Optimal Parallel All-Nearest-Neighbors Using the Well-Separated Pair Decomposition

(Preliminary Version)

Paul B. Callahan *

Abstract

We present an optimal parallel algorithm to construct the well-separated pair decomposition of a point set P in $\Re^d$. We show how this leads to a deterministic optimal $O(\log n)$ time parallel algorithm for finding the k-nearest-neighbors of each point in P, where k is a constant. We discuss several additional applications of the well-separated pair decomposition for which we can derive faster parallel algorithms.

1 Introduction

In [4] we introduced the well-separated pair decomposition of a set P of n points in $\Re^d$, and showed how to apply this decomposition to develop efficient parallel algorithms for two problems posed on multi-dimensional point sets. One of these applications led to the fastest known deterministic parallel algorithm for finding the k-nearest-neighbors of each point in P using $O(n)$ processors. The time required for this algorithm is $\Theta(\log^2 n)$, which is within a $\log n$ factor of optimal.

In this paper, we close the gap by developing an optimal $O(\log n)$ time parallel algorithm for computing the well-separated pair decomposition. We have already shown that the k-nearest-neighbors problem can be solved in $O(\log n)$ time given the decomposition, so this result leads directly to an optimal all-nearest-neighbors algorithm.

The k-nearest-neighbors problem has received a great deal of attention [3, 7, 9, 11, 15, 17]. The first optimal sequential algorithm was presented by Vaidya [17]. More recently, Frieze, Miller, and Teng [11] have developed an optimal randomized parallel algorithm using substantially different techniques. Their computational model assumes that prefix sum can be performed in $O(1)$ steps. The algorithm we present here is, to our knowledge, the first optimal deterministic parallel algorithm for solving all-nearest-neighbors, and requires $O(\log n)$ time with $O(n)$ processors on a standard CREW PRAM.

We have developed several additional applications of the well-separated pair decomposition [4, 6] and we feel it is of general utility in multi-dimensional proximity problems. Hence, we believe our results will be useful in the development of optimal parallel algorithms for many other problems posed on point sets in $\Re^d$.

The well-separated pair decomposition of P consists of a binary tree whose leaves are points in P, and a list of pairs of nodes satisfying certain properties we will define later. In section 3, we present an improved parallel algorithm for constructing the tree. This algorithm requires several non-trivial techniques, and will constitute the largest portion of the paper. In section 4, we present an improved parallel algorithm for finding the pairs. In section 5, we show how our improved parallel algorithm can be used to derive fast parallel algorithms for several of the applications we have considered so far, including k-nearest-neighbors.

2 Definitions

Let P be a set of points in $\Re^d$, where d is a constant denoting the dimension. We define the *bounding rectangle* of P, denoted by $\mathbf{R}(P)$, to be the smallest rectangle that encloses all points in P, where the word "rectangle" denotes the cartesian product $\mathbf{R} = [x_1, x_1'] \times [x_2, x_2'] \times \cdots \times [x_d, x_d']$ in $\Re^d$.

We denote the length of $\mathbf{R}$ in the ith direction by $l_i(\mathbf{R}) = x_i' - x_i$. We denote the maximum and minimum lengths by $l_{\max}(\mathbf{R})$ and $l_{\min}(\mathbf{R})$. When all $l_i(\mathbf{R})$ are equal, we say that $\mathbf{R}$ is a *d-cube*, and denote its length by $l(\mathbf{R}) = l_{\max}(\mathbf{R}) = l_{\min}(\mathbf{R})$. We write $l_i(P)$, $l_{\min}(P)$, and $l_{\max}(P)$ as shorthand for $l_i(\mathbf{R}(P))$, $l_{\min}(\mathbf{R}(P))$, and $l_{\max}(\mathbf{R}(P))$, respectively.

We say that point sets A and B are *well-separated* iff $\mathbf{R}(A)$ and $\mathbf{R}(B)$ can each be contained in d-spheres of radius r whose distance of closest approach is at least sr, where s is the *separation*, assumed to be fixed throughout our discussion at a value strictly greater

*Computer Science Department, Johns Hopkins University, Baltimore, MD 21218. Supported by the National Science Foundation under Grant CCR-9107293.

than 0. While we can define this notion in a somewhat 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 labelled by a singleton set containing one of the points in P, and each internal node is labelled by the union of all sets labelling the leaves of its subtree. We will refer to each node in T by the name of the set labelling it. Since leaves are labelled 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:

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

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

A set $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ of pairs of non-empty subsets of A and B is said to be a *realization* of $A \otimes B$ iff

i. $A_i \cap B_i = \emptyset$ for all $i = 1, \ldots, k$.

ii. $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$ for all i, j such that $1 \leq i < j \leq k$.

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

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

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

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

3 Constructing the fair split tree

The tree we normally associate with the point set P will be of a special kind, which we call a fair split tree. To formalize this notion, we introduce some additional definitions.

We define a *split* of P to be its partition into two non-empty 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 $\hat{\mathbf{R}}(A)$, for each node A top down as follows:

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

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

We have shown [4] that given a fair split tree T, we can always construct a realization of $P \otimes P$ containing $O(n)$ pairs. We have presented a $\Theta(n \log n)$ algorithm for constructing such a tree, which we have proven optimal, and an $O(\log^2 n)$ time parallel algorithm. We now consider the problem of constructing such a tree optimally in $O(\log n)$ time in parallel.

We use the technique of *multiway divide-and-conquer* [2], in which a problem of size n is split into a set of subproblems, each of size $O(n^\alpha)$ for some $\alpha < 1$. This approach cannot be applied in an obvious fashion, since we must preserve the fair-split condition during the "divide" phase of our algorithm. For this reason, we develop a non-trivial splitting phase that constructs a partial fair split tree whose leaves are rectangles, each containing $O(n^\alpha)$ points.

We begin by splitting our point set into slabs in each dimension, using $\lceil n^{1-\alpha} \rceil + 1$ axis-parallel hyperplanes, spaced so that each slab contains at most n^α points. We will refer to these hyperplanes as *slab boundaries*. Recall that the definition of a fair split allows some freedom in the choice of the split hyperplane. In the present algorithm, we will use the slab boundaries to guide our choice of splits, thus limiting the kinds of rectangles that appear as nodes in the tree. In particular, we will be able to represent each side of a rectangle by a constant-size arithmetic expression in terms of slab boundaries and small integers. We will

later show how to choose α to substantially limit the number of constructable rectangles. This will be essential in maintaining a reasonable processor bound.

Given a rectangle for which certain properties hold, we will develop a way of splitting it that results in new rectangles that preserve these properties and are constructable in the sense used above. In certain cases, we will not be able to split the rectangle, and must instead "shrink" it to a much smaller rectangle sharing a corner with the original and containing all but $O(n^\alpha)$ of the points in the original. This will lead to a special compressed edge in the resulting tree. After we have finished constructing the tree, we will expand each compressed edge into a chain of splits using a technique developed in our previous parallel algorithm [4]. Finally, we will remove any rectangles that do not contain points, thus completing the splitting phase.

Let $\mathbf{R}$ be a rectangle that we wish to split. For the remainder of this discussion, let $\mathbf{R}'$ denote the largest rectangle that is contained in $\mathbf{R}$ and whose sides lie on slab boundaries. We will maintain the following invariants with respect to the rectangles we consider.

1. In each dimension, at least one side of $\mathbf{R}$ lies on a slab boundary.
2. For all $i = 1, \ldots, d$, either $l_i(\mathbf{R}') = l_i(\mathbf{R})$ or $l_i(\mathbf{R}') \leq \frac{2}{3} l_i(\mathbf{R})$
3. $l_{\min}(\mathbf{R}) \geq \frac{1}{3} l_{\max}(\mathbf{R})$.

We now consider the three cases of our split rule. For the following, we use $i_{\max}$ to denote the longest dimension of $\mathbf{R}$. The application of our split rule will be said to *produce* one or two new rectangles that must preserve the above invariants. In our illustrations, slab boundaries are shown with thin dotted lines, while each rectangle and its corresponding split are shown with thick solid lines.

Case 1: $l_{\max}(\mathbf{R}) = l_{i_{\max}}(\mathbf{R}')$ (see figure 1). In this case, both sides of $\mathbf{R}$ in the $i_{\max}$ dimension lie on slab boundaries. We find the slab boundary in this dimension that comes closest to splitting $\mathbf{R}$ into two equal halves. If (a) it lies at a distance of at least $\frac{1}{3} l_{\max}(\mathbf{R})$ from the closest side of $\mathbf{R}$ in the $i_{\max}$ dimension, we use it for our split. Otherwise (b), we split $\mathbf{R}$ into two equal halves in the $i_{\max}$ dimension. This case produces two rectangles, each lying on one side of the split.

Note that if case 1 does not hold, then by the second invariant,

$$l_{i_{\max}}(\mathbf{R}') \leq \frac{2}{3} l_{\max}(\mathbf{R}) \qquad (1)$$

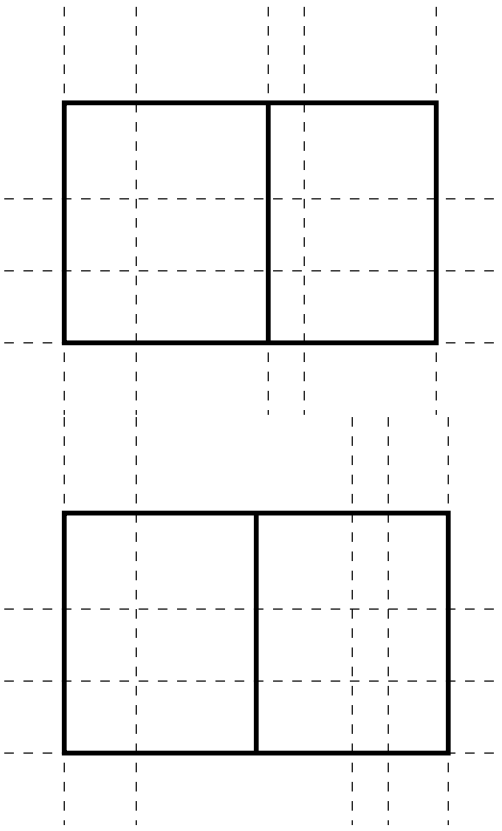

Figure 1: Cases 1 (a) and (b), shown left to right.

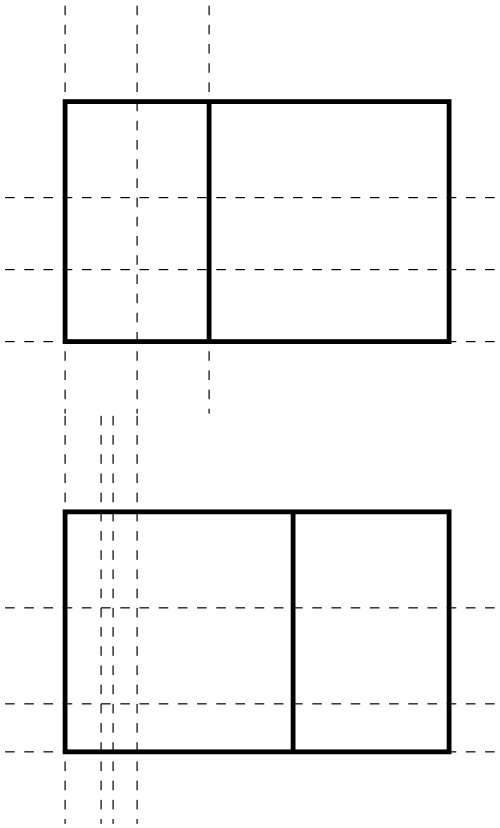

Figure 2: Cases 2 (a) and (b), shown left to right.

We will assume this inequality in the remaining cases. In the following, c is a constant whose value we will determine later.

Case 2: $l_{\max}(\mathbf{R}') \geq c l_{\max}(\mathbf{R})$ (see figure 2). Exactly one of the sides of $\mathbf{R}$ in the $i_{\max}$ dimension lies on a slab boundary, which we denote by H. If (a) $l_{i_{\max}}(\mathbf{R}') \geq \frac{1}{3} l_{\max}(\mathbf{R})$, we use the hyperplane containing the side of $\mathbf{R}'$ farthest from H for our split. Otherwise (b), we must choose another split which preserves our invariants. It is sufficient to use a hyperplane splitting $\mathbf{R}$ at a distance from H in the interval $(\frac{1}{2}, \frac{2}{3}] l_{\max}(\mathbf{R})$. We find the unique integer j such that $\frac{2}{3}\left(\frac{4}{3}\right)^j l_{\max}(\mathbf{R}')$ lies in the interval, and use a hyperplane at this distance. This case produces only one rectangle, namely the one containing $\mathbf{R}'$.

Case 3: $l_{\max}(\mathbf{R}') < c l_{\max}(\mathbf{R})$ (see figure 3). For sufficiently small values of c, exactly one side of $\mathbf{R}$ lies on a slab boundary in each dimension. In other words, $\mathbf{R}'$ shares a unique corner with $\mathbf{R}$. We construct a d-cube $\mathbf{C}$ containing $\mathbf{R}'$ such that $l(\mathbf{C}) = \frac{3}{2} l_{\max}(\mathbf{R}')$ and $\mathbf{C}$ shares the same corner with $\mathbf{R}$ as $\mathbf{R}'$. We will later show how to find a sequence of fair splits decomposing the region between $\mathbf{R}$ and $\mathbf{C}$ into rectangles that contain $O(n^\alpha)$ points. This case produces the rectangle $\mathbf{C}$.

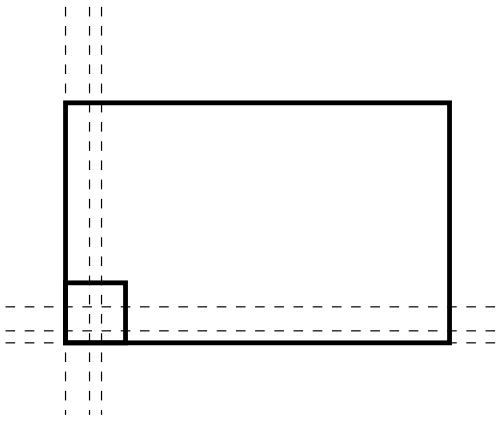

Figure 3: Case 3

We must prove that the above cases preserve the invariants. The first invariant follows immediately from the fact that we never split $\mathbf{R}'$ unless both sides parallel to the split lie on slab boundaries. The third invariant follows from the fair split condition. That is, after the split, the new minimum length cannot be less than $\frac{1}{3} l_{\max}(\mathbf{R})$, while the new maximum length cannot be more than $l_{\max}(\mathbf{R})$. The second invariant is somewhat more complicated to verify. If we split on a slab boundary, the invariant is clearly preserved. Otherwise, all slab boundaries passing through $\mathbf{R}$ in the $i_{\max}$ dimension must be at a distance of less than $\frac{1}{3} l_{\max}(\mathbf{R})$ from the sides parallel to them. Given this fact, it is not hard to see that our choice of a split hyperplane in cases 1 and 2 preserves the invariant. It is even easier to verify the invariant in case 3.

In the above cases, the new rectangles produced by a split can always be encoded in terms of a constant-size list of slab boundaries and O(1) bits of extra information. We will consider case 2(b) in some detail, because it is the most complex and allows the greatest number of new rectangles to be defined.

Suppose some dimension of a rectangle has been generated by an invocation of case 2(b). To fully specify this dimension, we must supply a slab boundary for one side of the rectangle, and a distance of the form $2\left(\frac{4}{3}\right)^j l_{\max}(\mathbf{R}')$, from which we may compute the position of the other side. We may represent this distance by the two slab boundaries that determine $l_{\max}(\mathbf{R}')$, and an integer j.

We consider the possible values for j. Recall that the distance we represent must lie in the interval $(\frac{1}{2}, \frac{2}{3}] l_{\max}(\mathbf{R})$. We know that $c l_{\max}(\mathbf{R}) \leq l_{\max}(\mathbf{R}') <$

$l_{\max}(\mathbf{R})$, and the right-hand side immediately gives us $j \geq 0$. From the left-hand side, one may easily verify that $j \leq -\lfloor \log c / \log \frac{4}{3} \rfloor$. Since c is a constant, there are O(1) choices of j.

It is easy to see that rectangle dimensions generated by other split rules can be treated in a similar manner. An upper bound on the number of rectangles, valid to within a constant factor, may be obtained by considering only those rectangles whose dimensions come from case 2(b). There are $O(n^{1-\alpha})$ slab boundaries, and at most $3d$ slab boundaries are required to represent a rectangle. This gives us the following result.

Lemma 1 *There are* $O(n^{3d(1-\alpha)})$ *constructable rectangles.* □

Given a starting rectangle $\mathbf{R}_0$, we would like to construct a tree in which $\mathbf{R}_0$ is the root, and each internal node has one or two children according to the split rule given previously. A rectangle intersected by exactly two slab boundaries in the $i_{\max}$ dimension will be considered a leaf and will have no children.

$\mathbf{R}_0$ will normally be inherited from a previous phase in the algorithm, so the third invariant will be guaranteed automatically (initially, we use some d-cube that contains the entire point set). To guarantee the first and second invariants, we set the extreme slab boundaries so that they contain the sides of the starting rectangle.

Consider the directed graph in which each node corresponds to a constructable rectangle, and arcs are placed from each node that is not a leaf to any node it can produce by an invocation of the split rule. The tree we wish to construct will consist of all nodes accessible from $\mathbf{R}_0$, along with their out-going arcs. Let T_c denote this tree.

The first stage in tree construction will be to generate the set of constructable rectangles and compute their out-going arcs. We use G to denote the graph that results. For convenience, we may refer to nodes and rectangles interchangeably. It is hard to see how to construct T_c without finding a transitive closure of G, but the most general method may take too much time. Fortunately, the structure of G is very restricted.

In particular, we have the following property: Let u and v be two distinct nodes in G. If there exists a directed path from u to v, then this path is unique. In fact, given u and some v accessible from u, we can find the first arc along the path in constant time using the following observation:

Observation 1 *If there is a directed path from* u *to* v *in* G*, then the first arc in the path is the unique arc* (u, w)*, such that* (u, w) *is an arc in* G *and the rectangle* w *contains the rectangle* v*.*

Consider the graph G', which contains a node $[u, v]$ for each pair of nodes in G. Let there be an arc from $[u, v]$ to $[w, v]$ iff (u, w) is an arc in G and v is contained in w. Nodes in G' have outdegree of at most 1 by the observation that (u, w) is unique. Hence, G' is a forest of rooted trees in which each node that is not a root has an arc leading to its parent. Each node of the form $[v, v]$ is the root of some tree, and clearly v is accessible from u in G iff $[u, v]$ is a node in this tree.

Recall that for each v, we are ultimately interested in determining whether it is accessible from $\mathbf{R}_0$ in G. In other words, we wish to know whether the tree rooted at $[v, v]$ contains the node $[\mathbf{R}_0, v]$. With this observation, we are ready to present the complete algorithm for computing T_c:

1. Construct slabs in each dimension.
2. Construct G from slab boundaries.
3. Construct G' from G.
4. Use G' to find v in G accessible from $\mathbf{R}_0$.

Finally, T_c consists of those v accessible from $\mathbf{R}_0$, together with their out-going arcs.

Correctness of the above algorithm follows from the preceding discussion. We now consider the parallel complexity of each step.

Step 1 can be solved in the time it takes to sort the points in each dimension. This can be done with n processors in $O(\log n)$ time on an EREW PRAM [8].

Step 2 can be solved by assigning a processor to each possible formula for constructing a rectangle, so there are $O(|G|)$ processors. Each processor can find an explicit value for its rectangle in constant time, and compute its out-going arcs in $O(\log n)$ time using a binary search on a sorted list of slab boundaries in the appropriate dimension.

Step 3 can be solved in constant time once we have assigned a processor to every ordered pair of nodes in G. Each processor finds the out-going arc, if it exists, of the corresponding node in G'.

Step 4 can be solved in several ways. Our approach is as follows. Assign a weight of 1 to all nodes in G' of the form $[\mathbf{R}_0, v]$ and a weight of 0 to all other nodes. Now compute, for every node in G', the total weight of its subtree. The computed total for $[v, v]$ will be 1 iff $[\mathbf{R}_0, v]$ is a node in its subtree, or equivalently, v is accessible from $\mathbf{R}_0$ in G. This can be solved in $O(\log |G|)$ time with $O(|G|^2 / \log |G|)$ processors on an EREW PRAM using the Euler tour technique [16].

According to Lemma 1, $|G| = O(n^{3d(1-\alpha)})$. Hence, if we choose some $\alpha \geq 1 - \frac{1}{6d}$, then $|G|^2 = O(n)$. Therefore, the above algorithm requires $O(\log n)$ time with $O(n)$ processors.

We must further process T_c by expanding each edge produced by case 3 into a chain of splits. In presenting this part of our algorithm, we will derive the value of the constant c used in distinguishing cases 2 and 3.

Let $\mathbf{R}$ be a rectangle in T_c with child $\mathbf{C}$ produced by case 3. Construct the unique cube $\mathbf{C}'$ such that $l(\mathbf{C}') = \frac{2}{3} l_{\min}(\mathbf{R})$, and $\mathbf{C}'$ shares a corner with both $\mathbf{C}$ and $\mathbf{R}$. Find a sequence of splits from $\mathbf{R}$ to $\mathbf{C}'$ using a rule similar to case 1 in the preceding algorithm. Note that $\mathbf{C}'$ is sufficiently small to guarantee that such a sequence is possible, and sufficiently large to guarantee that only O(1) splits are necessary. Hence, this can be done in constant time with one processor per rectangle $\mathbf{R}$.

Now we need only find a sequence of fair splits between $\mathbf{C}'$ and $\mathbf{C}$. Let m be the number of points inside $\mathbf{C}'$ and outside $\mathbf{C}$. We have shown [4] how to find such a sequence of fair splits in $O(\log m)$ time with $O(m)$ processors. As a precondition, we require that $l(\mathbf{C}') \geq \frac{3}{2} l(\mathbf{C})$. We must choose c consistent with this constraint. Recalling that $l(\mathbf{C}) = \frac{3}{2} l_{\max}(\mathbf{R}')$, we obtain the equivalent constraint $l_{\min}(\mathbf{R}) \geq \frac{27}{8} l_{\max}(\mathbf{R}')$. By the third invariant in our split rule, $l_{\min}(\mathbf{R}) \geq \frac{1}{3} l_{\max}(\mathbf{R})$. Hence, if we let $c = \frac{81}{8}$, then the precondition on case 3 guarantees that the constraint is satisfied. If we expand all the compressed edges concurrently, then the total number of processors assigned cannot exceed $O(n)$, and the time will be $O(\log n)$.

Until this point, we have neglected the fact that case 2 actually resulted in two rectangles. It was convenient to say it produced only one because the other violated the first invariant. To correct this problem, we treat the other as a leaf. We insert all such leaves in constant time by assigning a processor to every rectangle $\mathbf{R}$ with a child produced by case 2.

To complete the splitting phase, we compress out any empty rectangles in the tree. This can be done as follows: Weight each leaf with the number of points contained in its rectangle. For each internal node, compute the total weight of the leaves in its subtree, and weight it with this value. Eliminate all nodes with weight zero, and compress non-branching paths to single edges. It is easy to verify that the tree has size $O(n)$ before compression. Hence, all these steps can be done in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM using standard techniques.

We define a *partial fair split tree* to be a tree that satisfies all the conditions of a fair split tree, except that its leaves need not be singleton point sets. We use T_p to denote the tree ultimately constructed by the previous algorithm, and we wish to show that T_p is a partial fair split tree.

Note that after the final compression, every leaf of T_p contains a non-empty point set, and every non-leaf node containing the point set P has two children formed by a split of P. It suffices to prove that such splits are fair.

We may focus our attention on those splits produced by cases 1 and 2 of the first part of the algorithm. Every node that is split in such a manner is labeled by a rectangle $\mathbf{R}$, and it may be easily verified that the split is placed at a distance of at least $\frac{1}{3} l_{\max}(\mathbf{R})$ from the nearest boundary of $\mathbf{R}$ parallel to the split. Because $\mathbf{R}$ contains P, $l_{\max}(\mathbf{R}) \geq l_{\max}(P)$. Moreover, the outer rectangle $\hat{\mathbf{R}}(P)$ contains $\mathbf{R}$, so the nearest boundary of $\hat{\mathbf{R}}(P)$ is no closer to the split than the nearest boundary of $\mathbf{R}$. Thus it follows that each such split is fair, and T_p is a partial fair split tree.

To prove the correctness of the splitting phase, we must finally argue that each leaf of T_p contains $O(n^\alpha)$ points. It suffices to show that the outer rectangle of every leaf lies entirely within two adjacent slabs in some dimension, since each slab contains n^α points. This can be verified by considering all the termination conditions that result in leaves. Note that the precise sequence of splits used in expanding case 3 edges is not important, because any rectangle in the region inside $\mathbf{R}$ and outside $\mathbf{C}$ must lie in two adjacent slabs. Other cases are even easier to verify.

From the above discussion, we have the following lemma:

Lemma 2 *Let P be a set of n points in $\Re^d$, and let α be a real number between $1 - \frac{1}{6d}$ and 1. In $O(\log n)$ time with $O(n)$ processors on an EREW PRAM, we can construct a partial fair split tree T_p of P such that each leaf of T_p contains $O(n^\alpha)$ points.* □

The above result leads directly to the following theorem.

Theorem 1 *Let P be a set of n points in $\Re^d$. We can find a fair split tree of P in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM.*

Proof. Our algorithm is as follows: Let α be some constant greater than or equal to $1 - \frac{1}{6d}$ and less than 1 (the precise value may be chosen to optimize the constants in the complexity). For n less than some constant n_0, solve the problem by brute force. Otherwise, construct a partial fair split tree T_p, whose leaves are point sets of size $O(n^\alpha)$. Find fair split trees for these point sets by applying the algorithm recursively in parallel.

The point sets at the leaves form a partition of P, so the total number of processors remains $O(n)$ at all stages of the recursion. The total time $T(n)$ satisfies the recurrence $T(n) \leq \beta \log n + \gamma T(n^{\alpha})$ for suitable choices of β and γ. From this, it is easily seen that $T(n) = O(\log n)$. □

4 Computing the pairs

Given the point set P, let T denote a fair split tree of P. We consider the problem of computing the well-separated pairs of the nodes of T. The pairs we compute will be the same as those produced by our sequential algorithm [4]. In the previous paper, we showed how to compute these pairs in $O(\log^2 n)$ time with $O(n)$ processors, and in this section, we improve the time bound to $O(\log n)$ using the same number of processors.

Roughly speaking, the sequential algorithm works by taking a pair (A, B) of nodes in T that are not well-separated, forming two pairs by splitting the node with the larger bounding rectangle into its two children, and applying the process recursively until the result is a set of well-separated pairs. The initial pairs are all pairs of siblings in T.

The pairs produced by this algorithm can be characterized as the set of all $\{A, B\}$ such that A and B are nodes in T with the following properties: A and B are well-separated, $p(A)$ and B are not well-separated and $l_{\max}(B) < l_{\max}(p(A)) \leq l_{\max}(p(B))$. For the leftmost inequality, we assume that some consistent method of tie-breaking is used in cases of equal lengths. The correctness of this characterization will be proven in the final version. The idea behind our parallel algorithm is to retrieve the set of pairs $\{A, B\}$ simultaneously for all A. To do this, we develop a way to enumerate all B for any given A in $O(\log n)$ time with one processor.

Consider the problem of retrieving, for an arbitrary query cube $\mathbf{C}$, all of those nodes B in T such that $l_{\min}(\hat{\mathbf{R}}(B)) \leq \delta l(\mathbf{C}) \leq l_{\min}(\hat{\mathbf{R}}(p(B)))$ for some constant δ, and the outer rectangle $\hat{\mathbf{R}}(B)$ overlaps C. Using a packing argument similar to that given in [4], we can show that the number of such B is bounded by a function of δ and d, and is thus a constant.

To enumerate all B paired with a given A, we construct a suitably large cube $\mathbf{C}$ around $p(A)$. We then solve the preceding problem for some δ, dependent upon s, and obtain a set of size $O(1)$ that contains all B paired with A. From this set, the pairs $\{A, B\}$ can be found in constant time by brute force. We now show how to solve the preceding problem in $O(\log n)$ time using one processor.

We preprocess T by constructing its *centroid decomposition* T' [10]. This is a balanced binary tree partition of the nodes in T. The subtree of any node in T' is a tree partition of the nodes in some subtree of T from which at most one node and its subtree have been removed. Because T' is balanced, its depth is $O(\log n)$.

Note that the union of all outer rectangles of nodes in a subtree of T' consists of a rectangle containing at most one rectangular hole (i.e., the outer rectangle of the missing subtree). We assume that each node in T' is labeled with such a geometric region. Given a node in T', we can therefore determine in constant time whether the outer rectangle of one of its descendants overlaps $\mathbf{C}$.

To retrieve the set of B that overlap $\mathbf{C}$ and satisfy the size condition, we begin at the root of T' and only descend into subtrees corresponding to a sufficiently large region that overlaps $\mathbf{C}$. Every node at which we terminate this search process then corresponds to some B. We can determine all B in this manner, since we only skip nodes that are guaranteed not to contain a B in their subtree. The subtree we traverse has a constant number of leaves and a depth no greater than the depth of T', which is $O(\log n)$. Hence, the time to perform the search is $O(\log n)$.

We can find the centroid T' in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM [10, 14]. After constructing T', we assign a processor to each node A in T, and perform the preceding search process for each A simultaneously. This may result in concurrent reads, so we must use the stronger CREW model. Therefore, the complete algorithm requires $O(\log n)$ time with $O(n)$ processors on a CREW PRAM.

Combining this with the result of the previous section, we obtain the following theorem.

Theorem 2 *Let P be a set of n points in $\Re^d$. We can find a well-separated pair decomposition of P in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM.* □

5 Applications

Our primary application is the k-nearest-neighbors problem. We have already shown [4] that given a well separated pair decomposition of a point set P, we can retrieve the k nearest Euclidean neighbors of each point in P for any constant k in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM.

To do this, we parallelize our sequential algorithm using rake and compress [1, 12, 14]. In this algorithm, we compute for each point x, a set of $O(1)$ candidates

that may have x as one of its k-nearest neighbors. This requires $O(\log n)$ time with $O(n/\log n)$ processors. We then extract the set of k-nearest neighbors of each point, in $O(\log n)$ time with $O(n)$ processors. By setting k equal to one, assuming all interpoint distances are distinct, we obtain an optimal parallel algorithm for the all-nearest-neighbors problem. The algorithm can be trivially modified to eliminate the assumption of distinct distances.

Often, we may be interested in the dependence on k in the complexity. In the revised version of our previous paper [5], we presented a sequential algorithm that solved the k-nearest-neighbors problem in $O(n \log n + kn)$ time. This is optimal with respect to dependence on k. The corresponding parallel algorithm requires $O(\log n)$ time with $O(kn)$ processors on a CREW PRAM, and its work bound is therefore within a $\min\{k, \log n\}$ factor of optimal.

Other applications [4] include a parallel version of the Fast Multipole Method [13] for computing potential fields. Given the well-separated pair decomposition of a point set representing a charge distribution, we can implement the Fast Multipole Method in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM. Using our improved parallel algorithm for computing the decomposition, the total time to solve this problem is now $O(\log n)$ with $O(n)$ processors.

In [6], we presented several new applications of the well-separated pair decomposition. The algorithms we presented were sequential, but the basic techniques we employed can be parallelized using rake and compress on the decomposition tree. This leads to $O(\log n)$ time algorithms for finding an ϵ-approximate minimum spanning tree of P as well as an ϵ-approximate complete Euclidean graph. For constant ϵ, the work bounds are optimal.

References

[1] K. Abrahamson, N. Dadoun, D. A. Kirkpatrick, and T. Przytcka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[2] M. J. Atallah and M. T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3:492–507, 1986.

[3] J. L. Bentley. Multidimensional divide-and-conquer. *CACM*, 23(4):214–229, 1980.

[4] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. In *Proceedings 24th Annual AMC Symposium on the Theory of Computing*, pages 546–556, 1992.

[5] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. Revised Version, 1992.

[6] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings 4th Annual Symposium on Discrete Algorithms*, pages 291–300, 1993.

[7] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proceedings 24th Annual Symposium on Foundations of Computer Science*, pages 226–232, 1983.

[8] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770–785, 1988.

[9] R. Cole and M. T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. In *Proceedings 4th ACM Symposium on Computational Geometry*, pages 201–210, 1988.

[10] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[11] A. M. Frieze, G. L. Miller, and S.-H. Teng. Separator based parallel divide and conquer in computational geometry. In *Proceedings 4th Annual Symposium Parallel Algorithms and Architectures*, pages 420–429, 1992.

[12] H. Gazit, G. L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In S. K. Tewsburg, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations*, pages 139–155. Plenum Publishing, 1988.

[13] L. F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. The MIT Press, 1988.

[14] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In J. Reif, editor, *VLSI Algorithms and Architectures: Proceedings of the 3rd Aegean Workshop on Computing*, pages 101–110. Springer-Verlag, 1988.

[15] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings 32nd Annual Symposium Found. Comp. Sc.*, pages 538–547, 1991.

[16] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing*, 14:862–874, 1985.

[17] P. M. Vaidya. An optimal algorithm for the all-nearest-neighbors problem. In *Proceedings 27th Annual Symposium Found. Comp. Sc.*, pages 117–122, 1986.