = \{l, N_4, k, j, N_5, N_3\}$ respectively. Here, l is discovered as the first NN of q. At this time, BF terminates with l as the final result, since the next entry (N_4) in Q is farther (from q) than l. In addition, BF uses *minmaxdist* and *maxdist* to reduce the number of node access during the NN search processing.

Furthermore, it is "incremental" (i.e., reporting NNs of q in ascending order of their distances to q).

2.2 The parallel R-tree

To maximize the parallelism for large queries and engage as few disks as possible for small queries, Kamel and Faloutsos [23] proposed "Multiplexed R-tree", which distributes the nodes of a traditional R-tree with cross-disk pointers, based on a simple hardware architecture consisting of one processor with several disks attached to it. Figure 2.2, for example, shows one possible multiplexed R-tree corresponding to the R-tree of Figure 2.1 (b) in Section 2.1.

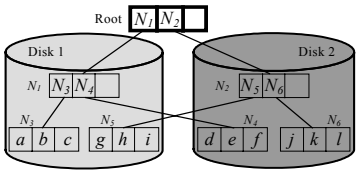


Figure 2.2: Example of multiplexed R-tree

The multiplexed R-tree operates exactly like a single-disk R-tree. But the only difference is that its nodes are carefully distributed over multiple disks. Specifically, the root node (representing as thick line in Figure 2.2) is kept in main memory, while other nodes (e.g., N_3, ..., N_6) are distributed over the two disks (i.e., *disk* 1 and *disk* 2). Furthermore, for multiplexed R-tree, each pointer contains a *disk_id* (denoting an identifier of one disk), in addition to the *page_id* (specifying a label of one page) of the traditional R-tree. In addition, [23] also presented and evaluated several heuristics, including *Round Robin* (RR), *Minimum Area* (MA), *Minimum Intersection* (MI) and *Proximity Index* (PI), to assign nodes to disks within the multiplexed R-tree framework. In this paper, however, we only use RR heuristic to distribute the node entries among multiple disks, in order to compare the efficiency of the proposed algorithms (including the BFPNN and the FBFPNN algorithms) against the existing parallel algorithms (i.e., CRSS) for NN queries.

2.3 Existing parallel algorithms for NN search

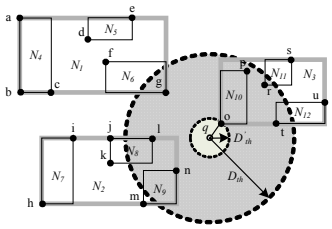
Currently, although most of existing methods of NN queries can be efficiently suitable for a single-disk setting, they have poor performance in the multi-disk environment. Therefore, effectively parallel NN search algorithms are necessary for improving the efficiency of the NN retrieval in the multiple disks setting. However, they do not attract sufficient attention in the past decade. To the best of our knowledge, the only parallel NN algorithms in the literature are the ones proposed in [19], which are based on various heuristics and leave some rooms for improvement.

Papadopoulos and Manolopoulos [19] studied similarity queries (retrieving objects that are similar to a given query vector or query point, where similarity is defined by means of a distance metric) on a *Redundant Array of Inexpensive Disks* (RAID) level 0 system. In particular, they examined four algorithmic techniques, namely, (i) *Branch and Bound Similarity Search* (BBSS), (ii) *Full Parallel Similarity Search* (FPSS), (iii) *Candidate Reduction Similarity Search* (CRSS), and (iv) *Weak OPTimal Similarity Search* (WOPTSS). Specifically, The BBSS is based on a previous *branch-and-bound* algorithm (e.g., DF algorithm [2]). The FPSS is depended on a greedy philosophy. It supposes that all the residual entries (after pruning per step) would contribute to the final answer and have to be retrieved. The CRSS tries to exploit parallelism to a sufficient degree and avoid fetching unnecessary data (i.e., not acting on the final result) by using a threshold distance D_{th} and the *Candidate Reduction Criteria* (CRC for short). The WOPTSS is a non-existing and hypothetical optimal algorithm. It is used to compare aforementioned algorithms involving BBSS, FPSS, and CRSS. In addition, from the practical perspective, [19] also reported that the CRSS outperforms any other proposed algorithms (i.e., the BBSS and the FPSS) through considerable experiments with real and synthetic datasets. In particular, it is observed that the CRSS consistently presents the best performance in terms of speed-up, scale-up and query response time in the multi-disk environment. Therefore, here only introduces the CRSS algorithm together with an illustrative example.

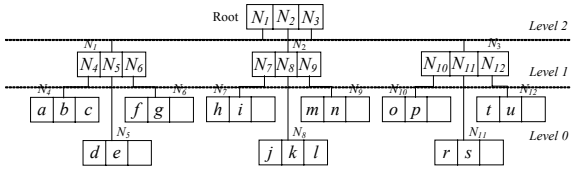
The CRSS uses a threshold distance D_{th} and CRC to prune the needless nodes. Specifically, given a query point q, D_{th}, and a MBR N, the CRC includes: (i) N is rejected when $D_{th} < mindist$ (q, N) holds, (ii) N is activated if $D_{th} \geq minmaxdist$ (q, N) satisfies, and (iii) N is saved for possible future reference when both $D_{th} \geq mindist$ (q, N) and $D_{th} < minmaxdist$ (q, N) meet. Clearly, the CRSS need some auxiliary data structures in order for it to effective work. In particular, it mainly utilizes three auxiliary structures: (i) *activation structure*, which stores the pointers to the nodes that are going to be fetched in the next step; (ii) *fetch structure*, which holds the newly fetched nodes in order to deal with them further; and (iii) *candidate structure*, which maintains the candidate nodes that have neither been accessed nor been rejected yet. Figure 2.3 (a), for instance, shows a set of points {a, b, ..., u} indexed by an R-tree (illustrated in Figure 2.3 (b)) assuming a capacity of three entries per node. Furthermore, nodes are numbered from N_1 to N_{12}. Here, given a query point q, let us trace the execution of the CRSS for a simple query requiring the NN of q (i.e., k = 1). The process of the algorithm is described as follows:

The CRSS algorithm starts from the root where the MBRs N_1, N_2 and N_3 reside (see Figure 2.3). According to the first threshold distance D_{th} and the CRC, N_2 and N_3 are activated immediately, and the pointers to N_2 and N_3 are stored in the *activation structure*; whereas N_1 is considered as a possible candidate node and pushed into the *Candidate Stack* (CS for short). Note that the candidates are inserted into the CS in decreasing order

with respect to the *mindist*(s) from q (i.e., *mindist* (q, N_3)). This case is depicted in Figure 2.4 (a), where the shaded boxes indicate *guard entries* that can be used to separate two different candidate nodes, and the letters (involving *"I"*, *"S"*, *"R"* and *"C"*) specify *inspected nodes*, *selected nodes*, *rejected nodes* and *saved candidate nodes*, respectively. After updating the CS, N_2 and N_3 can be ready to fetch from the disks that they reside. It must be noted that the requirements of fetching operation can be performed in parallel way assuming that these nodes (i.e., N_2 and N_3) store in different disks.



(a) Points and node extents



(b) The R-tree

Figure 2.3: Example of a CRSS algorithm and an R-tree

Similarly, in the next stage, nodes N_7 through N_{12} are inspected. As a result, N_8, N_9 and N_{10} are activated immediately; N_{11} and N_{12} are regarded as two possible candidate nodes; while N_7 is rejected. Here, the status of the CS in this situation is shown in Figure 2.4 (b).

The following step includes the access of the nodes N_8, N_9 and N_{10}. This is the first time that real data objects contribute to the formulation of the upper bound to the k-th (here k = 1) NN during the running of the CRSS. So, the best one out of seven objects (i.e., o) are selected and the threshold distance D_{th} is updated accordingly (i.e., D'_{th} insteads of D_{th}). Then, the first candidate nodes consisting of N_{11} and N_{12} are popped from the CS and investigated. As a consequence, they can be safely rejected by means of comparing *mindist* (q, N_{11}) and *mindist* (q, N_{12}) against the D'_{th}. The current case is illustrated in Figure 2.4 (c). Here still contains N_1 in the CS. Therefore, N_1 is popped from the stack and can be also safely discarded due to D'_{th} < *mindist* (q, N_1). Now, the CRSS has been terminated, the NN of q has been determined (i.e., o) and D_{th} = *mindist* (q, k) as well.

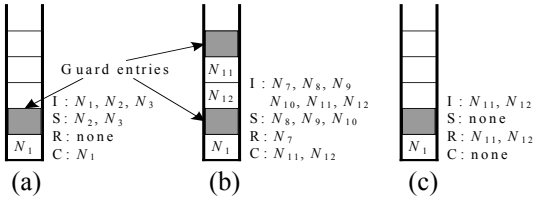


Figure 2.4: Illustration of the first three steps of CRSS

3. Definitions and problem characteristics

The objective of a NN query is to retrieve the closest data point of a given query point q. For example, the NN of q in the aforementioned Figure 2.1 is l, which has the smallest distance to q among all the distances of nodes (contained in the Figure 2.1) from q. In this section, we present some definitions and study several problem characteristics in order to derive our proposed algorithms (including the BFPNN and the FBFPNN) discussed detailedly in the Section 4 of the paper. First of all, Table 3.1 describes various symbols used in the following definitions for facilitating presentations.

Symbol	Description
q	A given query point
s	A dataset
D_n	A Euclidean distance in n-dimensional space
p	The nearest neighbor of q
N	A set of nodes (involving leaf nodes and intermediate nodes)
N_p	A node containing p, and $N_p \in N$
$N_{Dmindist}$	A node with the minimum *mindist* to q in N
$N_{Dminmaxdist}$	A node with the smallest *minmaxdist* to q in N
$N_{Dmaxdist}$	A node with the minimal *maxdist* to q in N

Table 3.1: Symbols and descriptions used in definitions

Definition 1 (distance of the NN from q): The distance $D_n(q, p)$ between q and p is defined as follows: $D_n(q, p) = \{D_n \square\square \exists D_n(q, p') < D_n(q, p)\}$, where $p' \in \{s - p\}$ and $p \in s$.

Definition 2 (consistency): Let $p \in N_p$ ($N_p \in N$) indicates the fact that the point p is exactly contained in the node N_p, then the following inequality holds: $D_n(q, N_p) \leq D_n(q, p)$.

Definition 3 (minimum *mindist*): Let *Dmindist* be the minimum *mindist* among all *mindist*(s) between q and node(s) in N. Then, the *Dmindist* can be formulated as follows: *Dmindist* = {*mindist* $\square\square \exists$ *mindist* (q, N') < *mindist* (q, $N_{Dmindist}$)}, where $N' \in \{N - N_{Dmindist}\}$ and $N_{Dmindist} \in N$.

Definition 4 (minimum *minmaxdist*): Let *Dminmaxdist* be the smallest *minmaxdist* within all *minmaxdist*(s) from q to node(s) in N. Then, the formal definition of the *Dminmaxdist* is: *Dminmaxdist* = {*minmaxdist* $\square\square \exists$ *minmaxdist* (q, N') < *minmaxdist* (q, $N_{Dminmaxdist}$)}, in which $N' \in \{N - N_{Dminmaxdist}\}$ and $N_{Dminmaxdist} \in N$.

Definition 5 (minimum *maxdist*): Let *Dmaxdist* be the minimal *maxdist* among all *maxdist*(s) to q in N. Then, the formalization of the *Dmaxdist* is described as follows: *Dmaxdist* = {*maxdist* $\square\square \exists$ *maxdist* (q, N') < *maxdist* (q, $N_{Dmaxdist}$)}, where $N' \in \{N - N_{Dmaxdist}\}$ and $N_{Dmaxdist} \in N$.

Figure 3.1 shows an example of above four distance metrics, that is, (i) distance of the NN from q (i.e., $D_n(q, g)$, as g is the NN of q), (ii) *Dmindist* (equals *mindist* (q, N_{10}), which is the smallest *mindist* among all *mindist*(s)

between a given query point q and a set of leaf nodes N (involving nodes N_4 through N_{12}), similarly, (iii) *Dminmaxdist* (equals *minmaxdist* (q, N_6)), and (iv) *Dmaxdist* (equals *maxdist* (q, N_6)). On the other hand, carefully considering these definitions, we can derive and prove the following lemmas as well:

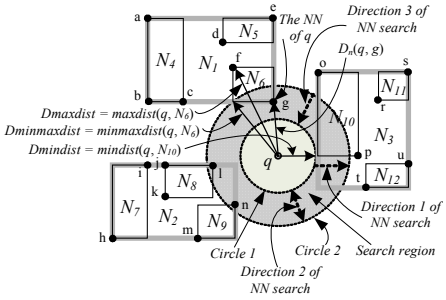


Figure 3.1: Example of four distance metrics

Lemma 1: According to the definitions of three distance metrics (i.e., D_n (q, p), *Dmindist* and *Dminmaxdist*), the following inequality holds: *Dmindist* ≤ D_n (q, p) ≤ *Dminmaxdist*.

Proof. (Sketch) In the first place, we prove the left side of the inequality in Lemma 1, i.e., *Dmindist* ≤ D_n (q, p). According to the Definition 2, the distance from a given query point q to its nearest neighbor p is not smaller than the distance between q and the node N_p containing p. So, D_n (q, N_p) ≤ D_n (q, p) holds. Similary, by Definition 3, *Dmindist* is the smallest *mindist* between q and a set of nodes N and $N_p \in N$. Hence, *Dmindist* ≤ D_n (q, N_p) holds. Associating above two inequalities, i.e., D_n (q, N_p) ≤ D_n (q, p) and *Dmindist* ≤ D_n (q, N_p), the inequality *Dmindist* ≤ D_n (q, p) is accurate.

As for the proof of the right side of the Lemma 1 (i.e., D_n (q, p) ≤ *Dminmaxdist*), we assume, to the contrary, that the inequality D_n (q, p) □ *Dminmaxdist* holds. Clearly, according to the definition of *minmaxdist*, which offers an upper bound on the actual distance of object (point) to q, there exists another data point o whose distance from q meets following inequality: *minmaxdist* ≥ D_n (q, o). At the same time, *Dminmaxdist* ≥ D_n (q, o) holds as well because the *Dminmaxdist* is the minimum *minmaxdist* among all nodes enclosed in N to q by the Definition 4. Hence, combining two inequalities, i.e., D_n (q, p) □ *Dminmaxdist* and *Dminmaxdist* ≥ D_n (q, o), we can get the inequality: D_n (q, p) □ D_n (q, o). However, because the data point p is the NN of q, we can also get another inequality: D_n (q, p) □ D_n (q, o) in terms of the Definition 1, which contradicts the derived inequality: D_n (q, p) □ D_n (q, o). Therefore, we can conclude that the inequality, i.e., D_n (q, p) ≤ *Dminmaxdist*, holds.

Thus, combining two parts of the above proof, we can determine the Lemma 1 is accurate. ■

Lemma 2: According to the definitions of three distance metrics (i.e., D_n (q, p), *Dmindist* and *Dmaxdist*), the following inequality satisfies: *Dmindist* ≤ D_n (q, p) < *Dmaxdist*.

Proof. (Omitted) We omit the description of this proof of the Lemma 2 due to the proof method is the same as that of the Lemma 1. ■

As pointed out in Figure 3.1. Specifically, we can get four distance metrics, that is, the distance of the NN q (equals D_n (q, g)), *Dmindist* (equals *mindist* (q, N_{10})), *Dminmaxdist* (equals *minmaxdist* (q, N_6)) and *Dmaxdist* (equals *maxdist* (q, N_6)), which are denoted by solid lines together with arrows. Evidently, these distances satisfy above Lemma 1 and Lemma 2. In other words, the following two inequalities, i.e., *mindist* (q, N_{10}) < D_n (q, g) < *minmaxdist* (q, N_6) and *mindist* (q, N_{10}) < D_n (q, g) < *maxdist* (q, N_6), hold.

At the same time, we can also know that the NN of q distributes between *Dmindist* and *Dminmaxdist*. Continuing the above example, for instance, in Figure 3.1, the NN of q (i.e., g) distributes at the area of between *circle* 1 and *circle* 2 (representing as the shaded area, called *search region*), which centred at q with radius *Dmindist* (i.e., *mindist* (q, N_{10})) and *Dminmaxdist* (i.e., *minmaxdist* (q, N_6)), respectively. It must be noted that the NN of q lies in between *Dmindist* and *Dmaxdist* as well, because the *Dmaxdist* is not smaller than the *Dminmaxdist*. Here, a problem arises, i.e., how to find the NN of q as quick as possible. Intuitively, assuming that the direction of the NN search is unique during the processing of the NN queries, three search strategies can be used to quickly discovery the NN of q, that is, (i) from *circle* 1 to *circle* 2, (ii) from intermediate to the sides of *circle* 1 and those of *circle* 2, and (iii) from *circle* 2 to *circle* 1. Figure 3.1 also shows these three directions of the NN search that are illustrated by dashed arrows. However, which one is the best choice among them? The following analysis answers this issue.

Let T be the total time of retrieving *search region*, T_m be the time of searching area between the side of *circle* 1 and that of circle which falls into the *search region*. There are following three cases together with analysis of their search time:

Case 1: Following strategy 1 for searching the NN of q. The best situation is that the NN of q just distributes on the side of *circle* 1. Hence, algorithm can immediately find the NN of q. On the contrary, as the NN of q just falls on the side of *circle* 2, the algorithm need cost T time to discovery the NN of q, which is also just the worst case. Nevertheless, note that above worst situation little occurs in real life, since it depends on a particular configuration of both data objects and query object. Therefore, we can get the time range [0, T] towards the NN search algorithm that follows the strategy 1 for retrieving the NN of q. Furthermore, the average search time of this case is $T/2$.

Case 2: Finding the NN of q according to the strategy 2. As the analysis of the case 1, we can derive the time range

[T_m, T] concerning the algorithm for NN search which follows the strategy 2 to get the NN of q. Moreover, the mean search time of this case is $(T_m + T)/2$.

Case 3: Discovering the NN of q by the strategy 3. Similarly, we can also get the time range [T, T], and the average search time of this case is T.

To summarize above discussion, we use the first search strategy (i.e., retrieving the NN of q from *circle* 1 to *circle* 2) in the proposed algorithms to obtain the NN of q as soon as possible, since its cost is smaller than that of other search strategies.

4. BFPNN and FBFPNN algorithms

Section 4.1 introduces several pruning strategies which permit the development of efficient algorithm presented in Section 4.2. Section 4.3 gives an illustrative example for simulating the execution of the BFPNN algorithm and analyzes its performance with respect to BF algorithm for NN search.

4.1 Pruning strategies

Like the BF algorithm discussed in Section 2, BFPNN and FBFPNN algorithms also employ *branch-and-bound* techniques to prune the *search space*. Specifically, starting from the root, the parallel R-tree is traversed by using the following principles: (i) when a leaf entry (i.e., a data point) p is encountered, the NN of a given query point q is found if p contains the minimum distance to q; (ii) for an intermediate entry, algorithms visit its subtree only if it may enclose any qualifying data points, which contribute to the final answer. The advantage of this solution over exhaustive scan is that it avoids accessing unnecessary nodes, which do not act on the final result. In the sequel, we present several heuristics for pruning these needless nodes.

According to the Lemma 1 discussed in Section 3, we can formulate the following strategies to prune node entries during processing of the NN retrieval:

Heuristic 1: Given a query point q, a node entry E with *mindist* (q, E) greater than *Dminmaxdist* can be safely discarded since it can not contain the NN of q (according to the Lemma 1).

In Figure 3.1, for instance, the node entries (i.e., N_4, N_5, N_7, N_{11} and N_{12}) can be quickly pruned in the phase of the NN search processing, because their distances (i.e., *mindist*(s)) to q are larger than *Dminmaxdist* (equals *minmaxdist* (q, N_6)), which is the smallest *minmaxdist* among all node entries (involving N_4 through N_{12}), but the algorithms need visit those node entries (i.e., N_6, N_8, N_9 and N_{10}) intersecting with the *search region* that lies in between *circle 1* and *circle 2* centered at q with radius *Dmindist* and *Dminmaxdist*, respectively. Applying the heuristic 1, the algorithm can avoid accessing unnecessary node (e.g., N_4, N_5, etc.), leading to quickly finding the NN of q.

Heuristic 2: Given a query point q, if an actual distance from q to a data point p is larger than *Dminmaxdist*, then it can be safely pruned because p can not be the NN of q (by the Lemma 1).

As known that N_6, N_8, N_9 and N_{10} need to be retrieved in the phase of NN search depending on the *heuristic 1*. Therefore, continuing above example, applying the *heuristic* 2 again within those qualifying node entries (i.e., N_6, N_8, N_9 and N_{10}), the data points (i.e., f, j, k, m, o and p) contained in Figure 3.1 do not have to be accessed during the processing of the NN search, since their actual distances from q are also bigger than *Dminmaxdist*. However, algorithms need retrieve those data points (i.e., g, l and n) which fall into the *search region* until finding the NN of q (i.e., g). Therefore, the algorithms can reduce the amount of accessed nodes and speed up their execution through using the *heuristic* 2 as well.

Similarly, we can derive other two strategies for pruning the unnecessary node entries during the processing of the NN retrieval in terms of the Lemma 2 described in the Section 3:

Heuristic 3: Given a query point q, a node entry E that satisfies the inequality *mindist* (q, E) > *Dmaxdist* can be safely discarded since it can not enclose the NN of q (relying on the Lemma 2).

Heuristic 4: Given a query point q, if an actual distance between q and a data point p is bigger than *Dmaxdist*, then it can be safely pruned because p is not the NN of q (on the basis of the Lemma 2).

To summarize aforementioned discussion, the proposed algorithms in this paper can not only avoid retrieving non-qualifying nodes (i.e., unnecessary nodes) so that they can find the NN of q as soon as possible, but also reduce the number of accessed nodes and speed up their executions as well, by employing the *heuristic* 1 through the *heuristic* 4.

4.2 Algorithms

This subsection presents the first Best-First based Parallel Nearest Neighbor (BFPNN for short) search algorithm and Full BFPNN (FBFPNN for short) algorithm for NN queries in spatial databases, associating with above four pruning strategies.

Specifically, the proposed BFPNN algorithm implements an ordered best first traversal. It begins with the parallel R-tree root node and proceeds down the tree. First of all, algorithm inserts all entries in the root node into their corresponding priority queues (e.g., heaps) that they reside, and records current minimum *minmaxdist* (i.e., *Dminmaxdist*) within all entries for pruning unnecessary node entries in sequel. Then algorithm

iterates until either all queues for M disks (assuming that the number of disks is M) are empty or the algorithm finds the NN of a given query point q: Each iteration, algorithm applies *heuristic 1* and *heuristic 2* to discard useless node entries firstly. Then it finds the entry E with the minimum distance to q among all queues by parallel process. In practical implementation, algorithm gets all entries EH_i(s) at the head of each priority queue firstly suppose that EH_i(s) is (are) maintained in ascending order with respect to its (their) *mindist*(s). Next it finds the entry E with the smallest distance to q among EH_i(s). Here, E has two possibilities, that is (i) E is an data object, then algorithm reports it as the NN of q and terminates the algorithm; otherwise (ii) E is an intermediate node entry, then algorithm deals with M disks in parallel way as follows: if each queue Q_i is not empty and at the head of Q_i is not a data object, then algorithm dequeues Q_i and enqueues all entries of it into corresponding queues that they store. Furthermore, algorithm also records and updates the current value of *Dminmaxdist*, when it is smaller than its old value, such that the algorithm can discard more node entries that do not contain any qualifying data point (in other words, not contributing to the final answer) in the next iteration.

To summarize aforementioned description of the BFPNN algorithm, Figure 4.1 presents the pseudo-code description of a BFPNN algorithm. In particular, the inputs of the BFPNN algorithm involve a given query point q, parallel R-tree indexing structure, and the number of disks M; at the same time, its output is the NN of q. Specifically, line 1 constructs and initializes priority queues for M disks. Line 7 applies *heuristics* 1 and 2 (see section 4.1) to prune all unnecessary node entries. In line 10, the NN of q is reported. At that point, some other routines (e.g., a query engine) can take control, possibly resuming the algorithm at a later time to get the next NN of q, or alternately terminating it if no more NNs of q are desired. From line 20 to line 21, the value of *Dminmaxdist* is updated, such that it consistently maintains the minimum *minmaxdist* distance from q. In addition, it must be noted that the line 13 through line 21 can be performed by means of parallel process.

BFPNN (QueryObject *q*, Parallel R-tree, M)

/* M is the total number of disks; *Dminmax* denotes the minimal *minmaxdist* among all *minmaxdist*s from q to data points; Q_1, Q_2,…, and Q_M are maintained in ascending order with respect to *mindist*; *Temp_Dminmax* stores temporal value of *Dminmax*; FindDisk (E) return the identifier of disk residing entry E; First (Q_i) return the head entry in queue Q_i; *Dist* (q, E) calculates the *mindist* between q and E. */

1. Construct and initialize M priority queues;
2. For each entry E in the root node
3. i = FindDisk (E);
4. EnQueue (Q_i, E, *Dist* (q, E));
5. *Dminmax* = minimum *minmaxdist* of all entries;
6. While existing queue (s) is (are) not empty
7. Prune all entries for each queue according to the *heuristics* 1 and 2;
8. Find the entry E_{min} with the minimum distance from q in all queue (s);
9. If E_{min} is an data object then
10. Report the data object as the NN of q;
11. Return;
12. Else // E_{min} is an intermediate node.
13. For i = 1 to M parallel do
14. If not IsEmpty (Q_i) and First (Q_i) is not an data object then
15. N_i = DeQueue (Q_i);
16. For each entry E in N_i
17. j = FindDisk (E);
18. EnQueue (Q_j, E, *Dist* (q, E));
19. *Temp_Dminmax* = minimal *minmaxdist* in all entries;
20. If *Temp_Dminmax* < *Dminmax* then
21. *Dminmax* = *Temp_Dminmax*;
22. Enddo

End BFPNN

Figure 4.1: Pseudo-code of a BFPNN algorithm

As a matter of fact, the FBFPNN exactly like the BFPNN. Figure 4.2 proposes the pseudo-code description of a FBFPNN algorithm as well. Notice that most of the pseudo-code presentations of the FBFPNN algorithm are the same as above those of the BFPNN algorithm. However, the differences between them are discussed as follows: (i) the former (i.e., FBFPNN algorithm) employs *heuristics* 3 and 4 (see line 7 in Figure 4.2) for pruning all unnecessary node entries, while the latter (i.e., BFPNN algorithm) applies *heuristics* 1 and 2 (see line 7 in Figure 4.1) to discard all needless ones during the NN search processing; and (ii) for FBFPNN algorithm, the value of *Dmaxdist* is renovated from line 20 to line 21 in Figure 4.2, thus it always keeps the smallest *maxdist* distance to q, whereas towards BFPNN algorithm, the value of *Dminmaxdist* is updated in line 20 through line 21 of the Figure 4.1, such that it consistently maintains the minimum *minmaxdist* distance from q.

FBFPNN (QueryObject *q*, Parallel R-tree, M)

/* *Dmax* denotes the minimal *maxdist* among all *maxdist*(s) from q to data points; *Temp_Dmax* stores temporal value of *Dmax*. */

1. Construct and initialize M priority queues;
2. For each entry E in the root node
3. i = FindDisk (E);
4. EnQueue (Q_i, E, *Dist* (q, E));
5. *Dmax* = minimum *maxdist* of all entries;
6. While existing queue (s) is (are) not empty
7. Prune all entries for each queue according to the *heuristics* 3 and 4;
8. Find the entry E_{min} with the minimum distance from q in all queue (s);
9. If E_{min} is an data object then
10. Report the data object as the NN of q;
11. Return;
12. Else // E_{min} is an intermediate node.

13. For i = 1 to M parallel do
14. If not IsEmpty (Q_i) and First (Q_i) is not an data object then
15. N_i = DeQueue (Q_i);
16. For each entry E in N_i
17. j = FindDisk (E);
18. EnQueue (Q_j, E, *Dist* (q, E));
19. *Temp_Dmax* = minimal *maxdist* in all entries;
20. If *Temp_Dmax* < *Dmax* then
21. *Dmax* = *Temp_Dmax*;
22. Enddo

End FBFPNN

Figure 4.2: Pseudo-code of a FBFPNN algorithm

4.3 Discussion

This subsection only gives an illustrative example to simulate the execution of the BFPNN algorithm for NN search, and omits that of the FBFPNN algorithm, since the runnings of both algorithms are similar. Secondly, we compare the efficiency of the BFPNN algorithm with traditional BF algorithm. However, further performances (e.g., mean number of accessed nodes, mean query cost (time), etc.) of the both BFPNN and FBFPNN algorithms with respect to existing parallel NN search algorithms are systematically examined in the Section 5 of this paper.

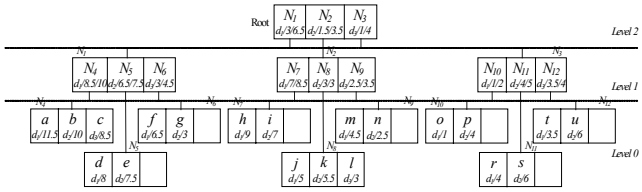


Figure 4.3: The parallel R-tree for Figure 2.3 (a)

Consider, for instance, Figure 2.3 (a), where shows a set of points {a, b, ..., u} indexed by a parallel R-tree (illustrated in Figure 4.3) assuming a capacity of three entries per node (i.e., the fanout of each node is three). As an example, suppose that here wants to find the NN of a given query point q in the parallel R-tree, where nodes are numbered from N_1 to N_{12}; the symbols d_1, d_2, d_3 in each entry indicates *disk* 1, *disk* 2 and *disk* 3, respectively; the one two digits (e.g., 3/6.5, 1.5/3.5, etc.) for per entry within *level* 1 and *level* 2 refer to the *mindist* and *minmaxdist* to q respectively (for intermediate entries), whereas the number (e.g., 11.5, 10, etc.) for each entry within *level* 0 specifies the actual distance to q (for data objects). It must be noted that these numbers are not stored previously, while computed dynamically during the NN search processing. Below, we show the steps of the execution of the BFPNN algorithm together with the contents of each priority queue (representing as Q_1, Q_2 and Q_3 in this example) and the values of some parameters (e.g., *Dminmax*, E_{min}, *Temp_Dminmax*, etc.). In addition, notice that all entries within each priority queue are listed in ascending order with respect to *mindist* to q. The BFPNN algorithm starts by enqueuing the root node in parallel way, after which it performs the following steps:

1. EnQueue (Q_1, N_1, 3), EnQueue (Q_2, N_2, 1.5), EnQueue (Q_3, N_3, 1).
   Dminmax = *minmaxdist* (q, N_2) = 3.5,
   E_{min} = Dist (q, N_3) = 1.
   DeQueue (Q_1, N_1),
   EnQueue (Q_1, N_4, 8.5), EnQueue (Q_2, N_5, *6.5*),
   EnQueue (Q_3, N_6, 3);
   DeQueue (Q_2, N_2),
   EnQueue (Q_1, N_7, 7), EnQueue (Q_2, N_8, 3),
   EnQueue (Q_3, N_9, 2.5);
   DeQueue (Q_3, N_3),
   EnQueue (Q_1, N_{10}, 1), EnQueue (Q_2, N_{11}, 4),
   EnQueue (Q_3, N_{12}, 3.5).
   Queues: Q_1: {(N_{10}, 1), (N_7, 7), (N_4, 8.5)},
   Q_2: {(N_8, 3), (N_{11}, 4), (N_5, 6.5)},
   Q_3: {(N_9, 2.5), (N_6, 3), (N_{12}, 3.5)}.
   Temp_Dminmax = *minmaxdist* (q, N_{10}) = 2,
   Update *Dminmax* = *Temp_Dminmax*
   = *minmaxdist* (q, N_{10}) = 2.
2. Prune N_7 and N_4 from Q_1 since their *mindist*(s) (equals 7 and 8.5, respectively) are greater than *Dminmax* (equals 2);
   Discard N_8, N_{11} and N_5 from Q_2 because their *mindist*(s) (equals 3, 4 and 6.5, respectively) are larger than *Dminmax* (equals 2);
   Prune N_9, N_6 and N_{12} from Q_3 as their *mindist*(s) (equals 2.5, 3 and 3.5, respectively) are bigger than *Dminmax* (equals 2).
   E_{min} = Dist (q, N_{10}) = 1.
   DeQueue (Q_1, N_{10}), EnQueue (Q_1, o, 1),
   EnQueue (Q_2, p, 4).
   Queues: Q_1: {(o, 1)}, Q_2: {(p, 4)}, Q_3: empty.
   Do not update *Dminmax* since current *Temp_Dminmax* (equals 2) is not smaller than old *Dminmax* (equals 2).
3. Prune p from Q_2 as its *mindist* (equals 4) is larger than *Dminmax* (equals 2).
   E_{min} = Dist (q, o) = 1

   Report o is the NN of q.

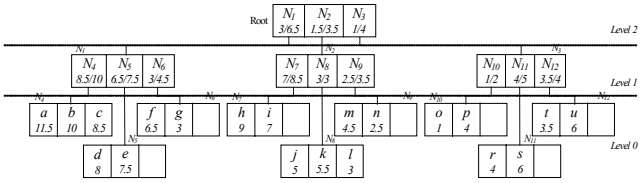


Figure 4.4: The R-tree for Figure 2.3 (a)

Continuing above example, suppose that we also use traditional BF algorithm (i.e., incremental NN algorithm) to find the NN of q in the R-tree shown in Figure 4.4, where the meanings of the number in each entry are as the same as above Figure 4.3. The BF algorithm begins with by enqueuing the root node, after which it executes the following steps:

1. DeQueue Root, EnQueue N_1, N_2 and N_3.
   Queue: {(N_3, 1), (N_2, 1.5), (N_1, 3)}
2. DeQueue N_3, EnQueue N_{10}, N_{11} and N_{12}.
   Queue: {(N_{10}, 1), (N_2, 1.5), (N_1, 3), (N_{12}, 3.5), (N_{11}, 4)}

3. DeQueue N_{10}, EnQueue o and p.
   Queue: {(o, 1), (N_2, 1.5), (N_1, 3), (N_{12}, 3.5), (p, 4), (N_{11}, 4)}
4. DeQueue o. The distance of o is 1, which is not larger than the distance of N_2, so o is reported as the NN of q.
   Queue: {(N_2, 1.5), (N_1, 3), (N_{12}, 3.5), (p, 4), (N_{11}, 4)}

Observe that the BFPNN algorithm outperforms the conventional BF algorithm in terms of query cost (time) through above instance. Actually, the former is better than the latter in most cases, especially for larger dataset. In particular, the advantages of the BFPNN algorithm mainly involve: (i) fetch more nodes at the same time, considering above example again, for example, the BFPNN algorithm can access the nodes N_4 through N_{12} in the first step simultaneously; and (ii) prune more unnecessary nodes, for instance, the BFPNN algorithm discards all node entries besides N_{10} in the step two by utilizing the *heuristic* 1 and *heuristic* 2. Therefore, the BFPNN algorithm can efficiently reduce the query cost, so as to speed up its executive process.

5. Experiments

In this section, we perform considerable experiments to evaluate the efficiency of the proposed algorithms (including BFPNN and FBFPNN algorithms) and compare them against the existing parallel NN search algorithms (e.g., CRSS), using three real datasets (summarized in Table 5.1) and three synthetic datasets. Specifically, three real datasets involve *CA*[1] that contains 2-dimensional points representing geometric locations in California, *Wave* [2] including the 3-dimensional measurements of 60k wave directions at the National Buoy Center, and *Color* [3] which consists of the 4-dimensional color histograms of 65k images. Toward above real datasets, we normalize each dimension of the data space to range [0, 10000]. In addition, we also create three synthetic datasets following the *Uniform*, *Gaussian* and *Zipf* distributions, respectively. In particular, the coordinates of each point in a *Uniform* dataset are generated randomly in [0, 10000], whereas, for *Gaussian* dataset, the coordinates are generated randomly in [5000, 250], and the coordinates of each point in a *Zipf* dataset follow a *Zipf* distribution with a skew coefficient 0.8. Noted that when the skew coefficient equals 1, all numbers generated by the *Zipf* distribution are equivalent, while the *Zipf* distribution degenerates to uniformity, as the coefficient equals 0. With respect to all aforementioned datasets, the coordinates of a point on various dimensions are mutually independent in all cases.

[1] *CA* (dataset) can be downloaded from *http://www.census.gov/geo/www/tiger*.
[2] *Wave* (dataset) can be downloaded from *http://www.ndbc.noaa.gov*.
[3] *Color* (dataset) can be downloaded from *http://www.cs.cityu.edu.hk/~taoyf/ds.html*.

	CA	***Wave***	***Color***
Dimensionality	2	3	4
Cardinality	62k	60k	65k

Table 5.1: Statistics of the three real datasets used

Every dataset is indexed by a parallel R-tree [30] which is distributed among multiple disks and disk assignment straightforwardly follows the *Round Robin (RR) heuristic*. Furthermore, the node size of the parallel R-tree is fixed to 1024 bytes. Therefore, the node capacity (i.e., the maximum number of entries in a node) equals 50, 36, 28 and 23 entries for dimensionalities 2, 3, 4, and 5, respectively. All parameters used in our experimental assessment are presented in Table 5.2.

Parameters	**Description**	**Assigned Value**
S_{node}	Node capacity	1024 bytes
dim	Space dimensionality	2□5
N	dataset cardinality	≥ 60k
k	Number of NNs	1□81
disks	Number of disks	2□10

Table 5.2: Description of all parameters in experiments

The experiments investigate the effect of the following parameters: (i) *disks*, (ii) *dim* and (iii) *N*. Performance is measured by executing workloads, each consisting of 100 queries generated as follows: the locations of the queries are uniformly distributed in the corresponding data space. Moreover, the reported results represent the average cost per query for a workload with the same parameters. Notice that the query cost is calculated as the sum of the CPU time and I/O overhead that is computed by charging 10ms for each node access. All the experiments are conducted on a Pentium IV 3.0 GHz CPU, 2048 Mega bytes memory and 160 Giga disk (whose model is *"Maxtor 6Y160LO"*). Furthermore, all the algorithms are coded in C++. Additionally, since it is very difficult to provide experimental results by modifying all parameter values, we only choose to illustrate representative results that shed light on the effectiveness and scalability of our proposed algorithms. Moreover, the proposed algorithms are only in comparison with the CRSS, because the efficiency of the CRSS is the best among existing parallel NN search algorithms.

The first set of experiments evaluates the number of accessed nodes and query cost (in second) of parallel algorithms for single NN search as a function of *disks* (changing from 2 to 10) through various real and synthetic datasets. Figures 5.1 and 5.2 illustrate these experimental results. Furthermore, in Figure 5.2, the number above each column indicates the percentage of I/O cost in the total query cost. Clearly, the performance of the BFPNN resembles that of the FBFPNN for most datasets, since the basic ideas of both algorithms are similar and the difference only lies in the pruning strategies (discussed in Section 4). However, evidently, the efficiency of them outperforms that of the CRSS, as illustrated the two

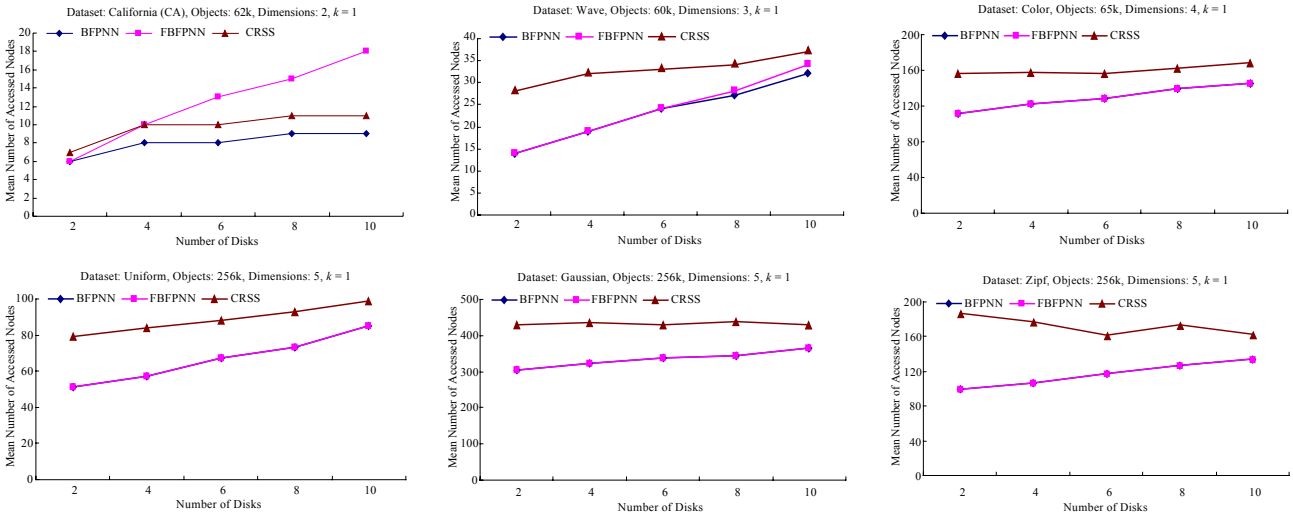


Figure 5.1: Number of accessed nodes VS. Number of disks

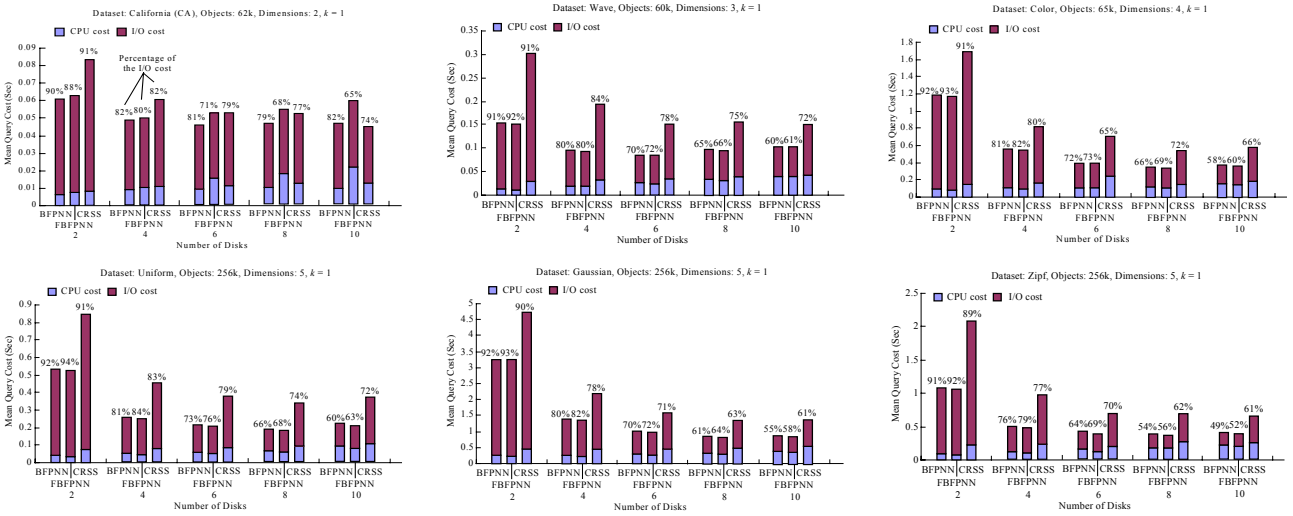


Figure 5.2: Mean Query Cost (Sec) VS. Number of disks

diagrams. Specifically, as number of disks increases, the number of accessed nodes ascends, because we simply employ *Round Robin* heuristic to assign disks in our experimental setting. This case does not guarantee that the node entries that will be retrieved by the same query are distributed to diverse disks in order to augment the parallelism. Nevertheless, assuming that the node entries are distributed into multiple disks according to the *proximity measure*, which allocates all node entries visited by the same query to the different disks, and hands out those accessed by the distinct queries to the same disk, such that obtaining higher parallelism (i.e., fetching more node entries from the disks that they reside simultaneously for per retrieval.). This is just one part of our further work as well. Similarly, the query cost for each workload decreases along with the ascending of the number of disks, as the parallelism increases gradually. Moreover, all the algorithms are I/O bounded. However, as the number of disks grows, the CPU cost of all demonstrating algorithms accounts for a larger fraction of the total query cost (indicated by its decreasing with the percentage of the I/O cost). This situation is depicted in Figure 5.2.

The next set of experiments inspects the influence of the dimensionality. Towards this, we deploy two synthetic datasets including *Gaussian* and *Zipf* that contain 256k and 512k data points (i.e., cardinality N = 256k and N = 512k) of dimensionality varying between 2 and 5, respectively. Furthermore, the parameter "*disks*" is fixed to the value 2. Figures 5.3 and 5.4 compare the number of accessed nodes and the query cost of the BFPNN and the FBFPNN against those of the CRSS in answering single NN search. As expected, the performance of all algorithms degrades because, in general, R-trees become less efficient with the growing of dimensionality [24] (due to the larger overlap among the MBRs at the same level).

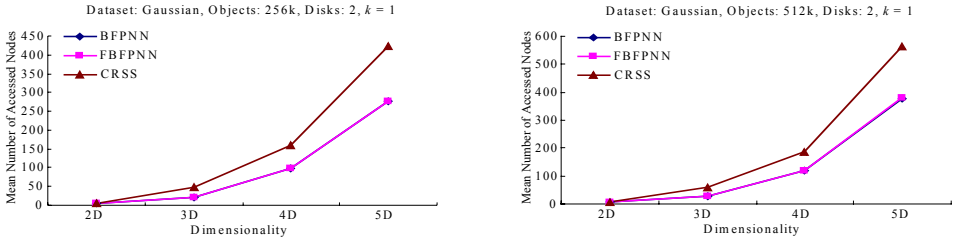


Figure 5.3: Number of accessed nodes VS. Dimensionality

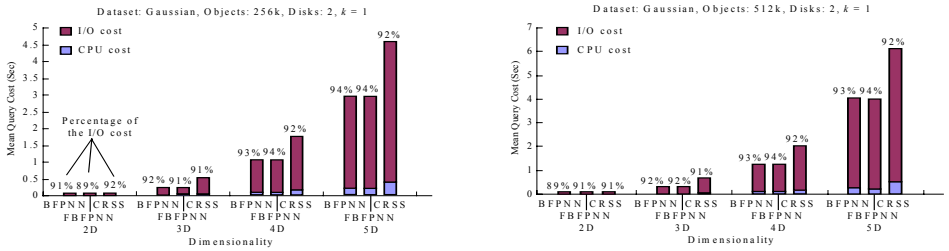


Figure 5.4: Mean Query Cost (Sec) VS. Dimensionality

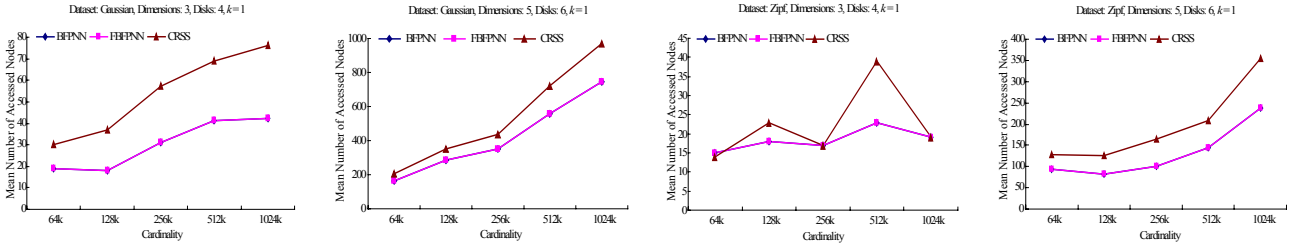


Figure 5.5: Number of accessed nodes VS. Cardinality

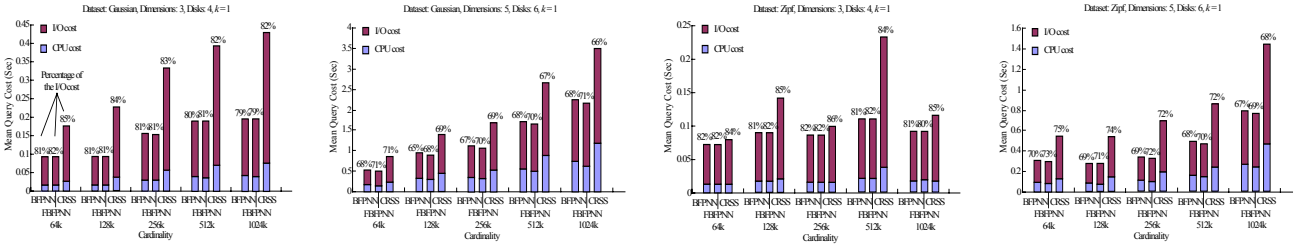


Figure 5.6: Mean Query Cost (Sec) VS. Cardinality

Moreover, the total query cost increases as the increasing of the dimensionality, since the I/O cost occupies a greater portion of the total query cost, that is, the algorithms have to spend more time in accessing all required node entries which act on the final outcome for finding the NN of a given query point. However, obviously, the efficiency of the BFPNN still exceeds that of the CRSS. At the same time, as aforementioned set of experiments, the performance of the BFPNN is similar to that of the FBFPNN no matter the dimensionality is low or high. In addition, for all the datasets, notice that all examining algorithms show similar efficiency in low dimensionality (e.g., 2D), but both the BFPNN and the FBFPNN are still more effective than the CRSS in high dimensionality (e.g., 3D, 4D, etc.).

Finally, to study the effect of the dataset cardinality, we use 3-dimensional (5-dimensional) *Gaussian* and *Zipf* datasets whose cardinality range alter from 64k to 1024k, by fixing *disks* the values 4 and 6, respectively. Figures 5.5 and 5.6 demonstrate these experimental results of BFPNN, FBFPNN and CRSS for single NN search as a function of the cardinality size. Like above two sets of experiments, the capability of the BFPNN is better than that of the CRSS under all the cases. Furthermore, the efficiency of both the BFPNN and the FBFPNN is similar regardless of the size of cardinality. Specifically, as cardinality ascends, the number of accessed nodes adds, since algorithms need retrieve more node entries for discovering the NN of a given query point q. Similarly, the query cost also augments along with the growing of the cardinality, because the algorithms have to expend

more cost to process NN retrieval. In particular, the CPU cost of all illustrating algorithms becomes higher, as the algorithms need access more necessary nodes which contribute to the final answer as well for finding the NN of q. In addition, as shown in two diagrams, the step-wise cost growth corresponds to an increase of the magnitude of the cardinality between 64k and 1024k. For instance, for 3-dimensional *Zipf* dataset, the increment evidently occurs at cardinality 128k and 512k, whereas the growth obviously arises at cardinality 1024k with respect to 5-dimensional *Zipf* dataset.

6. Conclusion

The NN search is one of the most important operations in spatial databases and spatio-temporal databases. Motivated by most existing methods for NN queries focus on a single disk to find the NN of a given query point, the problems of the number of accessed nodes and query cost become severe with the increasing volume of datasets, especially for the dataset not fitting in one disk. To address these problems, this paper presents the first Best-First based Parallel NN (BFPNN) query algorithm and Full BFPNN (FBFPNN) algorithm for effective processing of the NN retrieval, by means of parallelism (i.e., fetching more nodes or data points from multiple disks simultaneously) in multi-disk setting. Furthermore, an extensive experimental comparison verifies that, in addition to correctness, the proposed algorithms outperform the previous techniques in terms of efficiency and scalability under most cases, by using various real and synthetic datasets.

In the future, some promising directions for future work mainly include the following issues: (i) extend the presented algorithms to support kNN queries since the BFPNN and FBFPNN algorithms only concentrate on the 1NN search; (ii) study on diverse disk assignment approaches to enhance parallelism during the processing of NN retrieval; (iii) derive analytical models for estimating the execution cost of the parallel NN search algorithms, such that facilitating query optimization and revealing new problem characteristics that could lead to even faster algorithms.

Acknowledgements

This research was supported by the National High Technology Development 863 Program of China under Grant No. 2003AA4Z3010-03.

References

[1] Henrich, A. A Distance-Scan Algorithm for Spatial Access Structures. In *ACM GIS*, 1994.

[2] Roussopoulos, N., Kelley, S., Vincent, F. Nearest neighbor queries. In *SIGMOD*, 1995.

[3] Cheung, K.L., Fu, A.W.-C. Enhanced Nearest Neighbour search on the R-tree. *ACM SIGMOD Record*, 27: 16-21, 1998.

[4] Hjaltason, G.R., Samet, H. Distance Browsing in Spatial Databases. *ACM TODS*, 24: 265–318, 1999.

[5] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. In *SSTD*, 2001.

[6] Tao, Y., Papadias, D. Time-Parameterized Queries in Spatio-Temporal Databases. In *SIGMOD*, 2002.

[7] Tao, Y., Papadias, D., Shen, Q. Continuous Nearest Neighbor Search. In *VLDB*, 2002.

[8] Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, 2002.

[9] Iwerks, G.S., Samet, H., Smith, K. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *VLDB*, 2003.

[10] Xiong, X., Mokbel, M.F., Aref, W.G. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, 2005.

[11] Papadias, D., Tao, Y., Kyriakos, M., Chun, K.H. Aggregate Nearest Neighbor Queries in Spatial Databases. *ACM TODS*, 2005. (to appear)

[12] Yiu, M.L., Mamoulis, N., Papadias, D. Aggregate Nearest Neighbor Queries in Road Networks. *TKDE*, 17: 820-833, 2005.

[13] Korn, F., Muthukrishnan, S. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD*, 2000.

[14] Stanoi, I., Agrawal, D., Abbadi, A. Reverse Nearest Neighbor Queries for Dynamic Databases. In *SIGMOD Workshop DMKD*, 2000.

[15] Yang, C., Lin, K-I. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *ICDE*, 2001.

[16] Maheshwari, A., Vahrenhold, J., Zeh, N. On reverse Nearest Neighbor Queries. In *CCCG*, 2002.

[17] Singh, A., Ferhatosmanoglu, H., Tosun, A. High Dimensional Reverse Nearest Neighbor Queries. In *CIKM*, 2003.

[18] Tao, Y., Papadias, D., Lian, X. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, 2004.

[19] Papadopoulos, A.N., Manolopoulos, Y. Similarity Query Processing Using Disk Arrays. In *SIGMOD*, 1998.

[20] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[21] Sellis, T., Roussopoulos, N., Faloutsos, C. The R^+-tree: A Dynamic Index for Multi-dimensional Objects. In *VLDB*, 1987.

[22] Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.

[23] Kamel, I., Faloutsos, C. Parallel R-trees. In *SIGMOD*, 1992.

[24] Theodoridis, Y., Sellis, T.K A Model for the Prediction of R-tree Performance. In *PODS*, 1996.