Parallel All Nearest Neighbor Searching in High Dimensional Spaces

Majid Ghaderi Mohammad Ghodsi
Computer Engineering Department
Sharif University of Technology

{majid, ghodsi}@ce.sharif.edu

November 2001

Abstract

Consider a set P of n points in real d-dimensional space $\mathbb{R}^d$, where distances are measured using any Minkowski metric. In this paper, using $O(n)$ processors in CREW PRAM model, a BBD-tree of point set is built in time $O(\log^2(n))$. This tree has $O(n)$ nodes, each of which has the complexity $O(d)$. Based on this tree, we will represent a parallel algorithm for all nearest neighbors problem in P. Query time of this algorithm is $O(\log^2(n))$ in CREW PRAM model.

1 Introduction

Given a set P of n points $\{p_1, \ldots, p_n\}$ in $\mathbb{R}^d$, *All Nearest Neighbor Search* problem is defined as follow: for each point q in P, find its nearest neighbor in P. The notion of *nearest*, is defined based on the distance measures in metric space $\mathbb{R}^d$. Each Minkowski measure can be used to compute inter point distances. For an integer number $m \geq 1$, L_m-distance between two points $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ is defined as follow

$$d(p,q) = \sqrt[m]{\sum_{i=1}^{d} (|p_{(i)} - q_{(i)}|)^m}$$

With this definition L_2 is the famous Euclidean space.

Nearest Neighbor problem is applied to a variety of computing fields as a major technique. For example in Information Retrieval [5], Pattern Recognition [6], Data Compression [6] and Multimedia Databases [8], this technique has been used. Objects are mapped to points in a multi dimensional space. Typically, dimensionality of these spaces is high and the number of points also is high. For example Multimedia Database applications include thousands of points in a space with hundreds of dimensions [8].

Nearest Neighbor Search (NNS) in these applications is very time consuming because of the high dimensions and large number of points. Therefore presenting a solution, which can lead to a fast search is of great importance. One of these techniques is to use multi processors to do such searches. Unfortunately in most cases we can not easily run the serial algorithms on multi processors. Even it is possible that a

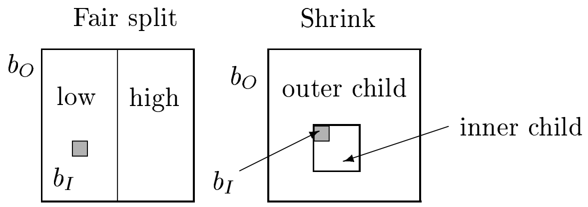

Figure 1: inner box vs. outer box

good serial algorithm does not have an acceptable performance on multi processors. In these situations, designing a parallel algorithm is the only solution.

In this paper we will present a parallel algorithm for the problem, which leads to search time $O(exp(d)\log(n))$ with the help of $O(n)$ processors in CREW PRAM model. Our algorithm uses a data structure that is the well-known BBD-tree. Preprocessing work needed to construct this data structure is proportional to $O(d\log^2 n)$. From the storage view point, our method uses $O(dn)$ storage to construct and maintain the BBD-tree.

The rest of the paper organized as follow. First, in section 2 we define the concepts and terminologies in this paper. In section 3 we will review the previous works in this area. In sections 4 and 5 we deal with parallel construction of FS-tree and BBD-tree, respectively. Section 6 describes our parallel algorithm for all nearest neighbor problem and represents its analysis. Finally section 7 is the conclusion of this paper.

2 Definitions

Let P be a set of n points in $\mathbb{R}^d$, where d is a constant in this paper. We define the *bounding rectangle* of P, denoted by $R(P)$, to be the smallest rectangle that encloses all points in P, where the word "rectangle" denotes any cartezian product $R = [x_1, x'_1] \times [x_2, x'_2] \times \cdots \times [x_d, x'_d]$ in $\mathbb{R}^d$. We use the term *open rectangle* to denote the product of open intervals. We denote the length of R in the ith dimension by $l_i(R) = x'_i - x_i$. We denote the maximum and minimum lengths by $l_{max}(R)$ and $l_{min}(R)$. When all $l_i(R)$ are equal, we say that R is a *hypercube*, and denotes its length by $l(R) = l_{max}(R) = l_{min}(R)$. We write $l_i(P)$, $l_{min}(P)$, and $l_{max}(P)$ as shorthand for $l_i(R(P))$, $l_{min}(R(P))$, and $l_{max}(R(P))$, respectively. We define a *ball* of radius r to be the set of all points in d dimensions at a distance less than or equal to r from some point, referred to as the center of the ball.

The size of a rectangle is the length of its longest side. By *aspect ratio* we mean the ratio between the longest and shortest sides of a rectangle. We define a *box* to be a rectangle whose aspect ratio is bounded by some constant. In particular, define a *cell* to be either a box or the set theoretic difference of two boxes, one enclosed within the other. Thus each cell is defined by an outer box and an optional inner box (See Figure 1). Each cell will be associated with the set of data points lying within the cell. Cells are considered to be closed, and hence points which lie on the boundary between cells may be assigned to either cell. The size of a cell is the size of its outer box.

An important concept which restricts the nature of inner boxes is a property called stickiness. Consider a cell with outer box b_O and inner box b_I . Intuitively, the box b_I is sticky for b_O if each face is either sufficiently far from or else touching the corresponding face of b_O . To make this precise, consider two closed intervals, $[x_I, y_I] \subset [x_O, y_O]$, and let $w = y_I - x_I$ denotes the width of the inner interval. We say that $[x_I, y_I]$ is *sticky* for $[x_O, y_O]$ if each of the two distances between the inner interval and the outer interval, $x_I - x_O$ and $y_I - y_O$, is either 0 or at least w . The inner box b_I is sticky for the outer box b_O if each of the d intervals of b_I is sticky for the corresponding interval of b_O . An equivalent condition for stickiness arises by considering the 3^d regular grid of copies of b_I , centered around b_I . Observe that b_I is sticky for b_O if and only if each rectangle in this grid either lies entirely within b_O or is disjoint from the interior of b_O.

Throughout, we maintain the property that for all cells the inner box is sticky for the outer box. Stickiness is needed for various technical reasons, which will be discussed later. In particular, it prohibits situations in which a large number of inner boxes are nested one within the next, and all are extremely close to the outer wall of a cell. In situations like this it will not be possible to prove our bounds on the search time.

3 Related Works

Typically, search algorithms are based on partitioning the space and constructing a search tree. Spatial search trees are the generalized version of the same binary search tree. In the other words, these are binary trees, which their nodes represent a region of the space. These trees are divided in two categories, Balanced trees and Bounded Aspect Ratio trees.

In Bounded Aspect Ratio trees, splitting operation is done such that always the aspect ratio will be less than a constant value. With the help of this property, it can be proved that the number of leaves overlap a ball of a definite radius, is bounded by a constant value independent of the total number of points. This property is called *packing constrain*. The efficiency of the search algorithm has a reverse relation with the number of points visited in order to find the nearest neighbor of a query point. Therefore if we can find a constant bound for this value, the efficiency of search algorithm will be improved.

In [3] a kind of partitioning, named *fair-split*, is introduced and based on this notion a fair-split tree is constructed. Splitting hyperplane is orthogonal to the coordinate axes and is chosen from the 1/3 median segment of the longest side. In this way, rectangles, which are generated due this splitting, have the aspect ratio 3, so the packing constrain is applicable to them. Unfortunately, it is clear that this kind of tree can not guarantee $O(\log n)$ height. For this reason, Callahan [3] associated it with a special structure.

In contrast to the bounded aspect ratio trees, in balanced trees splitting operation is performed in such a way that the number of associated points of each generated child, is less than a fraction of its parent points. This operation creates a tree of height $O(\log(n))$. Height of search tree is an effective parameter in efficiency of search algorithms; because these algorithms typically need to traverse the tree from its root to leaves.

A kind of these trees is kd-tree, which was presented by Bently [5] to search in a file of records. Because of its simple structure, kd-tree is very common. The main difference between variations of this tree is the splitting method. There are a number of these methods.

Because of the simple structure of kd-tree, many researchers have considered it. The result of their research is various kinds of this structure. For example Yiallinois [10] introduced a structure named vp-tree, which uses a point called *vantage point* instead of orthogonal hyperplanes. This point is the center of a ball, which splits the space to two regions. This method is of a special interest in the field of data bases. An other example is sr-tree [7], whose nodes are determined with the intersection of a ball and a rectangle.

BAR-tree [4] is another tree based on the kd-tree with the relaxation of orthogonal splitting hyperplanes. In the other words, splitting hyperplanes can be non-orthogonal. In the special case these planes have the angle 45° with coordinate axes. Existence of non-orthogonal planes prevents the search algorithm to simply compute the distance of a point from a node.

BBD-tree [1] is the same as standard kd-tree with the midpoint splitting rule, which is associated with another rule. When the splitting operation generates a child node with more than 2/3 points of its parent, a rectangular region from the interior of this parent node will be separated to reduce the number of its points to a value less than this bound. This separated region is considered as a child node of the tree. Although with this additional rule, BBD-tree can satisfy both properties, but it is possible to produce non-convex nodes.

Although some other structures are introduced for spatial point partitioning, but almost all of these structures and respective search algorithms are serial. To the authors knowledge the only person who has done an out standing work in the field of parallel algorithms of this problem, is Callahan. Of course we have contacted David Mount[1] and Callahan[2] and we found that we were right.

Callahan constructs a fair split tree in parallel. Then based on this tree, he constructs a structure called *well-separated pair decomposition* in parallel. This construction process takes $O(d \log n)$ time using $O(n)$ processor in CREW PRAM model. Finally, with respect to these structures, he represents a parallel algorithm for all nearest neighbors problem that reaches the search time $O((3(s\sqrt{d} + 2\sqrt{d} + 1) + 2)^d \log n)$. He uses a method named rake-and-compress [2]. It is well known that given an arithmetic expression tree whose leaves are constants and whose internal nodes are labeled by $+$ and $\times$ operators obeying certain algebraic properties, one can evaluate all nodes of this tree in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM, where n is the size of the tree.

In this paper we present a totally different approach to the problem. At first we construct a BBD-tree of points, which is simpler than the structures of Callahan. Another advantage of our work is the simplicity of presented algorithm. This parallel algorithm, which resembles the serial version, is very suitable for practical applications. To precise this claim, one can compare the technical aspects of two methods. Callahan uses two structures each of them needs $O(dn)$ storage, in contrast, we uses one structure that needs $O(dn)$ storage. Both his algorithm and ours have an exponential dependency to the dimension d, but constant values in our dependency are less than that of Callahan.

4 FS-tree: Parallel Construction

In this section, we define a type of tree that, we will later show, permits to compute a split shrinking box in $O(\log n)$ time. We show how to construct such a tree in $O(n)$ time using $O(n)$ processor in CREW PRAM.

We define a split of P to be its partition into two nonempty point sets lying on either side of a hyperplane (called the splitting hyperplane) perpendicular to one of the coordinate axes, and not intersecting any points in P. We define a split tree of P to be a binary tree, constructed recursively as follows. If $|P| = 1$, its unique split tree consists of the node P. Otherwise a split tree is any tree with root P and two subtrees that are split trees of the subsets formed by a split of P. For any node A in the tree, we denote its parent (if it exists) by $p(A)$.

We define the *outer rectangle* of A, denoted $\widetilde{R}(A)$, for each node A top down as follows:

- For the root P , let $\widetilde{R}(P)$ be an open $d$cube centered at the center of $R(P)$, with $l(\widetilde{R}(P)) = l_{max}(P)$.
- For all other A, the splitting hyperplane used for the split of $p(A)$ divides $\widetilde{R}(p(A))$ into two open rectangles. Let $\widetilde{R}(A)$ be the one that contains A.

[1] `http://www.cs.umd.edu/~mount`
[2] `http://www.cs.jhu.edu/~callahan`

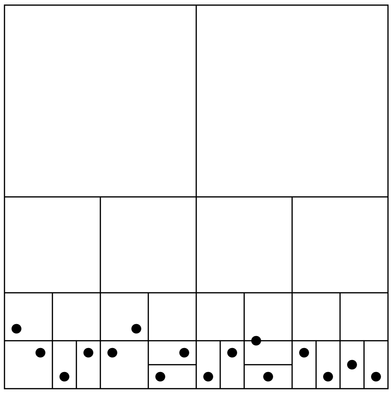

Figure 2: Space Partitioning in FS-tree

We define a fair split of A to be a split of A in which the splitting hyperplane is at equal distance from each of the two boundaries of $\widetilde{R}(A)$ parallel to it. A split tree formed using only fair splits is called a fair split tree, which we denoted by FS-tree through out this paper. In Figure 2 we have shown a sample FS-tree.

To parallelize the construction process of FS-tree, it is sufficient to do the splitting phase in parallel. We will show that in each level of the tree, this can be done in $O(d)$ by using $O(n)$ processors in CREW PRAM. Since the height of the tree can be as deep as $O(n)$, in the worst case, tree construction process takes $O(n)$ time.

Suppose that in a stage of the algorithm, we have given the set P and every processor is assigned to a point of this set. All points are stored in a global list L. For each dimension $i = 1, \ldots, d$ construct a doubly linked list, which is sorted according to the coordinates of that dimension. These lists, which we denote by L_i, have cross pointers to each other in a way that we could easily find the place of one point in other lists. Actual information about point coordinates are stored in the global list L and L_is only have pointers to the location of points in this list. With respect to the sorted lists L_i, we can find the coordinates of the enclosing rectangle of the point set in constant time.

Therefore at each stage we are given the lists L_i, which must split along the dimension s. Also triples (l_i, m_i, h_i) of each dimension represent the start,middle, and end point of that dimension. To split along the dimension s, with respect to (l_s, m_s, h_s) we know that m_s is the splitting point. For every dimension i create two empty lists low_i and $high_i$. Each processor p, which is assigned to a point q, check to see whether $q_s < m_s$? If $q_s < m_s$ then by tracking the pointers in L_i, updates its location in low_i. Otherwise, its new place will be in $high_i$. Afterward, for these $2d$ new lists, create triples (l, m, h). Notice that only for low_s and $high_s$ we must compute new values as follow:

$$\begin{cases} (l_{low}, m_{low}, h_{low}) = (l_s, \frac{l_s+m_s}{2}, m_s) \\ \\ (l_{high}, m_{high}, h_{high}) = (m_s, \frac{m_s+h_s}{2}, h_s) \end{cases}$$

After this step, previous lists L_i are deleted and this process continues recursively. It is obvious that each step can be done by $O(n)$ processor in $O(d)$ time. Therefore, in the worst case, the construction time is $O(dn)$.

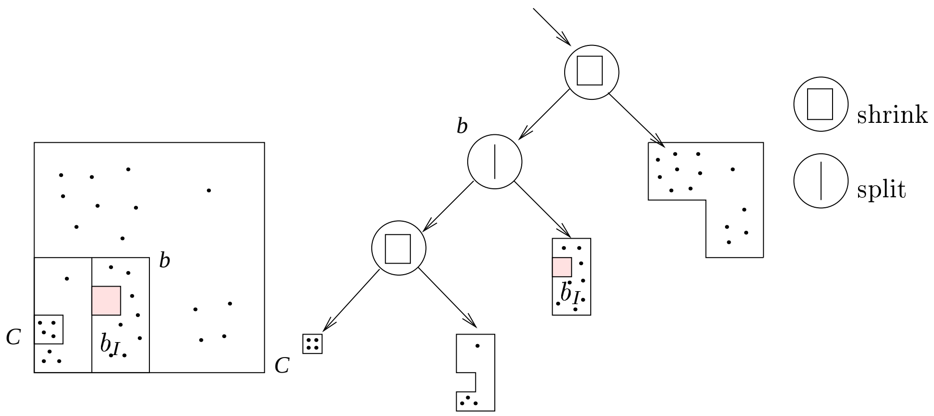

Figure 3: BBD-tree

5 BBD-tree: Parallel Construction

In this section we introduce the balanced box-decomposition tree or BBD-tree, which is the primary data structure used in our algorithm. It is among the general class of geometric data structures based on a hierarchical decomposition of space into d-dimensional rectangles whose sides are orthogonal to the coordinate axes. The main feature of the BBD-tree is that it combines in one data structure two important features that are present in these data structures.

An important feature of these data structures is that as one descends any path in these trees, the geometric size of the associated regions of space (defined, for example, to be the length of the longest side of the associated rectangle) decreases exponentially. The BBD-tree is based on a spatial decomposition that achieves both exponential cardinality and geometric size reduction as one descends the tree. The BBD-tree is similar to other balanced structures based on spatial decomposition into rectangles of bounded aspect ratio.

The principal difference between the BBD-tree and the other data structures is that each node of the BBD-tree is associated not simply with a d-dimensional rectangle, but generally with the set theoretic difference of two such rectangles, one enclosed within the other. Note, however, that any such region can always be decomposed into at most $2d$ rectangles by simple hyperplane cuts, but the resulting rectangles will not generally have bounded aspect ratio.

The BBD-tree is constructed through the repeated application of two operations, fair splits (or simply splits) and shrinks (See Figure 3). These operations will be described in detail later, but intuitively they represent two different ways of subdividing a cell into two smaller cells, called its children. A fair split partitions a cell by an axisorthogonal hyperplane. The two children are called the low child and high child, depending on whether the coordinates along the splitting coordinate are less than or greater than the coordinate of the splitting plane. A shrink partitions a cell into disjoint subcells, but uses a box (called the shrinking box) rather than a hyperplane to do the splitting. It partitions a cell into two children, one lying inside (the inner child) and one lying outside (the outer child) (Refer Figure 1). Both operations are performed so that the following invariants hold.

- All boxes satisfy the aspect ratio bound.
- If the parent has an inner box, then this box lies entirely within one of the two children. If the

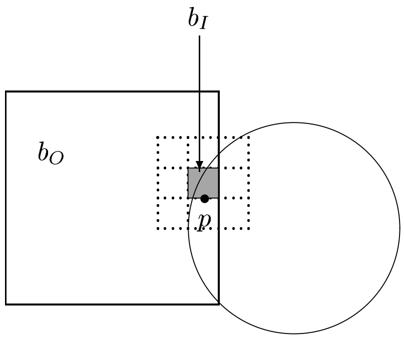

Figure 4: Packing Constraint in BBD-tree

operation is a shrink, then this inner box lies within the inner child of the shrink.

- Inner boxes are sticky for their enclosing outer boxes.

Observe that when only fair splits are performed, it may not generally be possible to guarantee that points are partitioned evenly. Hence a tree resulting from fair splits alone may not be balanced (The "fairness" of the split refers to the aspect ratio bound, not to the balance of the point partition).

The essential property of BBD-tree, which we use to prove our results is described in the following lemma, which we refer to as the *Packing Constraint* lemma (See Figure 4).

Lemma 1 . *[1] Given a BBDtree for a set of data points in* $\mathbb{R}^d$ *, the number of leaf cells of size at least* $s > 0$ *that intersect a Minkowski* L_m *ball of radius* r *is at most* $\lceil 1 + 6r/s \rceil^d$*.*

The BBDtree is constructed through a combination of fair split and shrink operations. Recall that we are given a set P of n data points in $\mathbb{R}^d$. Let C denote a hypercube containing all the points of P. The root of the BBDtree is a node whose associated cell is C and whose associated set is the entire set P. The recursive construction algorithm is given a cell and a subset of data points associated with this cell. Each stage of the algorithm determines how to subdivide the current cell, either through splitting or shrinking, and then partitions the points among the child nodes. This is repeated until the number of associated points is at most one, upon which the node is made a leaf of the tree.

Given a node with more than one data point, we first consider the question of whether we should apply splitting or shrinking. As mentioned before, splitting is preferred because of its simplicity, and the fact that after every d consecutive splits, the geometric size of the associated cell decreases by a constant factor. Splitting cannot guarantee that the point set is partitioned evenly, but we will see that shrinking can provide this. A simple strategy is that splits and shrinks are applied alternately. This will imply that both the geometric size and the number of points associated with each node will decrease exponentially as we descend a constant number of levels in the tree. A more practical approach is to perform splits exclusively, as long as the cardinalities of the associated data sets decrease by a constant factor after a constant number of splits. If this condition is violated, then a shrink is performed instead.

Once it has been determined whether to perform a split or a shrink, the splitting plane or shrinking box is computed, by a method to be described later. Once the splitting plane or shrinking box is known, we store this information in the current node, create and link the two children nodes into the tree, and

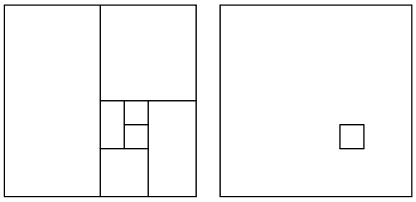

Figure 5: Split Shrinking Box

then partition the associated data points among these children. If any data points lie on the boundary of the splitting surface, then these points are distributed among the two children so that the final partition is as even as possible. Finally we recurs on the children.

Therefore the only step to be described is how to find the shrinking box in parallel. We suggest two methods for this purpose. The first method, which we call *Split Shrinking Box* needs to pre-build a FS-tree. The second method called *Corner Shrinking Box* needs a special process in each step. Now we are going to describe these two techniques in detail.

5.1 Split Shrinking Box

In this section we describe a method to find the shrinking box, which uses a pre-build FS-tree. Considering the technique introduced in [1] to compute these boxes, it is easily seen that midpoint splits and produced areas are always definite in the construction process (See Figure 5). There fore we can do these splits as a priori, which produces a FS-tree of point set. Once this tree has been constructed, to compute the shrinking box, it is sufficient to traverse this tree. This traversal go on until it reaches for the first time to a node whose cardinality is more than $1/3$ number of points associated with the current cell. Now we have two child nodes, each of them has at most $2/3$ of the points. It is obvious that the height of the constructed BBD-tree will be $\log_{\frac{3}{2}} n$.

You see that the problem of computing the splitting shrinking box has been reduced to the problem of traversing a binary tree, which has $O(n)$ nodes. A simple technique to traverse this tree in parallel, is via transformation to the *list ranking* problem. So this traversal can be done in $O(d \log n)$ by using $O(n)$ processor in CREW PRAM. To see details of this process refer to [9].

5.2 Corner Shrinking Box

Our method will depend on the following lemma:

Lemma 2 . *Let R be a ddimensional rectangle containing a set of n points. Let S be the set of rectangles that are similar to R and share a corner with it. Assume that no two points ever touch the boundary of a rectangle in S. Then there exists a rectangle $R' \in S$, such that $l_{max}(R') \leq \frac{1}{2} l_{max}(R)$. R' contains $\lfloor \frac{n}{2^d} \rfloor$ points in the set.*

Proof. Suppose we split R into 2^d similar rectangles by dividing it in half along each of its dimensions. Clearly, one of these rectangles contains at least $\lfloor \frac{n}{2^d} \rfloor$ points in the set. Choose such a rectangle and call it $\widetilde{R}$. Now, consider all the similar rectangles contained in $\widetilde{R}$ sharing the same corner of R as $\widetilde{R}$, and which have a point touching their boundary. Since no two points touch the boundary of the same rectangle, one of these rectangles contains exactly $\lfloor \frac{n}{2^d} \rfloor$ points inside (including the point on the boundary). Call this rectangle R'. Note that $l_{max}(R') \leq \frac{1}{2}\, l_{max}(R)$. □

The assumption that no two points touch the boundary of a rectangle is included for simplicity. It is easy to generalize the algorithm when this assumption is omitted. Our algorithm is as follows. We are given a set P containing n points, and its outer rectangle $\widetilde{R}(P)$, inherited from the preceding stage if P is not the root. Let R be a rectangle containing P and contained within $\widetilde{R}(P)$, that satisfies $l_{min}(R) \geq \frac{2}{3}\, l_{max}(R)$. Our splitting algorithm will guarantee that such an R always exists. We find a rectangle R' fulfilling the properties of Lemma 2.

To perform each step in constant time, we need some preprocessing. For every corner of R, we construct an array of the points sorted in the order determined by the similar rectangles of R that share that corner. There are a constant number of corners, so we can construct the sorted arrays in $O(\log n)$ time with $O(n)$ processors. From these arrays, we can easily find R' and the associated corner.

6 Parallel Search Algorithm

Before describing the main algorithm, we consider the case in which one processor finds the nearest neighbor of a point $q \in P$. It is obvious that this problem can be solved in $O(n)$ time with a simple *brute-force* algorithm. Many algorithms have been provided to reduce the constant factors in this algorithm. In this paper, our focus is on the methods, which first construct a suitable data structure in memory and then search in this structure. The algorithm we use in this paper has been represented in [5] for the first time. Brief description of the algorithm is as follow.

Algorithm 1 *[5] A recursive algorithm answers queries. In the terminal case, when search algorithm reaches a leaf, computes the distance between query point and the points associated with this leaf. Minimum of these values is saved as the nearest neighbor up to this time. When algorithm visits an internal node, at first it determines the side of the hyperplane in which the query point lies. Then algorithm will recurs on this child. Returning from this recursion, algorithm determines whether the other child of this node is closer to the query point than the nearest neighbor to this stage? If this is the case, this child also will be visited recursively. When the search returns to the root, it returns the nearest neighbor, which has been found to this time.*

An important observation in this algorithm is that for every query point, the algorithm visits each leaf whose distance to the query point is smaller than the nearest neighbor. In the other words, if we suppose that the distance between nearest neighbor and the query point is r, all the leaves overlap ball of radius r are visited by the search algorithm. With respect to the lemma of Packing Constraint 1, number of these leaves is independent of n.

Now we are ready to describe our parallel search algorithm for All Nearest Neighbors problem.

Algorithm 2 *There are n processor, each of them is responsible to find the nearest neighbor of one point in P. Therefore each processor runs the previous algorithm independent of other processors. Our model is CREW, so these processors can read required information simultaneously. Algorithm description is as follow:*

1. *An associated BBD-tree of point set P is constructed in parallel.*
2. *Every point q_i is assigned to a processor p_i.*
3. *Each processor executes Algorithm 1.*

Theorem 1 . *Search time of the Algorithm 2 is $O(\log n)$.*

Proof. Let r denote the distance from the query point to the last leaf cell that did not cause the algorithm to terminate. We know that all cells that have been encountered so far are within distance r from the query point. If p is the closest data point encountered so far, then because we did not terminate we have

$$r \leq dist(q, p)$$

We claim that no cell seen so far can be of size less than r/d. Suppose that such a cell was visited. This cell is within distance r of q, and hence overlaps a ball of radius r centered at q. The diameter of this cell in any Minkowski metric is at most d times its longest side length, and hence is less than r. Since the cell is associated with a data point lying within the outer box of the cell, the search must have encountered a data point at distance less than r from q. However, this contradicts the hypothesis that p is the closest point seen so far. Thus the number of cells visited up until termination is bounded by the number of cells of size at least r/d that can overlap a ball of radius r. From Lemma 1 we know that the number of such cells is a function of d. Using the bounds derived in Lemma 1, the number of cells is at most $\lceil 1 + 6d \rceil^d$. So we have established Theorem 1. □

7 Conclusion

In this paper we have represented a parallel algorithm to construct a BBD-tree associated with a set P containing n points in $\mathbb{R}^d$ in time $O(\log^2 n)$. Our algorithm uses $O(n)$ processor in CREW PRAM model. Then based on this data structure we represented a parallel search algorithm for All Nearest Neighbors problem in P. Search time of this algorithm in CREW PRAM model is proportional to $O(\log n)$.

Optimizing the tree construction phase is a valuable work to be done in this context. In the case we use simple splits to find the shrinking box, it is required to pre-build a FS-tree. It seems that it is possible to construct a similar data structure in time $O(\log n)$. In the other hand, by doing some preprocessing, it will be possible to do each step in corner splitting rule in time $O(1)$. Implementing the algorithm and reducing the number of processor to $O(n/\log n)$ or P in general, are interesting works that can be done in this context.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of ACM*, pages 891-923, 1998.

[2] P. B. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. 34th Sympos. on Foundations of Computer Science*, pages 332-341, 1993.

[3] P. B. Callahan and S. R. Kosaraju. A decomposition of multi dimensional point sets with applications to k-nearest-neighbors and n-body potentional fields. *Journal of ACM*, pages 67-90, 1995.

[4] C. Duncan, M. Goodrich and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of kd-trees and octrees. In *Proc. 10th ACM-SIAM Sympos. on Discrete Algorithms*, pages 300-309, 1999.

[5] J. H. Friedman, J. L. Bentley and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, pages 209-226, 1977

[6] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman and A. Wu. An efficient k-means clustering algorithm: Analysis and Implementation. In *Proc. 16th ACM Sympos. on Computational Geometry*, 2000.

[7] N. Katayama and S. Satoh. The SR-tree: An index structure for high dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 369-380, 1997.

[8] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proc. 29th STOC*, 1997.

[9] M. J. Quinn. *Parallel Computing*. second edition, Mc-GRAW HILL Inc., 1994.

[10] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Sympos. on Discrete Algorithms*, pages 311-321, 1993.