Separator Based Parallel Divide and Conquer in Computational Geometry

Alan M. Frieze*
Department of Mathematics
Carnegie Mellon University
Pittsburgh, PA 15213

Gary L. Miller†
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Shang-Hua Teng
Xerox Corporation
Palo Alto Research Center
Palo Alto, CA 94304

February 26, 1992

Abstract

An $O(\log n)$ time, n processor randomized algorithm for computing the k-nearest neighbor graph of n points in d dimensions, for fixed d and k is presented. The method is based on the use of sphere separators. Probability bounds are proved using the moment generating function technique.

1 Introduction

In a series of papers Miller, Teng, Thurston, and Vavasis developed methods for finding separators for graphs "nicely" embedded in d dimensions, [7, 8, 6, 10]. In this paper we make significant use of these tools in designing parallel algorithms for computational geometry.

In particular, we present an $O(\log n)$ time parallel algorithm using n processors for finding the k nearest neighbors for each of n points in d dimensional space and k fixed. Thus our algorithm uses no more work than the best sequential algorithm of Vaidya, [11]. Our algorithm uses randomization and assumes a unit time scan or prefix sum operation. Thus, given n points in d dimensions we construct the k-nearest neighbor graph, a "nicely" embedded graph in d dimensions.

Definition 1.1 *The k-**nearest neighborhood graph** of the points $P = \{p_1, \ldots, p_n\}$ in $\mathcal{R}^d$, is a graph with vertex set P, and edges*

$$E = \{(p_i, p_j) \mid p_i \text{ is a } k\text{-nearest neighbor of } p_j \quad or \quad p_j \text{ is a } k\text{-nearest neighbor of } p_i\}.$$

*Research supported in part by National Science Foundation Grant CCR-089-00112.

†Research supported in part by National Science Foundation Grant CCR-87-13489.

The best previous published result is due to Cole and Goodrich. They gave an $O(\log^{d-1} n)$ time and n processor algorithm, [4]. An $O(\log^2 n)$ time and n processor algorithm appears in Teng [10] and an announcement appeared in FOCS 1991, [6] without proof. This paper includes a proof of this result and our stronger result.

The Cole-Goodrich algorithm uses the multi-dimensional divide and conquer technique developed by Bentley [1]. Bentley divides the problem into two subproblems using hyperplanes. Cole and Goodrich give an improved 2-dimensional algorithm which is then used in Bentley's algorithm for any recursive calls to 2-dimensional problems. Bentley picks the hyperplane by translating a fixed hyperplane until the points are divided in half. The disadvantage of using a hyperplane for partitioning is that the number of edges from the k-nearest neighbor graph that cross the hyperplane may be as large as $\Omega(n)$. Thus combining the solution from the two subproblems may be quite expensive.

Our approach will be to use spheres instead of hyperplanes to do the partitioning. We will use the fact that the k-nearest neighbor graph has a small **sphere separator**, [6, 7, 8]. That is given a set of points P and its associated k-nearest neighbor graph G there exists a sphere S such that the number of points interior to S is approximately equal to the number exterior to S, and there is a $o(n)$ size subset of vertices W such that every edge crossing S has one end point in W.

At one level all we do is divide and conquer using sphere separators instead of cutting planes. However, there are several complications that one encounters when trying to get this high level idea to work. First of all, we are not given the graph–the goal is to find it. Thus, we do not know how to determine quickly if a given sphere produces a small separator in the graph. We can however quickly determine if it partitions the points properly. Secondly, the candidate spheres that we produce usually give good separators but not with enough reliability to make the obvious methods work. Finally, Bentley's technique of "gluing" the recursively computed solutions together by working in one less dimension does not seem to work since we are working with spheres. Even if we knew how to reduce the number dimensions with each recursive call we still do not see how to get an optimal work algorithm.

Thus our algorithm is a recursive divide and conquer where the separator at each level is chosen randomly from an appropriate distribution. Each separator partitions the problem into two subproblems, each of which is solved recursively. Upon returning from a recursive call one may find that the chosen sphere is not actually a good separator. It is now "too late" to restart and instead we will "punt" i.e., we will fall back on a slower algorithm. Surprisingly this only leads to a constant factor increase in the running time. This approach, "run-A-first-if-unlucky-then-run-B", is often used in expected case design. But to the best of our knowledge this is its first application to worst-case analysis.

Algorithms presented in this paper use a data-parallel machine model, the *parallel vector model* (Blelloch [2]). Parallel vector models can be efficiently mapped onto a broad variety of parallel architectures [2]). This model uses SCAN as a primitive operation. We choose it not only because we can derive tight bounds for our application of geometric separator theorems but also because adding the SCAN primitive makes theoretical models such as the PRAM closer to their practical counterparts such as the Connection Machine [2]. We will see that SCAN as a primitive greatly simplifies the description of the parallel algorithm. This is another illustration that SCAN

as primitive provides a better way to specify various degrees of parallelism at a high level. If we use more complicated constructions including random permuting, integer sorting, and selection, then all the algorithms presented in the paper can be implemented on a CRCW PRAM with only an extra $O(\log\log)$ factor increasing in time.

2 Sphere Separators

A d-dimensional *neighborhood system* $\mathcal{B} = \{B_1, \ldots, B_n\}$ is a finite collection of balls in $\mathcal{R}^d$. Let p_i be the center of B_i ($1 \leq i \leq n$) and call $P = \{p_1, \ldots, p_n\}$ the *centers* of $\mathcal{B}$. For any integer k, $\mathcal{B}$ is a *k-neighborhood system* if for all $1 \leq i \leq n$, the interior of B_i contains no more than k points from P. For each point $p \in \mathcal{R}^d$, let the *ply* of p, denoted by $\text{ply}_{\mathcal{B}}(p)$, be the number of balls from $\mathcal{B}$ that contains p. $\mathcal{B}$ is a *k-ply* neighborhood system if for all p, $\text{ply}_{\mathcal{B}}(p) \leq k$.

The following lemma relates the above two definitions. A proof of the lemma can be found in [6, 10]:

Lemma 2.1 (Density Lemma) *In $\mathcal{R}^d$, each k-neighborhood system $\mathcal{B} = \{B_1, \ldots, B_n\}$ is $\tau_d k$-ply, where τ_d is the* kissing number *in d dimensions, i.e., the maximum number of nonoverlapping unit balls in $\mathcal{R}^d$ that can be arranged so that they all touch a central unit ball [5].*

2.1 A geometric separator theorem

Suppose $\mathcal{B} = \{B_1, \ldots, B_n\}$ is a neighborhood system in $\mathcal{R}^d$. Each $(d-1)$-dimensional sphere S partitions $\mathcal{B}$ into three subsets (see Figure 1): $\mathcal{B}_I(S)$, $\mathcal{B}_E(S)$, $\mathcal{B}_O(S)$, those ball that are in the interior of S, in the exterior of S, and intersect S, respectively.

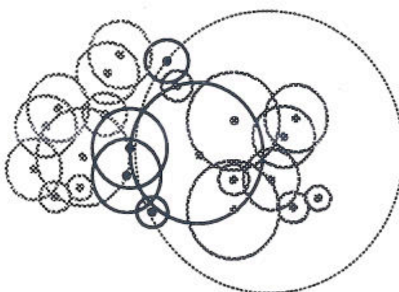

Figure 1: A sphere separator

The cardinality of $\mathcal{B}_O(S)$ is an important cost of S. This number is called the *intersection number* of S, denoted by $\iota_{\mathcal{B}}(S)$.

Notice that the removal of $\mathcal{B}_O(S)$ splits $\mathcal{B}$ into two subsets: $\mathcal{B}_I(S)$ and $\mathcal{B}_E(S)$, such that no ball in $\mathcal{B}_I(S)$ intersects any ball in $\mathcal{B}_E(S)$ and vice versa. In analogy to separators in graph theory, $\mathcal{B}_O(S)$ can be viewed as a separator of $\mathcal{B}$.

Definition 2.1 (Sphere Separators) *A $(d-1)$-sphere S is an $f(n)$-***separator** *that δ-***splits** *a neighborhood system $\mathcal{B}$ if $\iota_{\mathcal{B}}(S) \leq f(n)$ and $|\mathcal{B}_I(S)|$, $|\mathcal{B}_E(S)| \leq \delta n$, where f is a positive function and $0 < \delta < 1$.*

Miller, Teng, and Vavasis proved the following separator theorem, [6]:

Theorem 2.1 (Sphere Separator Theorem) *Each k-ply neighborhood system $\mathcal{B} = \{B_1, \ldots, B_n\}$ has a sphere separator S with intersection number $O(k^{\frac{1}{d}} n^{\frac{d-1}{d}})$ that $\frac{d+1}{d+2}$-splits $\mathcal{B}$.*

In this paper, we will use an algorithm of Miller-Teng-Thurston-Vavasis [6, 10]. The algorithm finds an $O(k^{\frac{1}{d}+\epsilon} n^{\frac{d-1}{d}+\epsilon})$-sphere separator of splitting ratio $(\frac{d+1}{d+2} + \epsilon)$ in random constant time, using n processors, with probability of success $1 - \frac{1}{n}$, where $\epsilon < \frac{1}{d+2}$ is a constant depending only on d. Further more, the algorithm only uses the centers of the neighborhood system. We will refer the algorithm as *Unit Time Separator Algorithm.*

3 Separator Based Divide and Conquer

At a high level, the separator based divide and conquer is very simple and intuitive. Given a neighborhood system $\mathcal{B}$ (explicitly or implicitly), it finds a sphere separator of a low intersection number and partitions $\mathcal{B}$ into two subsets of roughly equal size, those in the interior and exterior, respectively, and then recursively (sequentially or in parallel) solves the problem associated with the two sub-systems; the solutions for the two sub-systems are then combined to a solution to the whole problem.

To illustrate the main idea, we use the following query problem as our first example. The algorithm for this query problem will be used as a subroutine in the parallel algorithm for computing the nearest neighborhood graph. The algorithm has been presented in [10]. However, the proof of the parallel time complexity presented here is simpler and new.

3.1 Neighborhood query problem

The *neighborhood query problem* is defined as: given a k-ply neighborhood system $\mathcal{B} = \{B_1, \ldots. B_n\}$ in $\mathcal{R}^d$, preprocess the input to organize it into a search structure so that queries of the form "output all neighborhoods that contain a given point p" can be answered efficiently.

Like any other geometry query problem, there are three costs associated with the neighborhood query problem: the *query* time $Q(n,d)$ required to answer a query, the *space* $S(n,d)$ required to represent the search structure in memory, and the *preprocessing* time $T(n,d)$ required to build the search structure.

To our knowledge, there are no prior results on this problem. Nevertheless, as noted in [10], it is relatively straightforward to use multi-dimensional divide and conquer to build in $T(n.d) = O(n \log^{d-1} n)$ time a search structure with $Q(n.d) = O(k + \log^d n)$ and $S(n,d) = O(n \log^{d-1} n)$.

In contrast, the separator based divide and conquer constructs in $T(n.d) =$ random $O(n \log n)$ expected time a search structure with $Q(n,d) = O(k + \log n)$ and $S(n,d) = O(n)$.

By saying an algorithm runs in **random $t(n)$ time**, we mean that the algorithm never yields a wrong output but may terminate without producing any output. The probability of success – it produces a correct output in $t(n)$ steps – is at least $1 - \frac{1}{n^{\Omega(1)}}$. Moreover, our construction can be optimally parallelized. To simplify the discussion, we assume that the ply k is a constant. All algorithms can be easily generalized to handle a non-constant k.

3.2 A separator based search structure

Given a k-ply neighborhood system $\mathcal{B}$, we build a binary tree of height $O(\log n)$ to guide the search in answering a query. Associated with each leaf of the tree is a subset of balls in $\mathcal{B}$, and the search structure has the property that for all $p \in \mathcal{R}^d$, the set of balls covering p can be found in one of the leaves.

In the following construction, any sublinear separator with a constant splitting ratio can be used. The asymptotic complexity is the same. All that matters is that μ, δ are constants depending only on the dimension d with the properties that $0 < \mu < 1, 0 < \delta < 1$. If no further specified, we use $\mu = \frac{d-1}{d} + \epsilon$ and $\delta = \frac{d+1}{d+2} + \epsilon$ for a constant $0 < \epsilon < \frac{1}{d+2}$.

Let S be a sphere separator with intersection number n^μ that δ-splits centers of $\mathcal{B}$. Let $\mathcal{B}_0 = \mathcal{B}_I(S) \cup \mathcal{B}_O(S)$ and $\mathcal{B}_1 = \mathcal{B}_E(S) \cup \mathcal{B}_O(S)$ where $\mathcal{B}_E(S), \mathcal{B}_I(S), \mathcal{B}_O(S)$ are defined in Section 2.1. Clearly $|\mathcal{B}_0|, |\mathcal{B}_1| \leq \delta n + n^\mu$, and $|\mathcal{B}_1| + |\mathcal{B}_2| \leq n + n^\mu$. We store the information of S, its center and radius, in the root of the search tree, and recursively build binary search trees for $\mathcal{B}_0$ and $\mathcal{B}_1$, respectively. The roots of the tree for $\mathcal{B}_0$ and $\mathcal{B}_1$ are respectively the left and right children of the node associated with S. The recursive construction terminates when the subsets has cardinality smaller than m_0, for a parameter m_0 satisfying $m_0^\mu \leq \left(\frac{1-\delta}{2}\right) m_0$, to be specified later.

To answer a query when given a point $p \in \mathcal{R}^d$, we first check p against S, the sphere separator associated with the root of the search tree. There are three cases: (1) If p is in the interior of S then recursively search on the left subtree of S; (2) If p is in the exterior of S then recursively search the right subtree of S; (3) If p is on S then recursively search on the left subtree of S. When reaching a leaf, we then check p against all balls associated with the leaf and output all those that cover p.

The correctness of the search structure and the above searching procedure can be proved by induction: if p is in the interior (exterior) of S, then all balls that cover p must intersect either S or the interior (exterior) of S, and hence are in the left (right) subtree of S. The time complexity to answer a query is bounded by $O(h(n) + m_0)$, where $h(n)$ is the height of the search tree, given by the following recurrence.

$$h(m) \leq \begin{cases} 1 & \text{if } m \leq m_0 \\ h(\delta m + m^\mu) + 1 & \text{if } m \geq m_0. \end{cases}$$

The total space required is bounded by $m_0 s(n) + O(s(n))$, where $s(m)$ is a bound on the number of leaves in a search tree for m neighborhoods, given by the following recurrence.

$$s(m) \leq \begin{cases} 1 & \text{if } m \leq m_0 \\ s(\delta_1 m + m^\mu) + s((1-\delta_1)m) & \text{if } m > m_0, \end{cases}$$

where $\delta_1 \leq \delta$. It has been shown in [10] that

Lemma 3.1 *For a sufficient large constant m_0 depending only on d, δ, and μ,* (1) $h(n) = O(\log n)$ *and* (2) $s(n) = O(n/m_0)$.

Consequently, $Q(n, d) = O(\log n + m_0)$ and $S(n, d) = O(n)$. Using the random linear time sphere separator algorithm of Miller-Teng-Thurston-Vavasis, such a search structure can be computed in random $O(n \log n)$ time sequentially [10].

3.3 A parallel construction

The following parallel algorithm computes an $O(\log n)$ query time, $O(n)$ space search structure for the neighborhood query problem.

Parallel Neighborhood Querying

1. If $m \leq m_0$, output a tree of a single node storing all balls;
2. Otherwise,
   Iteratively apply Unit Time Sphere Separator Algorithm until finding a good sphere separator S;
3. Let $\mathcal{B}_0 = \mathcal{B}_I(S) \cup \mathcal{B}_O(S)$ and $\mathcal{B}_1 = \mathcal{B}_E(S) \cup \mathcal{B}_O(S)$;
4. Call Parallel Neighborhood Querying to recursively construct the search structure T_0 for $\mathcal{B}_0$ and T_1 for $\mathcal{B}_1$ in parallel;
5. Construct a tree with root storing S and left subtree T_1 and right subtree T_2;

Theorem 3.1 *With probability* $1 - \frac{1}{n^{\Omega(1)}}$, *Parallel Neighborhood Querying terminates in* $O(\log n)$ *time using n processors.*

Proof: Let $\mathcal{P} = v_1, \ldots, v_m$ be a path in the search tree constructed by Parallel Neighborhood Querying from a leaf v_1 to the root v_m. Associated with this path is a sequence of calls to Unit Time Sphere Separator Algorithm. The parallel time complexity of Parallel Neighborhood Querying is thus the length of the longest sequence so induced. We label a call 'head' if it returns a good sphere separator and 'tail', otherwise. The total number of heads along $\mathcal{P}$ is exactly m. Notice that each call at node v_i succeeds (turns 'head') with probability at least $1 - \frac{1}{2^i} \geq \frac{1}{2}$. Because calls are independent of each other, we can view the random sequence of calls as a Bernoulli trials with probability at least $\frac{1}{2}$ as 'head'. Therefore,

$$Pr\left(\sum_{i=1}^{m} x_i > 3m\right) \leq 2^{-2m}$$

Notice, $m = \log n$ in our setting, and the total number of such paths are n. Hence the algorithm terminates with probability $1 - \frac{1}{n^{\Omega(1)}}$. Clearly, it uses n processors. □

In comparison, the multi-dimensional divide and conquer takes $O(\log^d n)$ time to build its search structure.

4 The Punting Lemma for Probabilistic Divide and Conquer

In this section, we present a useful technique for devising and analyzing probabilistic parallel divide and conquer algorithms. This technique applies to the following scenario: Suppose we have two probabilistic algorithms A and B to do dividing and/or combining such that algorithm A is faster yet its probability of success is relatively low, while algorithm B is relatively slower but has higher probability of success. We will show that if B is at most $\log n$ factor slower than A, then the "run-A-first-if-unlucky-then-run-B" hybrid partition and/or combination procedure yields a parallel divide and conquer algorithm which is as fast as if only algorithm A is used, while with probability of success as high as if algorithm B is used.

To analyze the "run-A-first-if-unlucky-then-run-B" approach, we study the following class of probabilistic trees. Suppose a and b are a pair of integer functions. A *probabilistic (a,b)-tree* of size n (a power of 2) is a complete binary tree with n leaves whose internal nodes are weighted according to the following rule: a node v whose subtree contains m leaves has weight $a(m)$ with probability $1-\frac{1}{m}$ and $b(m)$ with probability $\frac{1}{m}$. The *weighted depth* of a leaf in T is the sum of weights of the nodes on the path from the leaf to the root of the tree.

Lemma 4.1 (Punting Lemma) *Let $RD(n)$ be the random variable which is equal to the largest weighted depth among leaves in a probabilistic $(0,\log m)$-tree T of size n. Let $\rho = \frac{\sqrt{e}}{2}$ and $A = e^{\rho/(1-\rho)}$. Then*

$$Pr(RD(n) > 2c\log n) \leq nAe^{-c\log n},$$

Proof: Let $v_1, \ldots, v_m$ be the path from a leaf v_1 to root v_m (where $m = \log n$). Let X_i be the weight of v_i. By the definition of the weight, $Pr(X_i = 0) = 1 - \frac{1}{2^i}$ and $Pr(X_i = i) = \frac{1}{2^i}$. For all $\lambda > 0$ and $t > 0$,

$$\begin{aligned}
Pr(X_1 + \cdots + X_m \geq t) &= Pr\left(e^{\lambda(X_1+\cdots+X_m - t)} \geq 1\right) \\
&\leq E\left(e^{\lambda(X_1+\cdots+X_m-t)}\right) && \left(Pr(X \geq a) \leq \frac{E(X)}{a}\right) \\
&= e^{-\lambda t}\prod_{i=1}^{m} E\left(e^{\lambda X_i}\right) \\
&= e^{-\lambda t}\prod_{i=1}^{m}\left[\left(1 - \frac{1}{2^i}\right) + \frac{1}{2^i}e^{\lambda i}\right] \\
&\leq e^{-\lambda t}\prod_{i=1}^{m}(1+\rho^i) && \left(\rho = \frac{e^{\lambda}}{2}\right) \\
&\leq e^{-\lambda t}\prod_{i=1}^{m} e^{\rho^i} && (1 + x \leq e^x,\ \forall x) \\
&\leq e^{-\lambda t + \rho + \rho^2 + \rho^3 + \cdots} \\
&= e^{-\lambda t + \frac{\rho}{1-\rho}} && (\text{assume } \rho < 1)
\end{aligned}$$

Therefore, let $\lambda = 1/2$, we have,

$$Pr(X_1 + \cdots + X_m \geq 2cm) \leq Ae^{-cm}.$$

Since there are n leaves, we have $Pr(RD(n) > 2c\log n) \leq nAe^{-c\log n}$. □

Consequently,

Corollary 4.1 *Suppose C is a constant positive integer. Let $RD(n)$ be the random variable which is equal to the largest weighted depth among leaves in a probabilistic $(C, \log m)$-tree T of size n. Then*

$$Pr(RD(n) > 2(c+C)\log n) \leq nAe^{-c\log n},$$

5 An $O(\log^2 n)$ Time all-Nearest-Neighborhood Algorithm

In this section, we present the random $O(\log^2 n)$ time n processor algorithm of [10] for computing the k-nearest neighborhood graph of a set of points. This algorithm will be used to motivate the $O(\log n)$ time algorithm of the next section. We give a simpler proof of the algorithm.

5.1 The Algorithm

Suppose $P = \{p_1, \ldots, p_n\}$ is a set of n points in $\mathcal{R}^d$. For each i, Let B_i be the largest ball centered at p_i whose interior contains at most $k-1$ points from P. B_i is called the *k-neighborhood ball of p_i in P* and $\{B_1, \ldots, B_n\}$ is called *the k-neighborhood system of P*. Given the the radius of each ball B_i it is not to hard to see how to construct the k-nearest neighbor graph in $O(\log n)$ time using n processors. Thus, for the rest of the paper we will restrict our attention to computing the k-neighborhood system for P.

We next reduce k-neighborhood system problem to the neighborhood query problem. The reduction is very simple and works as follows:

Simple Parallel Divide-and-Conquer

1. Choose any hyperplane dividing the points P into two equal size subsets P_l and P_r.
2. Recursively compute the k-neighborhood system $\mathcal{B}_l$ for P_l and $\mathcal{B}_r$ for P_r in parallel;
3. Let B_i be the ball associated with point p_i from the recursive construction. Assume $p_i \in P_l$ (P_r). There are two cases:
   - **Case 1**: If B_i does not intersect h, then B_i is the k-neighborhood ball of p_i in P;
   - **Case 2**: If B_i intersects h, then B_i could be larger than the k-neighborhood ball of p_i in P because there might be points from P_r (P_l) in the interior of B_i. In this case, correct the radius of B_i.

The key observation is that the correction step for Case 2 is just a neighborhood query problem which can be solved in random $O(\log n)$ time. It is interesting to point out that the correction step is where Bentley used the multi-dimensional reduction. He cannot simply stop the recursive call. Instead, he reduces the problem to one lower dimension. The separator based divide and conquer removes the necessity of a multi-dimensional recursive call and hence improves the algorithm.

Lemma 5.1 *Simple Parallel Divide-and-Conquer uses $O(n)$ processors and terminates in $O(\log^2 n)$ time with probability at least $1 - \frac{1}{n^{\Omega(\log n)}}$.*

Proof: Simple Parallel Divide-and-Conquer induces a complete binary tree T of height $O(\log n)$. Let $\mathcal{P} = v_1, \ldots, v_m$ be a path in T from a leaf v_1 to the root v_m. At v_i, let Γ_l be the set of balls of $\mathcal{B}_l$ that intersect h. Clearly Γ_l is a k-neighborhood system. So we can build a search structure for the neighborhood query problem of Γ_l. Using this search structure, we can decide in $O(\log n)$ time for each $p \in P_r$ (P_l) the set of balls in Γ_l (Γ_r) whose interior containing p and we correct the radius of B_i accordingly.

In the construction of search structure for Γ_l, we use Parallel Neighborhood Querying to build a search tree of height $O(i)$. Hence the path $\mathcal{P}$ induces $2^{1+2+\ldots+m}$ paths. Associated with each such path is a sequence of calls to Unit Time Sphere Separator Algorithm. The parallel time complexity of Simple Parallel Divide-and-Conquer is thus the length of the longest sequence so induced. Each call has probability of success (turning head) at least $\frac{1}{2}$. Each sequence has at most $c(1+2+\ldots+m)$ 'head's, for a constant $c > 1$ depending only on d, the dimension of the space. Let L be the random variable which is equal to the length of such a sequence. As a sequence of Bernoulli trials,

$$Pr(L > 3c(1+2+\ldots+m)) \leq 2^{-2c(1+2+\ldots+m)}$$

Since there are at most $n2^{1+2+\ldots+m}$ such paths and $m = \log n$, we have Simple Parallel Divide-and-Conquer terminates in $O(\log^2 n)$ time with probability at least $1 - \frac{1}{n^{\Omega(\log n)}}$. □

6 An $O(\log n)$ Time Nearest Neighborhood Algorithm

Recall that the parallel algorithm presented in Section 5 first partitions the points by a hyperplane, recursively solves the two subproblems, and then uses the parallel algorithm for the neighborhood query problem to make the final correction. The disadvantage of using a hyperplane for partitioning is that the number of balls intersecting the hyperplane may be as large as $\Omega(n)$. We thus cannot bound the number of balls we need to correct after the recursion. However, if we use a small cost sphere separator to do the first step partition, then we would expect to correct only a small number of balls. Hopefully, we could use the extra processors to do the correction step faster (with very high probability) – to obtain a random $O(\log n)$ time n processor algorithm.

The basic idea is as follows: we use sphere separator to do the partition. If the sphere separator at a node is good (with probability $1 - \frac{1}{m}$) and the sphere separators associated with the subtree induced by the node is well behaved (to be defined, also with probability $1 - \frac{1}{m}$), then we use a constant time linear processor parallel algorithm (to be described) to do the correction. However, if we are not lucky at the node, then we 'punt' and instead use Parallel Neighborhood Querying to do the correction. We will show that the probability of success of Parallel Neighborhood Querying is at least $1 - \frac{1}{n^2}$. Hence, by Punting Lemma 4.1, the new algorithm runs in random $O(\log n)$ time using n processors.

6.1 The Algorithm

We first make the following observation.

Lemma 6.1 *Suppose $P = \{p_1, \ldots, p_n\}$ is a set of n points in $\mathcal{R}^d$, and S is a $d-1$-sphere disjoint from P. If P_I and P_E are the sets of points of P in the interior or exterior of S, and $\mathcal{B}_I$, $\mathcal{B}_E$, and $\mathcal{B}$ be their k-neighborhood system of P_I, P_E, and P. respectively, then*

$$\iota_{\mathcal{B}_I}(S) + \iota_{\mathcal{B}_E}(S) = \iota_{\mathcal{B}}(S).$$

Proof: For each $B \in \mathcal{B}_I \cup \mathcal{B}_E$, suppose p_i is the center of B and B_i is k-neighborhood ball of p_i in P. If B does not intersect S, then clearly $B = B_i$. If B intersects S then B_i also intersects S. To see this, without loss of generality assume that $p_i \in P_I$. If B_i is completely in the interior of S, the B_i is equal to B, a contradiction, and therefore the lemma holds. □

Consequently, if we use a sphere S to partition the points, then we only need to correct $\iota_{\mathcal{B}}(S)$ balls. We now show how to perform the correction step in constant expected time with high probability.

For simplicity, in the following, we assume $k = 1$. The algorithm can be easily generalized to handle $k > 1$ and we will point out where to modify the algorithm.

Parallel Nearest Neighborhood

1. If $m \leq \log n$, deterministically compute the 1-neighborhood system in m time using m processors by testing all pairs of points.
2. Otherwise, repeated apply Unit Time Sphere Separator Algorithm until a sphere separator S that δ-splits P is found;
3. Let P_I and P_E be the sets of points of P in the interior and exterior, respectively, of S;
4. Recursively construct the 1-neighborhood systems $\mathcal{B}_I$ for P_I and $\mathcal{B}_E$ for P_E, in parallel;
5. Let T_I and T_E be the partition trees generated for P_I and P_E respectively;
6. Call Correction($\mathcal{B}_I$, $\mathcal{B}_E$, S, P, T) to correct all those balls that intersect S to obtain $\mathcal{B}$;

Correction($\mathcal{B}_I$, $\mathcal{B}_E$, S, P)

1. If $\iota_{\mathcal{B}_I}(S)+\iota_{\mathcal{B}_E}(S) \geq m^{\mu}$, then call Parallel Neighborhood Query to correct balls in $\mathcal{B}_I \cup \mathcal{B}_E$;
2. If $\iota_{\mathcal{B}_I}(S) + \iota_{\mathcal{B}_E}(S) \leq m^{\mu}$ then
   - Call Fast Correction($\mathcal{B}_I$, S, P_E.T);
   - Call Fast Correction($\mathcal{B}_E$, S, P_I.T).

We need only to describe the procedure for Fast Correction. throughout the discuss we assume the input is ($\mathcal{B}_I$,S,P_E,T).

Let Γ_I be the set of all balls of $\mathcal{B}_I$ that intersect S. Let T_r be the right subtree of T. By Lemma 6.1, we only need to correct balls in Γ_I. To do this, we need to compute, for each $B \in \Gamma_I$, the set of points of P_E that are in the interior of B. The basic idea is to "march" balls of Γ_I down the tree T_r: at each node, if a ball is contained in the interior (exterior) of the sphere separator, then it moves

down the left (right) subtree; If a ball intersects the sphere separator, then it is duplicated and one copy moves down the left subtree and another one moves down the right subtree. If the number of balls reaching a node is below a pre-defined constant, then the process stops in that subtree and all balls at the node become *in-active*.

Notice that there are $O(m)$ processors but there are only m^{μ} balls in Γ_I. We will show that the marching of the first $(1-\mu)\log m$ levels can be performed in constant time. Moreover, if the number of active balls at each level is sublinear in m, then the overall marching can be performed in constant time, because the height of T_r is $O(\log m)$. In subsection 6.4, we will prove the following lemma.

Lemma 6.2 *There is a constant $0 < \eta < 1$, such that with probability $(1 - \frac{1}{m})$, the number of active balls at all levels of T_r is at most $m^{1-\eta}$.*

When the number of active balls at some level is greater than $m^{1-\eta}$ (with probability $\leq \frac{1}{m}$) then we use Parallel Neighborhood Query procedure to do the correction.

6.2 Fast Correction

Suppose T is a complete binary tree of spheres and the height of T is h. Let B be a ball in $\mathcal{R}^d$. We define the following notion of *reachable* by B recursively:

1. The root r of T is always reachable;
2. If an internal node v of T is reachable and S_v is the sphere associated with v then
   - If B intersects S_v or its interior, then the left child of v is reachable;
   - If B intersects S_v or its exterior, then the right child of v is reachable;

Lemma 6.3 *The set of all reachable leaves in a divide and conquer tree T of height h can be computed in constant time, using $h2^h$ processors* [1].

Proof: For each internal node v, if B intersects S_v or its interior, then label $lc(v)$ 1 otherwise label $lc(v)$ 0; if B intersects S_v or its exterior, then label $rc(v)$ 1, otherwise label $rc(v)$ 0.

Clearly, the above labeling process can be computed in constant time using 2^h processors.

We make the following observation: a node v in T is reachable iff all nodes (including v) on the path from v to the root r of T are labeled with 1. The observation can be easily proved by induction on the level of the tree. Therefore, to recognize the set of all reachable leaves, it is sufficient to compute for each leaf whether all nodes on the path from it to the root of T are labeled with 1.

Notice that if we assign each leaf h processors, then we can assign one processor to each node on the path from the leaf to the root. Using the SCAN primitive, it can be decided in constant time,

[1] It is interesting to known whether the set of reachable leaves can be computed in constant time using 2^h processors.

whether all nodes on the path are labeled with 1. Since there are at most 2^h leaves in T, the set of reachable leaves of T can be computed in constant time, using $h2^h$ processors. $\square$

Now, suppose at each level of the tree T_r, there are at most $m^{1-\eta}$ active balls. Using the parallel algorithm of Lemma 6.3, we can compute for all $B \in \Gamma_I$, the set of leaves in T_r reachable by B, in constant time using m processors. By Lemma 2.1, for all $p \in P_E$, $\mathrm{ply}_{\Gamma_I}(p) \leq \tau_d$. Thus,

$$|\{(B,p) : B \in \Gamma_I, p \in P_E \cap B\}| \leq \tau_d m$$

Notice that $p \in B$ only if the leaf stores p is reachable by B. Therefore, we have just shown how, under the condition that there are at most $m^{1-\eta}$ balls at each level of T_r, to compute

$$\{(B,p) : B \in \Gamma_I, p \in P_E \cap B\}$$

in constant time using m processors.

To correct $B \in \Gamma_I$, it is sufficient to find the point in $\{p : p \in P_E \cap B\}$ that is closest to the center of B. This closest point can be computed in constant time with SCAN primitive using $|\{p : p \in P_E \cap B\}|$ processors.

> It worthwhile to point out that for $k > 1$, the computation of the closest point should be changed to the k closest points. The computation of the k closest points can be computed in random $O(\log\log k)$ time, with very high probability. Therefore for $k > 1$, the parallel time complexity has an $O(\log\log k)$ extra factor. It is an interesting question whether this extra factor can be eliminated.

The Fast Correction can now be specified as below:

Fast Correction(B_I, S, P_E,T)

1. Compute Γ_I;
2. Using the constant time algorithm in Lemma 6.3 to march balls of Γ_I down the tree T_r to compute the set of reachable leaves for each $B \in \Gamma_I$;
3. If the marching is successful, namely, there are at most $m^{1-\eta}$ balls at each level of T_r, correct balls in Γ_I as discussed above;
4. Otherwise, call Parallel Neighborhood Querying to do the correction.

6.3 The overall parallel complexity

Theorem 6.1 *The k-neighborhood system of a set of n points in $\mathcal{R}^d$ can be computed in random $O(\log n)$ time, using n processors.*

Proof: Let T be the partition tree induced by Parallel Nearest Neighborhood. For each node v in T, we assign a weight: $w(v) = 0$ if its sphere separator is 'good' and both of its Fast Corrections succeed. Otherwise, $w(v) = \log m_v$, where m_v is equal to the size of the subtree rooted at v. It follows from Lemma 6.2, with probability at least $1 - \frac{1}{m_v}$, $w(v) = 0$.

Let $\mathcal{P} = v_1, \ldots, v_m$ be a path in T from a leaf v_1 to the root v_m. Let $w(\mathcal{P}) = \sum_{i=1}^{m} w(v_i)$. By Punting Lemma 4.1,

$$Pr(w(\mathcal{P}) > 2c \log n) \leq \frac{1}{n^c}.$$

If $w(v_i) > 0$ we use Parallel Neighborhood Querying to build a search tree of height $O(i)$. Otherwise, we do nothing. Hence the number of paths that $\mathcal{P}$ induces is $O\left(2^{\sum_{i=1}^{m} w(v_i)}\right) \leq n^{2c}$ with probability at least $\frac{1}{n^c}$.

Associated with each such path is a sequence of calls to Unit Time Sphere Separator Algorithm. The parallel time complexity of Parallel Nearest Neighborhood is thus linear in the length of the longest sequence along all paths. Each sequence has at most $c_1(\sum_{i=1}^{m} w(v_i))$ 'head's, for a constant $c_1 > 1$ depending only on d. Let L be the random variable which is equal to the length of such a sequence. As a sequence of Bernoulli trials,

$$Pr\left(L > 3c_1(\sum_{i=1}^{m} w(v_i))\right) \leq 2^{-2c_1(\sum_{i=1}^{m} w(v_i))}$$

Since there are at most n^{2c} such paths and $m = O(\log n)$, we have Parallel Nearest Neighborhood terminates in $O(\log n)$ time with probability at least $1 - \frac{1}{n^{\Omega(1)}}$. □

6.4 The Probabilistic Analysis

The spheres generated by the Unit Time Sphere Separator Algorithm will be reused $O(\log n)$ time. Each sphere will first be used to construct a partition tree for the input points P. They will then be used $O(\log n)$ times, once for each level in the partition tree during the correction phase. In the correction phase these spheres will used for an application for which they where not intended. In particular, the sphere will not equally split the balls–unequal splits of the balls will be OK. But, we do need that with high probability each sphere intersects $o(n)$ balls. The next lemma follows easily from the methods in [6, 10].

Lemma 6.4 *Suppose $\frac{d-1}{d} < \alpha < 1$, $\beta = \alpha - \frac{d-1}{d}$, $\mathcal{B} \in \mathcal{R}^d$ is a k-ply neighborhood system, and P a set of points in $\mathcal{R}^d$ not necessarily related to $\mathcal{B}$. If S is a $(d-1)$-sphere generated by the Unit Time Sphere Separator on input P then,*

$$Prob\,(\iota_{\mathcal{B}}(S) > n^{\alpha}) \leq \frac{1}{n^{\beta}}.$$

Recall the process of marching Γ_I down the tree $\mathcal{T}_r$. Initially, the root r has $w = |\Gamma_I|$ balls. The total number of balls of the two children of r is equal to w plus the number of balls duplicated during the marching. By Lemma 6.4, with probability $1 - \frac{1}{w^{\beta}}$, the number of duplications is less than w^{α}.

We introduce the following process on a complete binary tree T of height K. The process constructs a subtree T' of T with the same root r, say, which is given a weight W. At a vertex $v \in T'$ of height k (in T) and weight w we proceed as follows:

- If $k = 0$ or $w \leq \bar{w}$ then v becomes a leaf of T' and is assigned a weight w. Here $\bar{w}$ depends only on d and α;
- Otherwise,
  where $\beta = \alpha - \frac{d-1}{d}$,
  - with probability $w^{-\beta}$, assign weight w to both the left child $lc(v)$ and right child $lc(v)$ of v, and recursively apply the random process on $lc(v)$ and $rc(v)$; otherwise
  - generate a number $w_0 \leq w$ (controlled by an adversary) and assign weight w_0 to $lc(v)$ and weight $w - w_0 + w^\alpha$ to $rc(v)$.

From Lemma 6.4, it follows that number of balls at a level of $\mathcal{T}_r$ is bounded above by the total weight at the same level in T if the same set of outcomes are used for the above two random processes.

Let $X(W, K)$ be a random variable which is equal to the total weight of the leaves of T'. It is not hard to prove that $E(X(W, K)) = O(WK)$. We now estimate the probability that $X(W, K)$ is large.

Lemma 6.5 *For all* $\frac{2d-1}{2d} < \alpha < 1$ *and* $\beta = \alpha - \frac{d-1}{d}$ *so that* $\alpha + \beta > 1$, *and for all* $\epsilon > 0$, *there exists a constant* $A > 0$ *such that*

$$Pr\left[X(W,K) \geq Ag(W)\log W\right] = O\left(\frac{1}{W^2}\right)$$

where $g(W) = W + 2^{(1-\alpha)k}(1+\epsilon)^k W^\alpha$.

Proof: For all $\lambda > 0$ and $t > 0$, we have

$$\begin{aligned} Pr\left(X(W,K) \geq t\right) &= Pr\left(e^{\lambda(X(W,K)-t)} \geq 1\right) \\ &\leq E\left(e^{\lambda(X(W,K)-t)}\right) \qquad \left(Pr(X \geq a) \leq \frac{E(X)}{a}\right) \\ &= e^{-\lambda t}E\left(e^{\lambda X(W,K)}\right) \end{aligned}$$

Let $\theta > 0$ be a small constant and let $\lambda = \frac{\theta}{g(W)}$.

We now prove the following inequality by induction :

$$E\left(e^{\lambda X(w,k)}\right) \leq e^{\lambda g(w)} \tag{1}$$

where $X(w, k)$ naturally refers to a generic vertex of weight w and height k in T.

Putting $A = 2/\theta$ and applying (1) we obtain the lemma.

Notice that when $k = 0$ or $w \leq \bar{w}$, then $X(w, k) = w$. So the base is correct. We thus assume $k \geq 1$ and $w \geq \bar{w}$.

By the definition of $X(w, k)$, we have

$$X(w,k) = \begin{cases} X(w - w_0 + w^\alpha, k-1) + X(w_0, k-1) & \text{with probability } 1 - \frac{1}{w^\beta} \\ 2X(w, k-1) & \text{with probability } \frac{1}{w^\beta} \end{cases}$$

Therefore, using the independence of the process in the left and right subtrees of a vertex, and assuming that $\bar{w}$ is sufficiently large,

$$
\begin{aligned}
& E\left(e^{\lambda X(w,k)}\right) \\
= & \left(1-\frac{1}{w^\beta}\right) E\left(e^{\lambda X(w-w_0+w^\alpha,k-1)}\right) \cdot E\left(e^{\lambda X(w_0,k-1)}\right) + \frac{1}{w^\beta} E\left(e^{\lambda X(w,k-1)}\right)^2 \\
\leq & \left(1-\frac{1}{w^\beta}\right) e^{\lambda\left(w_0+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}w_0^\alpha+(w-w_0+w^\alpha)+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}(w-w_0+w^\alpha)^\alpha\right)} \\
& + \frac{1}{w^\beta} e^{2\lambda\left(w+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}w^\alpha\right)} \qquad \text{(By Induction)} \\
\leq & \left(1-\frac{1}{w^\beta}\right) e^{\lambda\left((w+w^\alpha)+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}[w_0^\alpha+(w-w_0+w^\alpha)^\alpha]\right)} + \frac{1}{w^\beta} e^{2\lambda\left(w+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}w^\alpha\right)} \\
\leq & \left(1-\frac{1}{w^\beta}\right) e^{\lambda\left((w+w^\alpha)+2^{(1-\alpha)k}(1+\epsilon)^{k-1}[w+w^\alpha]^\alpha\right)} + \frac{1}{w^\beta} e^{2\lambda\left(w+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}w^\alpha\right)} \\
& \qquad [\, w_0^\alpha + (w-w_0+w^\alpha)^\alpha \leq 2^{1-\alpha}(w+w^\alpha)^\alpha] \\
\leq & \left(1-\frac{1}{w^\beta}\right) e^{\lambda\left(w+w^\alpha+2^{(1-\alpha)k}(1+\epsilon)^{k-1}\left[1+\frac{2\alpha}{w^{1-\alpha}}\right]w^\alpha\right)} + \frac{1}{w^\beta} e^{2\lambda\left(w+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}w^\alpha\right)} \\
\leq & \left(1-\frac{1}{w^\beta}\right) e^{\lambda\left(w+2^{(1-\alpha)k}(1+\epsilon)^{k-1}(1+\frac{\epsilon}{2})w^\alpha\right)} + \frac{1}{w^\beta} e^{2\lambda\left(w+2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}w^\alpha\right)}
\end{aligned}
$$

To prove (1) it is sufficient to prove that the right hand side of the last inequality divided by $e^{\lambda g(w)}$ is at most 1. Denote this ratio by RHS. Let

$$
\begin{aligned}
X &= \lambda 2^{(1-\alpha)k-1}\epsilon(1+\epsilon)^{k-1}w^\alpha \\
Y &= \lambda w 2^{(1-\alpha)(k-1)+2}(1+\epsilon)^{k-1}
\end{aligned}
$$

By a simple calculation, we have

$$RHS \leq \left(1-\frac{1}{w^\beta}\right)e^{-X} + \frac{1}{w^\beta}e^{Y}$$

Recall that $\lambda = \frac{\theta}{g(W)}$ and so both X and Y are small. Therefore, for a small constant $\delta > 0$, the following inequalities hold.

$$
\begin{aligned}
e^{-X} &\leq 1-(1-\delta)X \\
e^{Y} &\leq 1+(1+\delta)Y
\end{aligned}
$$

Using the above inequalities we have

$$
\begin{aligned}
RHS &\leq \left(1-\frac{1}{w^\beta}\right)[1-(1-\delta)X] + \frac{1}{w^\beta}[1+(1+\delta)Y] \\
&\leq 1-\left[\lambda w^\alpha 2^{(1-\alpha)(k-1)}(1+\epsilon)^{k-1}\left((1-\delta)(1-\frac{1}{w^\beta})\epsilon 2^{-\alpha} - \frac{4(1+\delta)}{w^{\beta+\alpha-1}}\right)\right] \\
&\leq 1,
\end{aligned}
$$

for $\bar{w}$ sufficiently large. Thus (1) is proven and the lemma follows. □

Choosing α very close to 1 and ϵ very close to 0, we obtain Lemma 6.2.

References

[1] J. L. Bentley. Multidimensional divide-and-conquer. *CACM*, 23:214–229, 1980.

[2] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT-Press, Cambridge MA, 1990.

[3] K. Clarkson. Fast algorithm for the all-nearest-neighbors problem. In *24th Annual Symposium on Foundations of Computer Science*, pages 226–232. IEEE, 1983.

[4] Richard Cole and Michael T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. Depart. of Computer Science 88-14, Johns Hopkins University, 1988.

[5] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, 1988.

[6] G. L. Miller, S.-H. Teng, and S. A. Vavasis. An unified geometric approach to graph separators. In *31nd Annual Symposium on Foundations of Computer Science*, pages 538–547. IEEE, 1991.

[7] Gary L. Miller and William Thurston. Separators in two and three dimensions. In *Proceedings of the 22th Annual ACM Symposium on Theory of Computing*, pages 300–309, Baltimore, May 1990. ACM.

[8] Gary L. Miller and Stephen A. Vavasis. Density graphs and separators. In *SODA*. ACM, 1991.

[9] J. H. Reif and S. Sen. Polling: a new randomized sampling technique for computational geometry. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 394–404. ACM, 1989.

[10] S.-H. Teng. *Points, Spheres, and Separators: a unified geometric approach to graph partitioning*. PhD thesis, Carnegie-Mellon University, School of Computer Science, 1991. CMU-CS-91-184.

[11] P. M. Vaidya. An $o(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4:101–115, 1989.

[12] F. F. Yao. A 3-space partition and its application. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 258–263. ACM, 1983.