A Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields

PAUL B. CALLAHAN AND S. RAO KOSARAJU

Johns Hopkins University, Baltimore, Maryland

Abstract. We define the notion of a *well-separated pair decomposition* of points in d-dimensional space. We then develop efficient sequential and parallel algorithms for computing such a decomposition. We apply the resulting decomposition to the efficient computation of k-nearest neighbors and n-body potential fields.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*: F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: All nearest neighbors, fast multipole method

1. *Introduction*

We define the notion of a *well-separated pair decomposition* of a set P of n points in d dimensions. This consists of a binary tree whose leaves are points in P, with internal nodes corresponding to subsets of P in the natural way, and a list of pairs of nodes, such that the sets corresponding to each node are geometrically separated in a manner to be defined, and each distinct pair of points is "covered" by exactly one of the pairs of nodes.

We show that although there are $\binom{n}{2}$ pairs of points, we can always find a well-separated pair decomposition using $O(n)$ pairs of nodes. Additionally, we show that such a decomposition can be computed in $O(n \log n)$ sequential time, which we prove is optimal, and in $O(\log^2 n)$ time on a CREW PRAM using $O(n)$ processors.

Using this decomposition, we show that the k-nearest neighbors of each point can be computed in $O(n)$ sequential time, and in $O(\log n)$ parallel time on a CREW PRAM with $O(n)$ processors, for any fixed k. Note that this gives

$O(\log^2 n)$ parallel time for the whole problem, including the construction of the decomposition. This is the first parallel algorithm for this problem that runs in deterministic $O(\log^c n)$ parallel time with $O(n)$ processors for a c that is not a function of d.[1] The composed sequential algorithm has a strong similarity to that of Vaidya [1986; 1989].

We also apply this decomposition to the efficient computation of n-body potential fields. Greengard [1988], Greengard and Rokhlin [1987], and Rokhlin [1985] have developed a very interesting $O(n)$ time sequential algorithm known as the *Fast Multipole Method* that, given n charges and their locations, calculates, for each of the n locations, the potential field due to remaining $n - 1$ charges, up to a fixed precision. However, the algorithm in Greengard [1988] makes assumptions about the geometric distribution of charges, which might be justifiable on pragmatic grounds, but are not normally made when treating problems in computational geometry. Also, when the desired output precision is much less than the input precision, the justification becomes weaker. Without these assumptions, we cannot even see an obvious variant of the Fast Multipole Method that runs in $O(n \text{ polylog}(n))$ steps. We show that once the well-separated pair decomposition is computed, we can adapt the Fast Multipole Method so that the remaining computation takes $O(n)$ sequential time and $O(\log n)$ parallel time on an EREW PRAM with $O(n/\log n)$ processors.

In Sections 3 and 4, we develop sequential algorithms for computing the tree and the pairs, respectively. The corresponding parallel algorithms are developed in Sections 6 and 7. In Section 5, we establish a lower bound on the depth of the tree when the number of pairs is linear. In Section 8, we develop sequential and parallel algorithms for computing k-nearest-neighbors. Finally, in Section 9, we develop sequential and parallel algorithms for computing n-body potential fields.

2. Definitions

Let P be a set of points in $\mathfrak{R}^d$, where d is a constant denoting the dimension. 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 cartesian product $R = [x_1, x_1'] \times [x_2, x_2'] \times \cdots \times [x_d, x_d']$ in $\mathfrak{R}^d$. 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 *d-cube*, and denote 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 d-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 d-ball.

We say that point sets A and B are *well-separated* if $R(A)$ and $R(B)$ can each be contained in d-balls of radius r whose minimum distance is at least sr, where s is the *separation*, assumed to be fixed throughout our discussion at a value strictly greater than 0. Although we can define this notion in a somewhat

---

[1] See, for example, Bentley [1980], Cole and Goodrich [1988; 1992], Frieze et al. [1992], and Miller et al. [1970].

more natural way without using bounding rectangles, this definition will make subsequent computation easier.

We will often assume that P has a binary tree T associated with it, where each leaf of T is labeled by a singleton set containing one of the points in P, and each internal node is labeled by the union of all sets labeling the leaves of its subtree. We will refer to each node in T by the name of the set labeling it. Since leaves are labeled by singleton sets, we may also refer to them by the names of points.

We define the *interaction product*, denoted $\otimes$, between any two point sets A and B as follows:

$$A \otimes B = \{\{p, p'\} \mid p \in A, p' \in B, \text{and } p \neq p'\}.$$

Note that $P \otimes P$ is the set of all distinct pairs of points in P.

The set $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ is said to be a *realization* of $A \otimes B$ if

(i) $A_i \subseteq A$ and $B_i \subseteq B$ for all $i = 1, \ldots, k$.

(ii) $A_i \cap B_i = \varnothing$ for all $i = 1, \ldots, k$.

(iii) $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \varnothing$ for all i, j such that $1 \le i < j \le k$.

(iv) $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$.

The realization is said to be *well-separated* if it satisfies the additional property

(v) A_i and B_i are well-separated for all $i = 1, \ldots, k$.

Let T be a binary tree associated with P. For $A, B \subseteq P$, we say that a realization of $A \otimes B$ *uses* T if all A_i and B_i in the realization are nodes in T. We define a *well-separated pair decomposition* of P to be structure consisting of a binary tree T associated with P, and a well-separated realization of $P \otimes P$ that uses T.

Agarwal et al. [1991; 1992] have also used the notion of partitioning the set of pairs of a point set into pairs of sets satisfying certain geometric criteria. In their formulation, the sets must lie in cones, each of some small angle α, whose axes are parallel and which intersect only at the apex. This decomposition does not guarantee that each pair in $P \otimes P$ is *uniquely* represented by a pair in the decomposition, and does not incorporate the notion of separation of clusters based on distance. Both of these properties are essential for the application to the Fast Multipole Method. This is because the influence of each charge must be taken into account exactly once in computing the potential at some point, and the convergence of a truncated multipole expansion depends on distance-based separation. These properties appear less critical with respect to purely geometric problems, but they may prove useful in future applications.

3. *Computing a Fair Split Tree of a Point Set*

Given a point set P, we would like to construct a binary tree T, such that there exists a linear size well-separated realization of $P \otimes P$ that uses T. In this section, we define a type of tree that, we will later show, permits an $O(n)$ size well-separated realization. We show how to construct such a tree in $O(n \log n)$ time sequentially.

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 $\hat{R}(A)$, for each node A top down as follows:

—For the root P, let $\hat{R}(P)$ be an open d-cube centered at the center of $R(P)$, with $l(\hat{R}(P)) = l_{\max}(P)$.

—For all other A, the splitting hyperplane used for the split of $p(A)$ divides $\hat{R}(p(A))$ into two open rectangles. Let $\hat{R}(A)$ be the one that contains A.

We define a *fair split* of A to be split of A in which the splitting hyperplane is at a distance of at least $l_{\max}(A)/3$ from each of the two boundaries of $\hat{R}(A)$ parallel to it. A split formed using only fair splits is called a *fair split tree*.

We present an algorithm to find a fair split tree for any point set. This algorithm will run in time $O(n \log n)$, which is optimal, as we will later prove.

We first present a high-level description of the algorithm, and then show how to implement it efficiently. Let P be the given point set. If $|P| = 1$, return a tree consisting of a single node labeled P. Otherwise, consider a hyperplane perpendicular to the longest dimension of $R(P)$ that divides it into two geometrically equal halves.[2] Clearly, this defines a fair split of P into two nonempty sets P_1 and P_2. Find fair split trees for these, and return a tree with root P, having two subtrees that are the split trees of P_1 and P_2.

Note that the splits defined above are at a distance $l_{\max}(A)/2$ from the nearest parallel boundaries of $R(A)$, and therefore satisfy a stronger condition than we require. The relaxed condition will become important when we present our parallel algorithm, so we will use it in all our analysis. However, the constants in the size of the realization can be improved by assuming the stronger condition.

Unfortunately, each split can result in all but one point going to one side, so a naive implementation could result in quadratic complexity. However, in linear time we can construct a partial fair split tree in which each leaf corresponds to a point set of size no greater than $n/2$. The splitting phase is as follows:

Assume that we have presorted the points by each of the coordinates. For each coordinate, we maintain a doubly linked list of points in increasing order, with cross-references so that, given any point, we may easily retrieve its place in each of the lists. We will be modifying these lists in the course of our splitting, so we make copies for later reference.

Note that a fair split divides one of the sorted lists into two contiguous pieces. We determine the split location in this list using a linear search that alternates between both ends of the list, heading towards the middle. The search time will be proportional to the size of the smaller piece.

We now delete all of the points in the smaller piece from all of the lists. Since we will need to consider these points later, we initialize empty linked-lists

---

[2] We make the nondegeneracy assumption that no points lie on this hyperplane. It is easy to adapt the algorithm to eliminate the need for this assumption.

for each coordinate; these will eventually contain all of the deleted points in sorted order. As we delete each point, we mark it with a pointer back to each list, but we do not insert it yet. This phase also requires time proportional to the size of the smaller piece.

We repeat the above splitting process on the remaining collection of lists until the resulting set of points is no more than half the size of the original. As we do this, we can construct a partial split tree consisting of a linear chain starting with the whole set and ending with the set of points left in the final step. Each node in this chain has an additional subtree consisting of the split tree (to be computed later) of the set of points deleted at that step. Note that the amount of work is proportional to the number of points deleted, and hence is linear.

The set of points deleted at a step is never more than half the size of the original set. Hence, if we can construct sorted lists for all of these point sets in linear time, we can continue the process recursively and obtain $O(n \log n)$ time by a divide-and-conquer approach. To construct all the lists for all the deleted points sorted by some coordinate, we return to the original list of points sorted by that coordinate (which we copied prior to the deletions), and traverse it in increasing order. Each time we find a point, we append it to the end of the appropriate list, given by the pointer that was set at the time of deletion. There are a constant number of coordinates, so this phase takes linear time.

The remainder of the algorithm is a straightforward application of divide-and-conquer (but with more than a constant number of subproblems). The time complexity, including presorting, is clearly $O(n \log n)$. This gives us the following theorem:

THEOREM 3.1. *Given a set P of n points in $\Re^d$, a fair split tree of P can be constructed in $O(n \log n)$ time.*

Note that as we generate each node A in the split tree T we can compute $R(A)$ in $O(d)$ steps, using the sorted lists we maintain. This property will hold for the parallel algorithm as well, so we assume that any solution of this problem includes a copy of $R(A)$ at each node A in T. For this reason, we will assume in subsequent sections that in $O(1)$ time, we can construct the smallest d-ball containing $R(A)$ for any A in T. This will allow us to test whether two nodes are well separated in constant time.

4. *Computing Well-Separated Pairs for a Point Set*

Suppose we have already computed a fair split tree T for the point set $P = \{p_1, \ldots, p_n\}$. We present an algorithm to compute a well-separated realization of $P \otimes P$, denoted by $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$, that uses T. We will show that the resulting realization has linear size. Note that $\binom{n}{2}$ is a trivial upper bound on k, since $P \otimes P$ has the well-separated realization $\{\{\{p_i\}, \{p_j\}\} | 1 \le i < j \le n\}$.

For the analysis of the algorithm, we require the following geometric result. We say that two rectangles of equal dimension *overlap* if the intersection of the corresponding open rectangles is nonempty. Let T be a fair split tree associated with P. Let C be a d-cube, and let $S = \{A_1, A_2, \ldots, A_l\}$ be a set of nodes in T such that $A_i \cap A_j = \varnothing$ for all $i \ne j$, and $l_{\max}(p(A_i)) \ge l(C)/c$ and

$R(A_i)$ overlaps C, for all $i = 1, \ldots, l$. Define $K(c, d)$ to be the maximum possible value for $|S|$, independent of P and T.

LEMMA 4.1. $K(c, d) \le (3c + 2)^d$.

PROOF.   For any node A in T, the following inequality holds:

$$l_{\min}(\hat{R}(A)) \ge \frac{l_{\max}(p(A))}{3}. \tag{1}$$

By the fair split condition, this is true when $p(A) = P$. Now, for $p(A) \ne P$, consider the fair split of $p(A)$ that formed A. If $l_{\min}(\hat{R}(A)) = l_{\min}(\hat{R}(p(A)))$ then the inequality is preserved, because $l_{\max}(p(A)) \le l_{\max}(p(p(A)))$. Otherwise, $l_{\min}(\hat{R}(A)) < l_{\min}(\hat{R}(p(A)))$. In this case, $\hat{R}(p(A))$ must have a unique minimum-length dimension, because only one dimension can be split at a time. In other words, there is a unique coordinate axis i such that $l_i(\hat{R}(p(A))) = l_{\min}(\hat{R}(p(A)))$. This axis must have been perpendicular to the split, so the fair split condition guarantees that $l_i(\hat{R}(A)) \ge l_{\max}(p(A))/3$.

Now, consider a d-cube C. Suppose we have a set of disjoint d-cubes of length l' that all overlap C. We can bound the size of this set by a packing argument. It is not hard to see that $(l(C)/l' + 2)^d$ is an upper bound, since the cubes in this set must all be *contained* in a d-cube of length $l(C) + 2l'$, centered at the center of C.

Note that the outer rectangles of disjoint A are disjoint. By (1), each $\hat{R}(A)$ overlapping C contains a d-cube of length $l_{\max}(p(A))/3$ that also overlaps C. Hence, our packing result applies to the number of $\hat{R}(A)$ that overlap C. This is at least as large as the number of $R(A)$ that overlap C. We assumed that $l_{\max}(p(A)) \ge l(C)/c$, so our bound is $(3c + 2)^d$, as stated.   □

LEMMA 4.2.   *Let* $P = \{p_1, \ldots, p_n\}$ *be a point set with an associated fair split tree* T. *Then* $P \otimes P$ *has a well-separated realization of size* $O(n)$ *that uses* T.

PROOF.   We present a simple algorithm for constructing such a realization, given T. Since our algorithm is recursive, we call the initial point set P_0 to distinguish it from those treated in recursive calls.

First, note that every internal node of T has exactly two children, which we call A and B. For each node, construct a well-separated realization $\{\{A_1, B_1\}, \ldots, \{A_l, B_l\}\}$ of $A \otimes B$. We can easily prove that the union of all such realizations is a well-separated realization of $P_0 \otimes P_0$. We use the following procedure to construct the realizations in a way that ensures the size bound.

Let P and P' be point sets (initially $P = A$ and $P' = B$). If P and P' are well separated, we return $\{\{P, P'\}\}$. Otherwise, we swap P and P' if necessary to ensure that $l_{\max}(P) \ge l_{\max}(P')$. It must be the case that $|P| > 1$; if not, $l_{\max}(P) = l_{\max}(P') = 0$, and P and P' are well separated. Hence, node P has two children, which we call P_1 and P_2. We observe that the subtrees rooted at these nodes are fair split trees of P_1 and P_2 and we associate them with the sets. We recursively construct well-separated realizations of $P_1 \otimes P'$, and $P_2 \otimes P'$, and return the union as our result.

It is easy to see that this procedure terminates and gives a well-separated realization of $A \otimes B$. We can unravel the above procedure into a computation tree with $\{A, B\}$ as the root. We establish necessary geometric criteria for a

pair $\{P, P'\}$ to be an internal node in a computation tree. By using Lemma 4.1 to count the number of nodes satisfying this criteria, we obtain an $O(n)$ bound on the number of leaves summed over all computation trees. This bounds the number of pairs output by the procedure.

Let $d(P, P')$ denote the minimum distance between $R(P)$ and $R(P')$. Note that $R(P)$ and $R(P')$ can each be contained in d-balls of radius $(\sqrt{d}/2)l_{\max}(P)$. The minimum distance between these d-balls is at least $d(P, P') - (\sqrt{d})l_{\max}(P)$, since $d(P, P')$ is at most the distance between their centers. Hence, P and P' are well separated as long as $d(P, P') - (\sqrt{d})l_{\max}(P) \geq s(\sqrt{d}/2)l_{\max}(P)$, where s is the separation. Equivalently, we may write this condition as:

$$d(P, P') \geq \left(s(\sqrt{d}/2) + \sqrt{d}\right)l_{\max}(P).   \tag{2}$$

We also have the following inequality:

$$l_{\max}(p(P')) \geq l_{\max}(P).   \tag{3}$$

First note that this is obvious in the special case where $P = A$ and $P' = B$, since they are both children of $p(P')$. Otherwise, consider the point in our procedure at which we split $p(P')$. It must have been true that $l_{\max}(p(P')) \geq l_{\max}(P'')$, where P'' denotes the point set paired with $p(P')$ at this point. Clearly, $l_{\max}(P'') \geq l_{\max}(P)$, so (3) follows by transitivity.

Now, consider the pairs $\{P, P'\}$ of point sets that appear in a call to our procedure and result in further recursive calls. Let S_P be the set of all P' paired with P in some procedure call in which P is the set to be split. By (2), all $P' \in S_P$ must overlap a d-cube centered at the center of $R(P)$ and with length $(s\sqrt{d} + 2\sqrt{d} + 1)l_{\max}(P)$. Moreover, all $P' \in S_P$ are disjoint. To see this, suppose otherwise. Then some $P' \in S_P$ would have an ancestor $P'' \in S_P$. Our algorithm insures that if P'' is paired with P, then no descendant of it will be, because P will be split for the next recursive call. Combining these facts with (3), we see that $|S_P|$ is bounded by $K(s\sqrt{d} + 2\sqrt{d} + 1, d)$, which by Lemma 4.1 is no more than $(3(s\sqrt{d} + 2\sqrt{d} + 1) + 2)^d$.

Now, consider a pair $\{A_i, B_i\}$ that is a leaf in one of our computation trees. This was produced by an invocation of our procedure on an internal node in the computation tree. We first consider the case in which the pair came from a trivial invocation in which A and B were themselves well separated. Clearly, there are only $n - 1$ such pairs.

Otherwise, $\{A_i, B_i\}$ has some parent in the computation tree of the procedure in which the point set pair is $\{P, P'\}$. We have bounded the number of P' for any P, and we know that any $\{P, P'\}$ pair has at most two children in the computation tree. Each P must be an internal node of T, so there are $n - 1$ of them. Hence, the number of $\{A_i, B_i\}$ pairs in this case is at most $2(n - 1)(3(s\sqrt{d} + 2\sqrt{d} + 1) + 2)^d$, which is $O(n)$. $\square$

We observe that the preceding proof is constructive, and provides a sequential algorithm whose time is linear in its output size. By our lemma, this is $O(n)$, which gives us the following theorem:

THEOREM 4.3. *Given a set P of n points in $\mathfrak{R}^d$ and a fair split tree of P, a well-separated pair decomposition of P can be constructed in $O(n)$ time.*

5. Minimum Depth for Linear-Size Realizations

It is easy to see that the depth of fair split trees can be linear in the worst case. One is tempted to modify the definition of a fair split tree to ensure that its

depth is bounded by a polylog function, since this would greatly simplify parallel algorithms that used it. In this section, we will show that there exist point sets P, such that any tree permitting a linear size well-separated realization of $P \otimes P$ must have depth $\Omega(n^\epsilon)$, for some $\epsilon > 0$.

We show that the lower bound holds even for $d = 1$, by constructing a class of one-dimensional sets for which the bound is attained. Let $\alpha = 2/s + 1$, where s is the separation. For all $n \geq 1$, define $\tilde{P}_n$ to be the set of points $\{\alpha^0, \ldots, \alpha^{n-1}\}$ on the real line.

We show that if the depth of any binary tree associated with $\tilde{P}_n$ is $d(n)$, then the number of pairs in a well-separated realization is at least $cn \log n / \log d(n)$ for some constant $c > 0$. The proof involves a reduction to a simple coding problem. (See Abramson [1963] for a discussion of coding theory.) Consequently, if the total number of pairs is no more than en for some constant e, then $d(n)$ is no less than $n^{c/e}$.

Assume $\tilde{P}_n$ has an associated binary tree T, and consider a well-separated realization of $\tilde{P}_n \otimes \tilde{P}_n$, denoted $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$, that uses T. For all $j = 1, \ldots, k$, A_j and B_j are well separated, so one of them lies entirely to the right of the other. Assume without loss of generality that it is B_j. We note the following property.

LEMMA 5.1.   *For all $j = 1, \ldots, k$, $B_j = \{\alpha^i\}$ for some i such that $1 \leq i \leq n - 1$.*

PROOF.   First note that B_j cannot contain α^0 since there is nonempty set A_j to its left. Now, suppose there were a set B_j containing two points α^i, $\alpha^{i'}$ with $i < i'$. Let r denote the radius of the smallest d-ball containing $R(B_j)$. Clearly, $r \geq (\alpha^{i'} - \alpha^i)/2$. Because $\alpha^{i'} \geq \alpha^{i+1}$, we have $r \geq \alpha^i(\alpha - 1)/2 = \alpha^i/s$, or $\alpha^i \leq sr$. Since A_j is nonempty and to the left of B_j, it contains at least one point in the interval $(0, \alpha^i)$. But all points in this interval are at a distance less than α^i from B_j. Hence, A_j and B_j are not well separated, giving us a contradiction. $\square$

For all $i = 1, \ldots, n - 1$, let S_i denote the set of all j, $1 \leq j \leq k$, such that $B_j = \{\alpha^i\}$. For all $i', \ldots, i - 1$, there must exist a unique $j \in S_i$, such that $\alpha^{i'} \in A_j$. Hence, it is clear from Lemma 5.1 that

$$\bigcup_{j \in S_i} A_j = \{\alpha^0, \ldots, \alpha^{i-1}\} \tag{4}$$

and $k = \sum_{i=1}^{n-1} |S_i|$.

Since we are proving a lower bound, we place no restrictions on the tree associated with a point set P, except to assume without loss of generality that it is binary. We will refer to this tree as the *subset tree* of P.

LEMMA 5.2.   *There exists a constant $c > 0$ such that for all n, there is a point set P, with $|P| = n$, such that for all subset trees of P with depth at most $d(n)$, the size of the smallest well-separated realization of $P \otimes P$ that uses the subset tree of P is at least $cn \log n / \log d(n)$.*

PROOF.   Consider a subset tree of $\tilde{P}_n$, for some $n \geq 1$, with depth at most $d(n)$. For every node in the subset tree, find the maximum i such that α^i is in the set at that node. We observe that the maximum of every internal node A is the same as that of one of its children, which we call A_1, and greater than that

of its other child, which we call A_2. For every A, we assign 1 to the edge $\{A, A_1\}$ and 0 to the edge $\{A, A_2\}$, resulting in a $\{0, 1\}$ assignment for all the edges of T.

Using (4), we can easily show that for every i, $|S_i|$ is at least as large as the number of 1's on the path from the root to the leaf α^i. Since $k = \sum_{i=1}^{n-1}|S_i|$, we can find a lower bound for k by seeking a tree with n leaves, having depth no more than $d(n)$, with an assignment of 0's and 1's to its edges such that the edges out of each node are labelled distinctly, and the sum of the number of 1's on each root-to-leaf path is minimized. This is equivalent to finding an assignment of $\{0, 1\}$ strings to n numbers such that no two numbers are assigned the same strings, no string is longer than $d(n)$, and the total number of 1's is minimized.

Note that the number of strings we can represent using at most l 1's is bounded above by $\sum_{i=0}^{l}\binom{d(n)}{i}$. Using the crude upper bound of $(l + 1)(d(n))^l$, we argue that there is a constant $c > 0$ such that if $l \le 2c \log n/\log d(n)$, then we can represent at most $n/2$ strings. It follows that at least $n/2$ strings contain more than $2c \log n/\log d(n)$ 1's, and this establishes the lemma. $\square$

As a consequence, we have the following theorem.

THEOREM 5.3. *For any n and d, there exists a set P of n points in $\mathfrak{R}^d$ such that any subset tree of P that results in $O(n)$ well-separated pairs must be of depth $\Omega(n^\epsilon)$, for some $\epsilon > 0$.*

PROOF. If en is a bound on the number of pairs, then choose $\epsilon = c/e$, and apply Lemma 5.2. $\square$

Note that we can always construct an $O(\log n)$-depth tree that permits an $O(n \log^2 n)$ realization. We form such a tree from a fair split tree T as follows: Assume that the leaves of T are numbered in inorder. Then any node in T contains all the leaves whose numbers fall within some interval. Let T' be $\lceil \log_2 n \rceil$ depth tree whose leaves have the same inorder numbering. Clearly, any node in T can be represented by the union of $O(\log n)$ nodes in T'. It follows that any well-separated pair in a decomposition that uses T can be represented by $O(\log^2 n)$ well-separated pairs in a decomposition that uses T', resulting in a realization of size $O(n \log^2 n)$. Note that the nodes of T' are no longer defined by splitting hyperplanes.

Rather than bounding the number of pairs of the form $\{A, B\}$, one may sometimes be interested in bounding the sum of $|A| + |B|$ taken over all pairs. Unfortunately, this can be $\Theta(n^2)$ in the worst case, as can be seen from the construction used in the preceding lower bound. This follows from the fact that every well-separated pair in the decomposition of $\tilde{P}_n$ must contain a singleton element. Because the realization is an exact covering of the set of unique pairs, the sum of $|A||B|$ taken over all pairs is exactly $\binom{n}{2}$. Hence, the sum of $|A| + |B|$ taken over all pairs is exactly $\binom{n}{2} + k$, where k is the number of pairs in the realization.

6. *Finding a Fair Split Tree in Parallel*

To parallelize the algorithm for constructing a fair split tree, it is sufficient to parallelize the splitting stage. Because each stage decomposes a set into sets

that are no more than half of its size, the number of stages is $O(\log n)$, allowing the algorithm to be performed efficiently in parallel.

The split obtained by the parallel algorithm *might not be the same* as that of the sequential algorithm, since it is hard to guarantee that each rectangle will be split exactly in half perpendicular to its longest dimension. However, we can guarantee that it will be a fair split as defined earlier, so that the previous results will hold.

Our method will depend on the following lemma:

LEMMA 6.1. *Let R be a d-dimensional 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 $l_{\max}(R') \leq \frac{1}{2}l_{\max}(R)$ and R' contains $\lfloor n/2^d \rfloor$ 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 $\lfloor n/2^d \rfloor$ points in the set. Choose such a rectangle and call it $\tilde{R}$. Now, consider all the similar rectangles contained in $\tilde{R}$ sharing the same corner of R as $\tilde{R}$, and that have a point touching their boundary. Since no two points touch the boundary of the same rectangle, one of these rectangles contains exactly $\lfloor n/2^d \rfloor$ points inside (including the point on the boundary). Call this rectangle R'. Note that $l_{\max}(R') \leq \frac{1}{2}l_{\max}(R)$. $\square$

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 $\hat{R}(P)$, inherited from the preceding stage if P is not the root. Let R be a rectangle containing P and contained within $\hat{R}(P)$, that satisfies $l_{\min}(R) \geq 2l_{\max}(\hat{R})/3$. Our splitting algorithm will guarantee that such an R always exists. We find a rectangle R' fulfilling the properties of Lemma 6.1.

We wish to decompose the set of m points not in R' into a collection of sets formed by a sequence of fair splits. Assume the points in R are sorted by increasing order of the size of the similar rectangle sharing the same corner as R', and with the point touching its boundary.

Our algorithm proceeds by divide-and-conquer. At each step, if we have not reached a terminal case, we can either split the problem with m points into two problems containing $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$ points, or else we can split it into a terminal case and one containing no more than $\lceil m/2 \rceil$ points. The number of steps is thus $O(\log m)$.

At each step in the algorithm, we are given R and R', and we assume the following inequalities hold:

$$l_{\min}(R) \geq \frac{2l_{\max}(R)}{3} \tag{5}$$

$$l_{\max}(R') \leq \frac{2l_{\max}(R)}{3}. \tag{6}$$

It is clear that these conditions are satisfied by our initial choice of R and R'. There are two terminal cases:

Terminal case 1. Suppose $m \leq 1$. Then there is at most one point outside R', and it can be separated from R' using a single fair split.

Terminal case 2. Suppose $l_{\max}(R') \geq l_{\max}(R)/3$. Then we decompose R by a succession of at most d fair splits. To do this, we consider each dimension of R in decreasing order of length. At each point, we can either perform a split with a hyperplane that contains the corresponding face of R', or else the region on the other side of this hyperplane is empty, and we do not need to perform a split. Conditions (5) and (6) guarantee that such splits are fair. When we finish, the final set will contain exactly those points in R'. The result is a split tree that includes R' along with at most d other enclosing rectangles that partition the set of points outside R'.

Note that the terminal cases can result in point sets that violate $l_{\min}(\hat{R}(A)) \geq 2l_{\max}(A)/3$. However, they will always satisfy $l_{\min}(\hat{R}(A)) \geq l_{\max}(A)/3$. In $O(1)$ steps, we can split any such point set into one that satisfies the first inequality by a recursive process of splitting the bounding rectangle exactly in half along each dimension i in decreasing order of length until the inequality is satisfied. The split will never be repeated along the same dimension, so the result will be a tree of fair splits with depth at most d.

If neither of the terminal cases is true, we can find a rectangle R'' that satisfies one of the following cases:

(a) $3l_{\max}(R')/2 < l_{\max}(R'') < 2l_{\max}(R)/3$, and R'' contains $\lfloor m/2 \rfloor$ points not contained in R' (hence, R contains $\lceil m/2 \rceil$ points not contained in R'').

(b) $l_{\max}(R'') = 2l_{\max}(R)/3$ and R'' contains no more than $\lfloor m/2 \rfloor$ points not in R'.

(c) $3l_{\max}(R')/2 = l_{\max}(R'')$ and R contains no more than $\lceil m/2 \rceil$ points not in R''.

It is easy to verify that, if cases (b) and (c) do not hold, then (a) must hold. Hence, all possibilities are covered. We recurse on rectangle pairs R', R'' and R'', R. Note that these pairs satisfy (6). The split trees obtained for these can be combined to form a split tree of R', R. In all cases in which the number of points has not been cut in half, the corresponding rectangle pair is covered by the second terminal case. Hence, the depth of recursion is $O(\log n)$. We can obtain parallel time of $O(\log n)$ if each step can be done in constant time.

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.

Now, assume that we have a list of similar rectangles $R_1, \ldots, R_n$, sorted in order of increasing size, all sharing a corner, and each having a point in the set on their boundary. For each rectangle R_i, find the smallest rectangle $\text{next}(R_i)$ such that $l_{\max}(\text{next}(R_i)) \geq 3l_{\max}(R_i)/2$, and the largest rectangle $\text{prev}(R_i)$ such that $l_{\max}(\text{prev}(R_i)) \leq 2l_{\max}(R_i)/3$.

Given the above preprocessing, we can perform each nonterminal step as follows: We are given R and R', represented by the locations in the sorted

array containing the smallest rectangles greater than or equal to each in size. Computing the midpoint of these locations can be done in constant time, and gives us an R'' fulfilling the properties of case (a) except, possibly, the condition $3l_{\max}(R')/2 < l_{\max}(R'') < 2l_{\max}(R)/3$. In this case, either a rectangle containing the points in prev(R) satisfies case (b), or a rectangle containing the points in next(R') satisfies case (c).

Thus, the splitting phase, including preprocessing, takes at most $O(\log n)$ time with n processors. The final result is a partial split tree in which all nodes A have size at most

$$\left\lceil \frac{2^d - 1}{2^d} n \right\rceil$$

and satisfy $l_{\min}(\hat{R}(A)) \geq 2l_{\max}(A)/3$.

Each point set A returned by this phase contains no more than

$$\left\lceil \frac{2^d - 1}{2^d} n \right\rceil$$

points, and its outer rectangle satisfies $l_{\min}(\hat{R}(A)) \geq 2l_{\max}(A)/3$.

Now, we need only recurse on the point sets determined by the splitting phase. The condition on $\hat{R}(A)$ allows us to choose a suitable R for each point set in the next iteration. The depth of recursion will be at most $O(\log n)$, so the total parallel time will be $O(\log^2 n)$ using $O(n)$ processors. This gives us the following theorem:

THEOREM 6.2. *Given a set P of n points in $\Re^d$, a fair split tree can be constructed in $O(\log^2 n)$ time using $O(n)$ processors.*

7. Finding the Well-Separated Realization in Parallel

Given a fair split tree of P, we would like to find a linear size well-separated realization of $P \otimes P$ with the properties stated in Lemma 4.2. We will assume that for any node A we can obtain $R(A)$ in constant time, since it can clearly be stored at each node when the split tree is being computed. Unlike the algorithm of the preceding section, this parallel algorithm will produce *precisely the same output* as the sequential algorithm.

As a step towards a parallel algorithm, we consider a modification of our sequential procedure that bounds the depth of recursion to $O(\log n)$. This modification results in a phase that we can perform in parallel, and which we need only perform $O(\log n)$ times for the entire algorithm.

Suppose we are computing the realization of $P \otimes P'$ using the procedure given earlier. We use $|P||P'|$ as a measure of the size of the current problem. Note that at each recursive step, we split the problem into two subproblems whose combined size is equal to the size of the original.

Consider an iterative procedure in which we store the smaller subproblem on a list, breaking ties arbitrarily, and continue to split the larger subproblem until we reach a terminal case. We add the final pair to our list of well-separated pairs. By applying this algorithm recursively to each problem in the list, we will construct the same set of well-separated pairs as in the original procedure.

Let $\hat{P}$ and $\hat{P}'$ be the pair before the iteration. Then, for every P, P' on our list of subproblems, $|P||P'| \leq |\hat{P}||\hat{P}'|/2$. Clearly, $|P||P'| = 1$ is a sufficient,

though not necessary, condition to terminate with a well-separated pair. This condition will be satisfied after at most $\log|P\|P'|$ recursive calls. Since $|P|, |P'|$ $\leq n$, we have $\log|P\|P'| \leq 2\log n = O(\log n)$.

We now consider the problem of constructing the list of subproblems in $O(\log n)$ time using $O(m)$ processors, where m is size of the list. First, we need to develop a characterization of the list that does not depend on a sequential algorithm.

For each internal node P in the split tree, we denote its children by $c_{\text{large}}(P)$ and $c_{\text{small}}(P)$, where $|c_{\text{large}}(P)| \geq |c_{\text{small}}(P)|$. We break ties arbitrarily to guarantee that these names are uniquely defined.

Let A^0 and B^0 be the initial point sets. Consider the lists $L_A = A^0, \ldots, A^{m_A}$ such that $A^i = c_{\text{large}}(A^{i-1})$ for all $i = 1, \ldots, m_A$, and $L_B = B^0, \ldots, B^{m_B}$ such that $B^j = c_{\text{large}}(B^{j-1})$ for all $j = 1, \ldots, m_B$. Let $L = P^0, \ldots, P^m$ be an interleaving of L_A and L_B that obeys the inequality $l_{\max}(P^i) \leq l_{\max}(P^{i-1})$ for all $1 \leq i \leq m$.

We require that m_A and m_B be chosen so that A^{m_A} and B^{m_B} are well separated, but either A^{m_A} and B^{m_B-1}, or A^{m_A-1} and B^{m_B} are not well separated. Also, if $P^m = A^{m_A}$, then $l_{\max}(c_{\text{large}}(B^{m_B})) \leq l_{\max}(P^m)$, and if $P^m = B^{m_B}$, then $l_{\max}(c_{\text{large}}(A^{m_A})) \leq l_{\max}(P^m)$. Note that the term A^{m_A-1}, B^{m_B-1}, $c_{\text{large}}(A^{m_A})$, and $c_{\text{large}}(B^{m_B})$ are not always defined. In such cases, the corresponding requirement is omitted.

The list L represents the order in which the point sets are split by the sequential algorithm, taking into account the manner in which ties were broken. Consider any two integers $i < j$ satisfying one of the following conditions for some i', j':

(i) $P^i = A^{i'}$, $P^j = B^{j'}$, and $\forall j'' > i, P^{j''} \neq B^{j'-1}$
(ii) $P^i = B^{i'}$, $P^j = A^{j'}$, and $\forall j'' > i, P^{j''} \neq A^{j'-1}$

Now, let $P = P^i$ and let P' be the sibling of P^j in the split tree. We claim that the set of P, P' defined in this way is the same as the set of subproblems constructed by the sequential algorithm. To see this, note that P, P' will be added to the list of subproblems at the point when $p(P^i)$ or $p(P^j)$ is split. This will have to occur at some point before P^j is split, because it is smaller (geometrically) than P^i. If we can construct L, then we can easily retrieve all P, P' pairs.

To permit the rapid construction of L, we need to preprocess the split tree. This will be done once, before our algorithm begins, and will take $O(\log n)$ time. For every node P in the split tree we find a pointer to $c_{\text{large}}(P)$. We can do this in constant time with n processors, assuming we have computed the size of the set at every node. This partitions the nodes of the tree into a number of linked-lists. We use standard list-ranking techniques to construct arrays containing these lists in order. For each node of the split tree, we find a pointer to the array containing it, and its position in the array. It is well known that this can be done in $O(\log n)$ time using $O(n/\log n)$ processors.

Given A^0 and B^0, our preprocessing allows us to easily retrieve two lists, $\hat{L}_A = A^0, \ldots, A^{\hat{m}_A}$ and $\hat{L}_B = B^0, \ldots, B^{\hat{m}_B}$, which contain L_A and L_B as prefixes. Hence, to determine L_A and L_B, we need only compute m_A and m_B satisfying the criteria given earlier. We can compute these in $O(\log n)$ time with a single processor, using a form of binary search.

Let sep(i, j) be a predicate that is true iff A^i and B^j are well separated. Note that sep(i, j) implies sep(i', j') for all $i' \geq i, j' \geq j$. In other words, sep(i, j) is monotone in i and j. Consider the matrix formed by taking i as the row, and j as the column. Note that (m_A, m_B) must lie on the boundary between the region where sep(i, j) is false and the region where it is true. We can characterize these regions, not including the boundary, by:

(1) $\{(i, j) | \neg \text{sep}(i, j)\}$

(2) $\{i, j) | \text{sep}(i - 1, j) \wedge \text{sep}(i, j - 1)\}$

Now consider the two regions defined as follows:

(I) $\{(i, j) | l_{\max}(A^{i+1}) > l_{\max}(B^j)\}$

(II) $\{(i, j) | l_{\max}(B^{j+1}) > l_{\max}(A^i)\}$

Note that the condition for (I) is monotone in $-i$ and j, and the condition for (II) is monotone in i and $-j$. It is easy to verify that our condition for m_A and m_B is equivalent to stating that the matrix element at (m_A, m_B) does not lie in any of the above regions. Using the monotonicity properties, we can verify the correctness of the following table:

(i, j) in	(I)	neither	(II)
(1)	$m_A > i$	$m_A \geq i$ $m_B \geq j$	$m_B > j$
neither	$m_A \geq i$ $m_B \leq j$	$m_A = i$ $m_B = j$	$m_A \leq i$ $m_B \geq j$
(2)	$m_B < j$	$m_A \leq i$ $m_B \leq j$	$m_A < i$

Since $\hat{L}_A$ and $\hat{L}_B$ are stored in arrays, we can calculate the position of the middle element of either list in constant time. Using the preceding table, we can perform a binary search that eliminates half of at least one of the lists at each step. Hence, we can find m_A and m_B in $O(\log n)$ time using a single processor.

Since L is the merge of L_A and L_B with respect to an easily computed total ordering, we can readily compute L in parallel in $O(\log n)$ time with $O(m)$ processors. It might seem as if we are better off doing the merge directly on $\hat{L}_A$ and $\hat{L}_B$, allowing us to simplify the binary search. However, $|\hat{L}_A| + |\hat{L}_B|$ could be much larger than m, and the resulting algorithm might not maintain a linear processor bound when many merges are being done simultaneously.

To complete the algorithm, we simply perform the above phase until all well-separated pairs have been found. Since each phase reduces the size of each subproblem by at least a factor of two, the number of phases will be $O(\log n)$. Each phase can be performed in $O(\log n)$ time, so the total parallel time bound is $O(\log^2 n)$.

To see that the processor bound is $O(n)$, note that the total number of processors assigned to all merges being done at any parallel step is linear in the number of new subproblems created. Each subproblem generates at least one new well-separated pair, so by Lemma 4.2, the number of new subproblems is $O(n)$. Therefore, the processor bound is also $O(n)$.

THEOREM 7.1. *Given a set P of n points in $\Re^d$ and a fair split tree of P, a well-separated pair decomposition of P can be constructed in $O(\log^2 n)$ time with $O(n)$ processors.*

8. *Computing k-Nearest-Neighbors*

One useful application of the well-separated pair decomposition is an optimal sequential algorithm for computing the k-nearest-neighbors of each point in a set. (See Preparata and Shamos [1985] for a discussion of this and other problems in computational geometry.) In this section, we present such an algorithm. The complete method, including the computation of the split tree and realization, is reminiscent of Vaidya's algorithm [1986; 1989]. However, the way in which we have split our algorithm into phases makes it easier to parallelize efficiently. Our algorithm for the special case of all-nearest-neighbors (corresponding to $k = 1$, when interpoint distances are unique) is somewhat simpler, and can be found in Callahan and Kosaraju [1992].

As in the preceding section, we will assume that for every node A in T, $R(A)$ was stored at that node when the split tree was being computed. This will be necessary for constant-time tests involving the distance from a point to the smallest d-ball containing $R(A)$.

Given a well-separated pair decomposition of P, we can compute the k-nearest-neighbors of P with only $O(kn)$ additional steps. Note that this is optimal and improves on the $O(n \log n + kn \log k)$ complexity of Vaidya's algorithm. The output of the k-nearest-neighbors algorithm is a list of all pairs (a, b), such that b is one of the k-nearest-neighbors of a. We refer to (a, b) as a k-nearest-neighbor pair.

LEMMA 8.1. *Let P be a point set, and let $\{A, B\}$ be a pair in a well-separated realization of $P \otimes P$, with a separation $s > 2$. Suppose that there is a k-nearest neighbor pair (a, b) such that $a \in A$ and $b \in B$. Then $|A| \leq k$.*

PROOF. Suppose $|A| > k$. Let $A' = A - \{a\}$. Then because A and B are well separated, $d(a, a') < d(a, b)$ for all $a' \in A', b \in B$. Since $|A'| \geq k$, B cannot contain a k-nearest-neighbor of a. $\square$

Note that as a consequence of this lemma (taking $k = 1$), if $\{a, b\}$ is the closest pair, then $\{\{a\}, \{b\}\}$ is a pair in the realization of $P \otimes P$. Hence, given the realization, we can find the closest pair in P in $O(n)$ time. This provides a very simple proof that our sequential algorithms are optimal, since $\Omega(n \log n)$ is a lower bound for the closest pair problem.

For the following, we use $\overline{xy}$ to denote the line segment between points x and y.

Let B be a point set, and let O_B be the center of the smallest d-ball containing $R(B)$. Let $\{a\}$ and $\{a'\}$ be singleton point sets, each well separated

from B, such that $d(a, O_B) \geq d(a', O_B)$, and let α represent the angle between $\overline{aO_B}$ and $\overline{a'O_B}$, given in radians.

LEMMA 8.2.   *If* $\alpha < s/(s + 1)$, *then* $d(a, a') < d(a, b)$ *for all* $b \in B$.

PROOF. Note that we can scale each dimension equally without affecting our claim. To simplify the analysis, we scale the points so that the smallest d-ball containing $R(B)$ has radius 1. Now we confine our attention to the plane containing $\overline{aO_B}$ and $\overline{a'O_B}$.

For convenience, let r and r' denote $d(a, O_B)$ and $d(a', O_B)$, respectively. We observe the following bound:

$$d(a, a') \leq r - (1 - \alpha)r'. \tag{7}$$

The right-hand side of this inequality is the length of the path obtained by going from a' along an arc segment of length $\alpha r'$ to the point on $\overline{aO_B}$ at distance r' from O_B, and going from this point to a along a line segment of length $r - r'$. Clearly, this is an upper bound on $d(a, a')$.

Because $\{a\}$ and $\{a'\}$ are well separated from B, both are at distance at least $s + 1$ from O_B. Hence, inequality (7) is preserved when we substitute $s + 1$ for r'. We can also substitute $s/(s + 1)$ for α since we are assuming this is a strict upper bound. With these substitutions, (7) becomes a strict inequality, and simplifying, we obtain $d(a, a') < r - 1$. We observe that $d(a, b) \geq r - 1$ for all $b \in B$, so $d(a, a') < d(a, b)$ for all $b \in B$.   □

Consider a convex cone C whose apex lies at O_B, and let θ denote the angle of C, defined as the maximum possible angle between $\overline{O_B x}$ and $\overline{O_B y}$ for any x and y in C. Let A be a set of points that all lie in C.

LEMMA 8.3.   *Let* P *be a point set containing* $A \cup B$ *as a subset, and let* a *be a point in* A. *If* $\theta < s/(s + 1)$ *and* $|\{a' \in A - \{a\}: d(O_B, a') \leq d(O_B, a)\}| \geq k$, *then no* $b \in B$ *can be a* k-*nearest-neighbor of* a *in* P.

PROOF. By Lemma 8.2, every $a' \in A - \{a\}$ such that $d(O_B, a') \leq d(O_B, a)$ satisfies $d(a, a') < d(a, b)$ for all $b \in B$. Hence, if $|\{a' \in A - \{a\}: d(O_B, a') \leq d(O_B, a)\}| \geq k$, then a has k neighbors in A nearer to it than any $b \in B$. Therefore, no $b \in B$ can be a k-nearest neighbor of a in P.   □

Yao [1982] defines a *frame* to be a partition of d-dimensional space into cones whose apexes all lie at the origin. The angle of a frame is defined to be the maximum angle over all its cones. Yao has shown that for any d and any $\psi > 0$ one can construct a finite frame with angle less than ψ. Note that the size of the frame depends on d and ψ.

To apply the above results to the k-nearest-neighbor problem, we assume we have a well-separated pair decomposition of our point set P with $s > 2$ (say $s = 2.1$).

For every node B in T, let $f(B)$ be the set of all a such that for some ancestor B' of B, $\{A, B'\}$ $(|A| \leq k)$ is in the realization and $a \in A$. Lemma 8.1 guarantees that if (a, b) is a k-nearest-neighbor pair and $b \in B$, then $a \in f(B)$. For node B, we compute $N(B)$, some subset of $f(B)$ that includes all $a \in A$ such that $\{A, B\}$ is a pair in the realization, and (a, b) is a k-nearest-neighbor pair for some $b \in B$.

Using Lemma 8.3, we can find an $N(B)$ of size $O(k)$ for every B in T. Note that the constant in the O-notation depends on d. The computation is done top down, starting at the root and ending at the leaves. The set for the root P is the empty set, since every point is in P. The initial set for any node B is the union of $N(p(B))$ and the set of a such that $\{A, B\}(|A| \leq k)$ is a pair in the realization and $a \in A$.

In time proportional to the size of this set we can compute $N(B)$, as follows: Construct a frame of angle less than $s/(s + 1)$, and translate it so that O_B is the origin. Now, partition the initial set into a constant number of subsets of points lying in each cone of the frame. For each of these subsets, select the k that are nearest to O_B. Note that we can perform this operation in linear time for each subset by selecting the kth nearest point (or maximum if the set has less than k elements) and then retrieving all points nearer to O_B than this element (and possibly some with the same distance). We let $N(B)$ consist of the set of all the points thus selected.

By Lemma 8.3, the points that we have not selected can be eliminated from $N(B)$. The overall top down computation takes $O(kn)$ steps, since the size of the set passed from any node to its children is $O(k)$.

Suppose b is a k-nearest-neighbor of a. Using the preceding arguments, we can establish that a will be in $N(\{b\})$. Now, for each a, we compute the set of all b such that $a \in N(\{b\})$, and select the k nearest b in each set, giving us the k-nearest-neighbors of a. There are at most $O(kn)$ total elements over which we perform selection, so the complexity of this step is $O(kn)$.

THEOREM 8.4. *Given a set P of n points in $\Re^d$ and a well-separated pair decomposition of P, the k-nearest neighbors of all points in P can be computed in $O(kn)$ steps.*

8.1. COMPUTING k-NEAREST-NEIGHBORS IN PARALLEL. It is well known that given an arithmetic expression tree whose leaves are constants and whose internal nodes are labelled 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. A standard approach to this problem is to apply the technique of rake-and-compress [Abrahamson et al. 1989; Gazit et al. 1988; Kosaraju and Delcher 1988], which is suitable for a wide variety of tree-structured computation problems.

In Callahan and Kosaraju [1992], we sketched a parallel algorithm for the all-nearest-neighbors problem using rake-and-compress [Abrahamson et al. 1989; Gazit et al. 1988; Kosaraju and Delcher 1988]. We can adapt this algorithm to the k-nearest-neighbors problem as follows.

At any stage of forward raking, let B' be the parent of B. At this stage, node B will store the set $N(B', B)$ defined below.

Let $f(B', B)$ be the set of all $a \in A$ such that $\{A, B''\}$ $(|A| \leq k)$ is a well-separated pair for some node B'' on the path from B to B', B inclusive, in the original tree. Then $N(B', B)$ is the subset of $f(B', B)$ that includes the k nearest points to $O_{B'}$ in each cone in the frame around B'.

This algorithm can be made to run in $O(\log n \log k)$ steps with $O(nk/\log n)$ processors on an EREW PRAM. If we wish to reduce the time to $O(\log n)$ steps independent of k, we run into some difficulty, since it appears that we need to apply k selection at each of $O(\log n)$ phases of the algorithm. We can eliminate this using several further geometric observations.

Roughly speaking, the well-separated pairs corresponding to a given node tend to represent decreasing distances as the depth of the node increases. Hence, rather than applying k selection in a top down fashion, we can use the structure of the fair split tree directly to obtain a set of $O(kn)$ pairs that contains all k-nearest-neighbor pairs.

In our sequential algorithm, we partitioned the space around each node in T into narrow cones, and selected the k nearest points within each cone. There was nothing to prevent points from changing cones during this process. If we are careful, however, we can choose our cones so that a point remains in the same cone as it is pushed down to the leaves of T.

For our purposes, it will be convenient to define each cone by an axis vector u. Let α denote the angle and b denote the apex of such a cone. Then, the cone of u consists of all points x lying above the hyperplane passing through b with normal u such that $\overline{xb}$ makes an angle of at most $\alpha/2$ with u. We can choose a set U of vectors with $|U|$ dependent only on s and d, such that for any well-separated pair $\{\{a\}, B\}$, there is a $u \in U$ such that for all $b \in R(B)$, a lies in the cone of u with apex b and angle $\alpha = O(s^{-1})$. This follows easily from the geometry of well-separated sets.

For our parallel algorithm, we first need to construct a well-separated pair decomposition with $s > 2$ sufficiently large to insure that some $\alpha < s/(s + 1)$ satisfies the above claim. The set of cones defined by the vectors in U covers the space around the origin in a manner similar to a frame, but the cones may overlap.

For every node B, we retrieve the set of a such that $\{A, B\}$ $(|A| \le k)$ is a pair in the realization and $a \in A$. Now, we associate every pair $\{B, \{a\}\}$ with some u such that a lies in the cone of u with apex at O_B. This can be done in constant time with $O(kn)$ processors by assigning a processor to each pair.

We perform a single pass of k selection to obtain for each B and u, the k nearest a within the cone of u. This can be done in $O(\log n)$ time with $O(kn)$ processors. We use $F(B)$ to denote the set of all a thus paired with B. By Lemmas 8.1 and 8.3, we know that for any k-nearest-neighbor pair (a, b) there is an ancestor B of b such that $a \in F(B)$.

Having guaranteed that every point is assigned to the same cone throughout the top down procedure, we can simplify our description of the set of points to compute at the leaves of T. Specifically, for every point b in P, we wish to compute, for each $u \in U$, the k nearest points a such that $\{B, \{a\}\}$ is a pair associated with u for some ancestor B of b.

To solve this new version of the problem, we exploit the fact that smaller distances tend to occur deeper in the tree. The ordering is not exact, so we will not try to obtain the k nearest points immediately. Instead, we will obtain, for each leaf in T and vector u, a set of size $O(k)$ that contains as a subset the k nearest points within the cone of u that have not already been eliminated by Lemma 8.1. We require several geometric results.

Suppose $\{A_i, B_i\}$ is a pair in a realization constructed by the algorithm in Section 4, and let $d(A_i, B_i)$ denote the minimum distance between $R(A_i)$ and $R(B_i)$. Then, for some d-dependent constant c,

$$sl_{\max}(A_i) \le d(A_i, B_i) \le csl_{\max}(p(A_i)). \tag{8}$$

Clearly $d(A_i, B_i) \ge sl_{\max}(A_i)$, since otherwise the pair would not be well separated. To prove the other half of (8), we need to consider several cases. A

trivial case occurs when $p(A_i) = p(B_i)$. Otherwise, there are two ways in which our algorithm could have formed the pair $\{A_i, B_i\}$. If $\{A_i, B_i\}$ was formed by splitting $p(A_i)$, then $p(A_i)$ and B_i are not well separated, so the inequality holds. Otherwise, $\{A_i, B_i\}$ was formed by splitting $p(B_i)$, so $d(A_i, B_i) \leq csl_{\max}(p(B_i))$. In this case, $p(A_i)$ was split at some earlier point. Hence, $l_{\max}(p(A_i)) \geq l_{\max}(p(B_i))$, and the inequality holds here as well.

We now consider an additional property of fair split trees. Let A be a node in some fair split tree T, and let $p^j(A)$ denote the ancestor of A that is j levels above it in the tree, if such an ancestor exists. Then, we have the following inequality.

$$l_{\max}(A) \leq \frac{2}{3} l_{\max}(p^{6d}(A)). \tag{9}$$

To see this, note that after a fair split is performed 3 times perpendicular to dimension i, then l_i can be no more than $2/3$ its original value. That is, if the first two splits did not reduce l_i sufficiently, then they must have each been close to opposite boundaries of $R(A)$ in the ith dimension. At this point, the third split must divide this length into two pieces whose size is no more than $2/3$ the original in order to ensure the fair split condition. Moreover, once l_i has been reduced to $4/9$ its original size, a fair split cannot occur in the ith dimension until $l_{\max}$ has been reduced to $2/3$ its original size. Hence, after $6d$ splits, $l_{\max}$ is no more than $2/3$ its original value. In practice, this is almost certainly an overestimate.

We say that a is *paired with* B when $a \in F(B)$. Let $S(B, u, b)$ denote the set of all a in the cone of u paired with b or one of its ancestors up to and including B. Also, let $B(u, b)$ denote the nearest ancestor of b such that $|S(B(u, b), u, b)| \geq k$.

LEMMA 8.1.1. *There exists a constant c' dependent only on d such that for any leaf b in T and $u \in U$, $S(p^{c'}(B(u, b)), u, b)$ contains the k nearest a to b in the cone of u paired with some ancestor of b.*

PROOF. Let a' denote the kth nearest neighbor of b in the cone of u paired with some ancestor of b. For at least one $a'' \in S(B(u, b), u, b)$, $d(a'', b) \geq d(a', b)$, because $S(B(u, b), u, b)$ has at least k elements. It follows from the well-separated set condition that $d(a'', B) \geq s/(s + 2) d(a', b)$, where B is the node paired with a''. Combining (8) and (9), we can determine some constant c' such that for all $a \in S(T, u, b) - S(p^{c'}(B), u, b)$, $d(a, b) \geq d(a', b)$. Hence, $S(p^{c'}(B), u, b)$ contains the k nearest a to b in the cone of u paired with some ancestor of b. B is either $B(u, b)$ or one of its descendents, so the original claim follows. $\square$

We now need to retrieve $N(u, b) = S(p^{c'}(B(u, b)), u, b)$ for each u and b. This is similar to the problem of retrieving, for every leaf in a tree, its k nearest ancestors. In fact, it can be reduced to a weighted version of the latter problem by constructing a separate tree for each vector u, and compressing out all nodes B not paired with any a in the cone of u. This is no longer a geometric problem, and can be done $O(\log n)$ time with $O(kn)$ processors using standard parallel techniques.

Finally, for each leaf a in T, we know by previous arguments that its k nearest neighbors are among those b such that $a \in N(u, b)$ for some $u \in U$.

We can easily retrieve the k-nearest-neighbors among these points in $O(\log n)$ time with $O(kn)$ processors, thus completing our parallel algorithm.

THEOREM 8.1.2. *Given a set P of n points in $\mathfrak{R}^d$ and a well-separated pair decomposition of P, the k-nearest neighbors of all points in P can be computed in $O(\log n)$ time with $O(kn)$ processors.*

9. *Applications to n-Body Potential Fields*

Particle simulation is a fundamental technique of scientific computing (see, for example, Aarseth et al. [1979], Appel [1985], Barnes and Hut [1986], Greengard and Gropp [1990], Miller and Prendergast [1968], Miller et al. [1970]). An essential step in such simulation is to compute the potential field of a set of particles with Coulombic or gravitational forces. For any particle at position a, we define a potential function $\phi_a(x)$ that associates a real number with every point x in the space around it, and depends only on the distance from a.

For example, in three dimensions, $\phi_a(x) = q_a/d(a, x)$, where q_a is the strength of the charge at a. This value is related to electrostatic charge in the Coulombic case and mass in the gravitational case. The potential field of a set of particles is the sum of the potential fields of each particle in the set.

We represent a collection of n distinct particles with potential fields as a set of points $\{z_1, \ldots, z_n\}$ and charge strengths associated with each point (we refer to a point as z_i in this instance to avoid confusion with the parameter p that we will later define). We would like to compute, for each point $z_i = 1, \ldots, n$, the potential due to the remaining points. The potential computation can clearly be performed in $O(n^2)$ time using the direct sum $\sum_{i \neq j} \phi_{z_j}(z_i)$ to find the potential at each z_i.

Greengard and Rokhlin [1987], Greengard [1988], and Rokhlin [1985] have developed a method of approximating this potential using truncated multipole expansions. This algorithm, known as the *Fast Multipole Method*, has received a great deal of attention in the scientific computing community.[3] Although we consider this an important (indeed, motivating) application for our data structure, we do not intend this section as an introduction to the Fast Multipole Method. Instead, we will only summarize the method here, and show how our decomposition fits into the general scheme. The interested reader is referred to Greengard's excellent exposition [Greengard 1988] for further details.

The multipole expansion of a collection of particles P is an infinite series that converges outside the sphere containing these particles and is equal to their potential field in this region. The truncated multipole expansion is an approximation of the multipole expansion containing only those terms of degree p or less. The size of this expansion depends only on p, and not on $|P|$.

For every point that is well-separated from P, the truncated multipole expansion approximates the potential field of P to a precision that depends only on p and s, the separation. In general, the precision increases monotonically with p and s, the relative error being $O(s^{-p})$. When high precision is required, there is clearly more to be gained in the long run by increasing p, but the issue is not as clear when lower precision is tolerable. This may be the case,

---

[3]See, for example, Hafner and Kuster [1991], Katzenelson [1989], Schmidt and Lee [1991], and Zhao and Johnsson [1991].

for example, when the charges themselves are not known to a high degree of precision. Because of the variety of ways of implementing the Fast Multipole Method, the most suitable trade-off between s and p will depend on many factors, and is best left as a final "tuning" step. We consider the flexible choice of s to be an advantage of our decomposition over Greengard's formulation, since his choice of s is fixed by the geometry of squares or cubes in a regular grid.

Using the truncated multipole expansion, we may approximate the potential due to arbitrarily large sets, provided we are sufficiently far away, by an arithmetic function with $O(1)$ terms. Each expansion is centered at a point x, and expansions centered at the same point are added by taking the pairwise sum of their coefficients. In addition to the multipole expansion, which describes the potential field outside a given sphere, Greengard defines the local expansion, which describes the potential field within a given sphere. Given an arbitrary sphere that is well separated from P, and a truncated multipole expansion of P, we may construct, in $O(1)$ time, a local expansion of the field due to P within this sphere, whose error bound is similar to that of the truncated multipole expansion. Local expansions may also be added, provided they are centered at the same point. Greengard shows that the centers of both types of expansions may be shifted in $O(1)$ time, allowing them to be added.

We now sketch how the above operations may be used for potential field computation given a split tree T of the set P and a well-separated realization of $P \otimes P$ that uses T. It should be emphasized that the sequential algorithm is an adaptation of the Fast Multipole Method when the split tree can have up to $O(n)$ depth.

9.1. SEQUENTIAL ALGORITHM. In the first phase, we compute the truncated multipole expansion for all nodes in T, proceeding bottom up from the leaves. We can compute the multipole expansion for all leaves in constant time, since each corresponds to a single particle. The expansion for an internal node is obtained by shifting the center of one of its children to that of the other, and adding the expansions. This also takes constant time. The total time for this phase is linear.

In the second phase, we compute, for each node, the local expansion of the potential field due to the nodes with which it is paired in the realization of $P \otimes P$. To do this, we convert the multipole expansions of the preceding phase to local expansions centered at the center of the bounding rectangle of this node, and take their sum. This requires time proportional to the total number of pairs in the realization, which is linear.

In the third phase, we proceed top down and compute a local expansion for each node in T, valid within the bounding rectangle of the set at this node, representing the field due to sets paired with this node or one of its ancestors in the realization. To compute this, we pass the local expansion of a node to its children, shift its center, and add it to the local expansion computed in the second phase.

Finally, to compute the potential at any point, we simply evaluate the local expansion of each leaf at the point in the set at this leaf. Our definition of a realization guarantees that this is the correct potential value. The total work required is clearly $O(n)$, giving us the following theorem.

THEOREM 9.1.1.   *Given a set P of n points with associated charge strengths, and a well-separated pair decomposition of P, we can compute the potential at all points in P in $O(n)$ time.*

This corresponds to the main result of Greengard [1988] and Greengard and Rokhlin [1987] but we eliminate the need for assumptions about input distribution by insisting that a well-separated pair decomposition be provided in the input.

9.2. PARALLEL ALGORITHM.   In this subsection, we sketch how the rake-and-compress technique can be applied to the parallel implementation of the Fast Multipole Method.

First, note that the operation of adding two multipole or local expansions is both commutative and associative (though we will not make use of commutativity). This follows from the fact that it is merely the exact addition of two functions. The translation operation necessary for addition only changes the representation of the function, not the function itself. Hence, there is sufficient algebraic structure to apply rake-and-compress.

Let the multipole expansion at node A be denoted $M(A)$. For the first phase, we use the fair split tree to construct an expression tree in which each leaf $\{z_i\}$ is labeled by the function $M(\{z_i\})$ (computed for all leaves simultaneously in $O(1)$ steps), and each internal node is labeled by a $+$ operator denoting the addition of two multipole expansions. The computation of $M(A)$ for all internal nodes A reduces to the problem of evaluating all nodes in this expression tree.

Now let $L(A)$ denote the local expansion at node A representing the contribution to potential due to all nodes with which A is paired in the realization. For the second phase, we first convert all multipole expansions to local expansions about the nodes with which they are paired. This can be done in $O(1)$ steps by assigning a processor to every pair in the realization. To evaluate $L(A)$ for any A, we must compute a sum of m_A terms, where m_A denotes the number of pairs of the form $\{A, B\}$. The sum of m_A over all A in T is exactly twice the number of pairs, which is linear, so this phase can be performed in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM.

Finally, let $\Phi(A)$ denote the sum of all $L(A)$ on a path from the root of T to A. For the third phase, we evaluate this using rake-and-compress as follows:

At any step in forward raking, let A be a node with parent B in the compressed tree. Node A will contain the function $\Phi_B(A)$, denoting the sum of all local expansions on the path from B to A, not including $L(B)$. We can maintain these values using a constant amount of work each time we compress a leaf. During backward raking, we use these functions to find $\Phi(A)$ for each node A as it emerges in the process of decompression. Finally, we obtain $\Phi(\{z_i\})$ for all leaves, which is the local expansion about any point in P.

All of the preceding phases can be performed in $O(\log n)$ time with $O(n/\log n)$ processors, giving us the following theorem.

THEOREM 9.2.1.   *Given a set P of n points with associated charge strengths, and a well-separated pair decomposition of P, we can compute the potential at all points in P in $O(\log n)$ time with $O(n/\log n)$ processors.*

In our potential computation algorithm, we have been treating p, the degree of the multipole expansion, as a constant. However, the dependency on p can

be important in practice. While preparing the conference version of this paper, a result of Pan et al. [1992] and Reif and Tate [1992] was brought to our attention, which significantly reduces this dependency in both two and three dimensions from that given in Greengard [1988] and Greengard and Rokhlin [1987]. It is easy to adapt their method to our framework. In this way, we remove the dependency on input precision from the time complexity of constructing the tree T. In both cases, there is no dependency on bit precision after the tree has been constructed.

10. Conclusions

It is interesting to note that our initial formulation of the well-separated pair decomposition came about in an attempt to adapt the Fast Multipole Method to guarantee a running time independent of the bit representation of the input. The term, *well-separated*, is used by Greengard, and his formulation implicitly makes use of a structure similar to the well-separated pair decomposition, but whose size can be a function of the bit representation of the input. We were somewhat surprised to discover that this decomposition could also be applied in a natural manner to the k-nearest-neighbors problem, and we believe that it will have other uses as well.

REFERENCES

AARSETH, S. J., GOTT, J. R., III, AND TURNER, E. L. 1979. *N*-body simulations of galaxy clustering. I. Initial conditions and galaxy collapse times. *Astrophys. J. 228*, 664–683.

ABRAHAMSON, K., DADOUN, N., KIRKPATRICK, D. A., AND PRZYTCHKA, T. 1989. A simple parallel tree contraction algorithm. *J. Algorithms 10*, 2, 287–302.

ABRAMSON, N. 1963. *Information Theory and Coding*. McGraw-Hill, New York.

AGARWAL, P. K., EDELSBRUNNER, H., SCHWARTZKOPF, O., AND WELZL, E. 1991. Euclidean minimum spanning trees and bichromatic closest pairs. *Disc. Computat. Geom. 6*, 407–422.

AGARWAL, P. K., AND MATOUŠEK, J. 1992. Relative neighborhood graphs in three dimensions. In *Proceedings of the 3rd Annual Symposium on Discrete Algorithms*. ACM-SIAM, New York and Philadelphia, pp. 58–65.

APPEL, A. W. 1985. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput. 6*, 85–103.

BARNES, J., AND HUT, P. 1986. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature 324*, 4, 446–449.

BENTLEY, J. L. 1980. Multidimensional divide-and-conquer. *Commun. ACM 23*, 4 (Apr), 214–229.

CALLAHAN, P. B., AND KOSARAJU, S. R. 1992. A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. In *Proceedings of the 24th Annual Symposium on the Theory of Computing* (Victoria, B. C., Canada, May 4–6). ACM, New York, pp. 546–556.

COLE, R., AND GOODRICH, M. T. 1988. Optimal parallel algorithms for polygon and point-set problems. In *Proceedings of the 4th ACM Symposium on Computational Geometry* (Urbana-Champaign, Ill., June 6–8). ACM, New York, pp. 201–210.

COLE, R., AND GOODRICH, M. T. 1992. Optimal parallel algorithms for point-set and polygon problems. *Algorithmica 7*, 3–23.

FRIEZE, A. M., MILLER, G. L., AND TENG, S.-H. 1992. Separator based parallel divide and conquer in computational geometry. In *Proceedings of the 4th Annual Symposium Parallel on Algorithms and Architectures* (San Diego, Calif., June 29–July 1). ACM, New York, pp. 420–429.

GAZIT, H., MILLER, G. L., AND TENG, S.-H. 1988. Optimal tree contraction in the EREW model. In *Concurrent Computations*, S. K. Tewsburg, B. W. Dickinson, and S. C. Schwartz, eds. Plenum Publishing, New York, pp. 139–155.

GREENGARD, L. F. 1988. *The Rapid Evaluation of Potential Fields in Particle Systems*. The MIT Press, Cambridge, Mass.

GREENGARD, L., AND GROPP, W. D. 1990. A parallel version of the fast multipole method. *Computers Math. Applic. 20*, 7, 63–71.

GREENGARD, L., AND ROKHLIN, V. 1987. A fast algorithm for particle simulations, *J. Comput. Physics 73*, 325–348.

HAFNER, C., AND KUSTER, N. 1991. Computations of electromagnetic fields by the multiple multipole method (generalized multipole technique). *Radio Sci. 26*, 1, 291–297.

KATZENELSON, J. 1989. Computational structure of the N-body problem. *SIAM J. Sci. Stat. Comput. 10*, 4 (July), 787–815.

KOSARAJU, S. R., AND DELCHER, A. L. 1988. Optimal parallel evaluation of tree-structured computations by raking. In *VLSI Algorithms and Architectures: Proceedings of the 3rd Aegean Workshop on Computing*, J. Reif, ed. Springer-Verlag, New York, pp. 101–110.

MILLER, G. L., TENG, S.-H. AND VAVASIS, S. A. 1991. A unified geometric approach to graph separators. In *Proceedings of the 32nd Annual Symposium on Foundations Computer Science*. IEEE, New York, pp. 538–547.

MILLER, R. H., AND PRENDERGAST, K. H. 1968. Stellar dynamics in a discrete phase space. *Astrophys. J. 151*, 699–709.

MILLER, R. H., PRENDERGAST, K. H. AND QUIRK, W. J. 1970. Numerical experiments on spiral structure. *Astrophys. J. 161*, 903–916.

PAN, V. Y., REIF, J. H., AND TATE, S. R. 1992. The power of combining the techniques of algebraic and numerical computing: Improved approximate multipoint polynomial evaluation and improved multipole algorithms. In *Proceedings of the 32nd Annual Symposium of Foundations of Computer Science*. IEEE New York, pp. 703–713.

PREPARATA, F. P., AND SHAMOS, M. I. 1985. *Computational Geometry. An Introduction*. Springer-Verlag, New York.

REIF, J. H., AND TATE, S. R. 1992. N-body simulation I: Potential field evaluation. Technical report, Comput. Sci. Dept., Duke Univ.

ROKHLIN, V. 1985. Rapid solution of integral equations of classical potential theory. *J. Computat. Phys. 60*, 187–207.

SCHMIDT, K. E., AND LEE, M. A. 1991. Implementing the fast multipole method in three dimensions. *J. Stat. Phys. 63* 5–6 (June), 1223–1235.

VAIDYA, P. M. 1986. An optimal algorithm for the all-nearest-neighbors problem. In *Proceedings of the 27th Annual Symposium Foundations of Computer Science*. IEEE, New York, pp. 117–122.

VAIDYA, P. M. 1989. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Disc. Computat. Geom. 4*, 101–115.

YAO, A. C. 1982. On constructing minimum spanning trees in k-dimensional space and related problems. *SIAM J. Comput. 11*, 721–736.

ZHAO, F., AND JOHNSSON, S. L. 1991. The parallel multipole method on the Connection Machine. *SIAM J. Sci. Stat. Comput. 12*, 6, 1420–1437.