1987

Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Richard Cole

Michael T. Goodrich

Report Number:

87-665

CASCADING DIVIDE-AND CONQUER: A TECHNIQUE FOR DESIGNING PARALLEL ALGORITHMS

Mikhail J. Atallah
Richard Cole
Michael T. Goodrich

CSD-TR-665
March 1987
(Revised June 1989)

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 499-532, June 1989

CASCADING DIVIDE-AND-CONQUER: A TECHNIQUE FOR DESIGNING PARALLEL ALGORITHMS*

MIKHAIL J. ATALLAH†, RICHARD COLE‡, AND MICHAEL T. GOODRICH§

Abstract. Techniques for parallel divide-and-conquer are presented, resulting in improved parallel algorithms for a number of problems. The problems for which improved algorithms are given include segment intersection detection, trapezoidal decomposition, and planar point location. Efficient parallel algorithms are algo given for fractional cascading, three-dimensional maxima, two-set dominance counting, and visibility from a point. All of the algorithms presented run in $O(\log n)$ time with either a linear or a sublinear number of processors in the CREW PRAM model.

Key words. parallel algorithms, parallel data structures, divide-and-conquer, computational geometry, fractional cascading, visibility, planar point location, trapezoidal decomposition, dominance, intersection detection

AMS(MOS) subject classifications. 68E05, 68C05, 68C25

1. Introduction.

This paper presents a number of general techniques for parallel divide-and-conquer. These techniques are based on nontrivial generalizations of Cole's recent parallel merge sort result [13] and enable us to achieve improved complexity bounds for a large number of problems. In particular, our techniques can be applied to any problem solvable by a divide-and-conquer method such that the subproblem merging step can be implemented using a restricted, but powerful, set of operations, which include (i) merging sorted lists, (ii) computing the values of labeling functions on elements stored in sorted lists, and (iii) changing the identity of elements in a sorted list monotonically. The elements stored in such sorted lists need not belong to a total order, so long as the computation can be specified so that we will never try to compare two incomparable elements. We demonstrate the power of these techniques by using them to design efficient parallel algorithms for solving a number of fundamental problems from computational geometry.

The general framework is one in which we want to design efficient parallel algorithms for the CREW PRAM or EREW PRAM models. Recall that the CREW PRAM model is the synchronous shared memory model in which processors may simultaneously read from any memory location but simultaneous writes are not allowed. The EREW PRAM model does not allow for any simultaneous access to a memory cell. Our goal is to find algorithms that run as fast as possible and are efficient in the following sense: if $p(n)$ is the processor complexity, $t(n)$ the parallel time complexity, and $seq(n)$ the time complexity of the best-known sequential algorithm for the problem

* Received by the editors September 14, 1987; accepted for publication (in revised form) August 12, 1988. This paper appeared in preliminary form as [3] and as portions of [17].

† Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by the Office of Naval Research under grants N00014-84-K-0502 and N00014-86-K-0689, and by the National Science Foundation under grant DCR-84-51393, with matching funds from AT&T.

‡ Courant Institute, New York University, New York, New York 10012. The research of this author was supported in part by National Science Foundation grants DCR-84-01633 and CCR-8702271, and by an Office of Naval Research grant N00014-85-K-0046.

§ Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218. The research of this author was supported by Office of Naval Research grants N00014-84-K-0502 and N000-86-K-0689, National Science Foundation grants DCR-84-51393, with matching funds from AT&T, and CCR-88-10568, and a David Ross grant from the Purdue Research Foundation.

under consideration, then $t(n) * p(n) = O(seq(n))$. If the product $t(n) * p(n)$ achieves the sequential lower bound for the problem, then we say the algorithm is *optimal*. When specifying the processor complexity, we omit the "big oh," e.g., we say "n processors" rather than "$O(n)$ processors"; this is justified because we can always save a constant factor in the number of processors at a cost of the same constant factor in the running time. In all of the problems listed below, we achieve $t(n) = O(\log n)$ and, simultaneously (except for planar point location), an optimal $t(n) * p(n)$.

Previous work on parallel divide-and-conquer has produced relatively few algorithms that are optimal in the above sense. Exceptions to this include some of the previous algorithms for the convex hull problem [1], [4], [6], [18], [27] and the problem of circumscribing a convex polygon with a minimum-area triangle [1]. Unfortunately, each of these approaches was very problem-specific. Thus, there is a need for techniques of wider scope.

This is in fact the motivation for our work, for we give a number of general techniques for efficiently solving problems in parallel by divide-and-conquer. We model the divide-and-conquer paradigm as a binary tree whose nodes contain sorted lists of some kind. The computation involves computing on this tree in a recursively defined bottom-up fashion using lists of items and labeling functions defined for each node in the tree. In Cole's scheme [13], the list at a node was defined to be the sorted merge of the two lists stored at its children. In our scheme, however, the lists at a node of the tree can depend on the lists of its children in more complex ways. For example, in our solution to the *segment intersection detection* problem, the lists at a node depend on computing, in addition to merges, set difference operations that are not directly solvable by the "cascading" method used by Cole [13]. Such operations arise here because the lists at a node contain segments ordered by their intersections with a vertical line (the so-called "above" relationship), which is obviously not a total order. One may be tempted to try to solve this problem by delaying the performance of these set difference operations until the end of the computation. Unfortunately, this is not feasible for many reasons, not the least of which is that this approach could lead to a situation in which a processor tries to compare two incomparable items. Nor does it seem possible to explicitly perform the set difference operations on-line without sacrificing the time-efficiency of the cascading method. Our solution avoids both of these problems by using an on-line "identity-changing" technique.

Another significant contribution of this paper is an optimal parallel construction of the "fractional cascading" data structure of Chazelle and Guibas [12]. This too is based on a generalization of Cole's method [13] in the sense that instead of having the computation proceeding up and down a tree, it now moves around a directed graph (possibly with cycles). Our solution to fractional cascading is quite different from the sequential method of Chazelle and Guibas (their method relies on an amortization scheme to achieve a linear running time).

The following is a list of the problems for which our techniques result in improved complexity bounds. Unless otherwise specified, each performance bound is expressed as a pair $(t(n), p(n))$, where $t(n)$ and $p(n)$ are the time and processor complexities, respectively, in the CREW PRAM model.

Fractional cascading. Given a directed graph $G = (V, E)$, such that every node v contains a sorted list $C(v)$, construct a data structure that, given a walk $(v_1, v_2, \cdots, v_m)$ in G and an arbitrary element x, enables a single processor to locate x quickly in each $C(v_i)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. In [12] Chazelle and Guibas gave an elegant $O(n)$ time, $O(n)$ space, sequential construction, where $n = \sum_{v \in V} |C(v)|$. We give a $(\log n, n/\log n)$ construction.

Trapezoidal decomposition. Given a set S of n line segments in the plane, determine for each segment endpoint p the first segment "stabbed" by the vertical ray emanating upward (and downward) from p. A $(\log^2 n, n)$ solution to this problem was given by Aggarwal et al. in [1], later improved to $(\log n \log\log n, n)$ by Atallah and Goodrich in [5]. We improve this to $(\log n, n)$.

Planar point location. Given a subdivision of the plane into (possibly unbounded) polygons, construct, in parallel, a data structure that, once built, enables one processor to determine for any query point p the polygonal face containing p. Let $Q(n)$ denote the time for performing such a query, where n is the number of edge segments in the subdivision. A $(\log^2 n, n)$, $Q(n)=O(\log^2 n)$ solution was given by Aggarwal et al. in [1], later improved to $(\log n \log\log n, n)$, $Q(n)=O(\log n)$ by Atallah and Goodrich in [5]. In [14] Dadoun and Kirkpatrick further improved this to $(\log n \log^* n, n)$, $Q(n)=O(\log n)$. We give a $(\log n, n)$, $Q(n)=O(\log n)$ solution.

Segment intersection detection. Given a set S of n line segments in the plane, determine if any two segments in S intersect. A $(\log^2 n, n)$ solution was given in [1], later improved to $(\log n \log\log n, n)$ in [5]. We improve this to $(\log n, n)$.

Three-dimensional maxima. Given a set S of n points in three-dimensional space, determine which points are maxima. A *maximum* in S is any point p such that no other point of S has x, y, and z coordinates that simultaneously exceed the corresponding coordinates of p. A $(\log n \log\log n, n)$ solution was given in [5]. We improve this to $(\log n, n)$.

Two-set dominance counting. Given a set $A=\{q_1, q_2, \cdots, q_l\}$ and a set $B=\{r_1, r_2, \cdots, r_m\}$ of points in the plane, determine for each point r_i in B the number of points in A whose x and y coordinates are both less than the corresponding coordinates of r_i. The problem size is $n=l+m$. A $(\log n \log\log n, n)$ solution was given in [5]. We improve this to $(\log n, n)$.

Visibility from a point. Given n line segments such that no two intersect (except possibly at endpoints) and a point p, determine that part of the plane visible from p if all the segments are opaque. A $(\log n \log\log n, n)$ solution was given in [5]. We improve this to $(\log n, n)$.

We recently learned that Reif and Sen [24] solved planar point location, trapezoidal decomposition, segment intersection and visibility in randomized $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. All of our algorithms are deterministic.

This paper is organized as follows. In § 2 we present a generalized version of the cascading merge procedure and in § 3 we give our method for doing fractional cascading in parallel. In § 4 we show how to apply the fractional cascading technique to a data structure we call the *plane sweep tree*, showing how to solve the trapezoidal decomposition and point location problems. In § 5 we show how to extend the cascading merge technique to allow for cascading in the "above" partial order of line segments, giving solutions to the problems of building the plane sweep tree and solving the intersection detection problem. In § 6 we use the cascading divide-and-conquer technique to compute labeling functions and show how to use this approach to solve three-dimensional maxima, two-set dominance counting, and visibility from a point. Finally, in § 7, we briefly describe how most of our algorithms can be implemented in the EREW PRAM model with the same time and processor bounds as our CREW PRAM algorithms, and we conclude in § 8.

2. A generalized cascading merge procedure.

In this section we present a technique for a generalized version of the merge sorting problem. Suppose we are given a binary tree T (not necessarily complete) with items, taken from some total order, placed at the leaves of T, so that each leaf contains at most one item. For simplicity, we assume that the items are distinct. We wish to compute for each internal node $v \in T$ the sorted list $U(v)$ that consists of all the items stored in descendant nodes of v. (See Fig. 1.) In this section we show how to constrict $U(v)$ for every node in the tree in $O(height\ (T))$ time using $|T|$ processors, where $|T|$ denotes the number of nodes in T. This is a generalization of the problem studied by Cole [13], because in his version the tree T is complete. Without loss of generality, we assume that every internal node v of T has two children. For if v has only one child then we can add a child to v (a leaf node) that does not store any items from the total order. Such an augmentation will at most double the size of T and does not change its height.

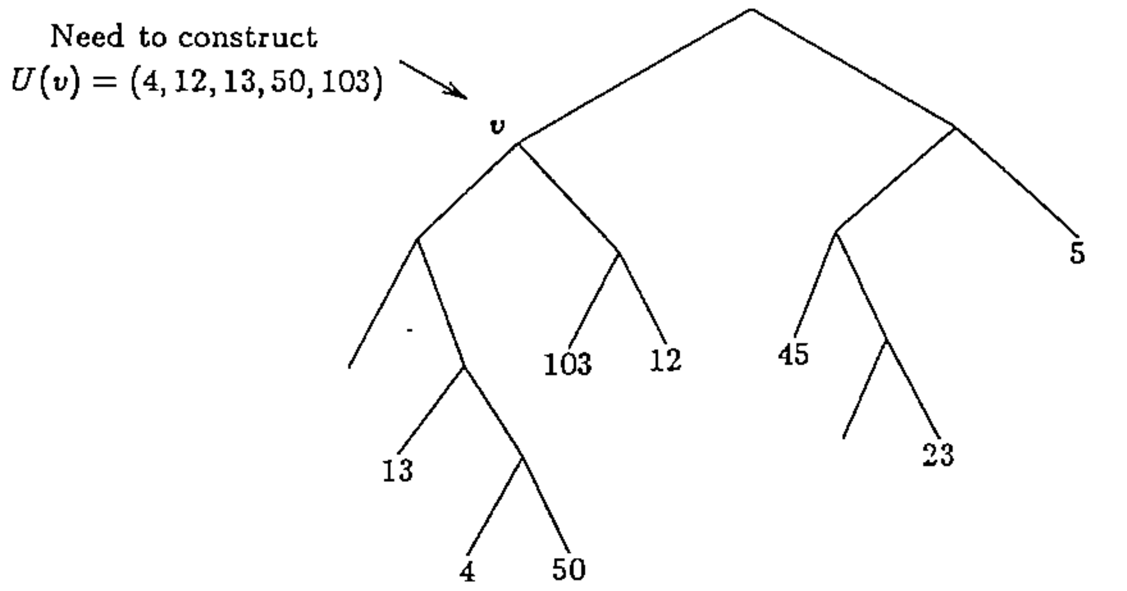


FIG. 1. *An example of the generalized merge problem.*

Let a, b, and c be three items, with $a \leqq b$. We say c is *between* a and b if $a < c \leqq b$. Let two sorted (nondecreasing) lists $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_m)$ be given. Given an element a, we define the *predecessor* of a in B to be the greatest element in B that is less than or equal to a. If $a < b_1$, then we say that the predecessor of a is $-\infty$. We define the *rank* of a in B to be the rank of the predecessor of a in B ($-\infty$ has rank zero). We say that A is *ranked* in B if for every element in A we know its rank in B. We say that A and B are *cross-ranked* if A is ranked in B and B is ranked in A. We define two operations on sorted lists. We define $A \cup B$ to be the sorted *merged* list of all elements in A or B. If B is a subset of A, then we define $A - B$ to be the sorted list of the elemented in A that are not in B.

Let T be a binary tree. For any node v in T we let $parent(v)$, $sibling(v)$, $lchild\ (v)$, $rchild\ (v)$, and $depth(v)$ denote the parent of v, the sibling of v, the left child of v, the right child of v, and the depth of v (the root is at depth zero), respectively. We also let $root(T)$ and $height(T)$ denote the root node of T and the height of T, respectively. The *altitude*, denoted $alt(v)$, is defined $alt(v) = height(T) - depth(v)$. $Desc(v)$ denotes the set of descendant nodes of v (including v itself).

Let a sorted list L and a sorted list J be given. Using the terminology of Cole [13], we say that L is a *c-cover* of J if between each two adjacent items in $(-\infty, L, \infty)$ there are at most c items from J (where $(-\infty, L, \infty)$ denotes the list consisting of $-\infty$, followed by the elements of L, followed by ∞). We let $SAMP_c(L)$ denote the sorted list consisting of every cth element of L, and call this list the *c-sample* of L. That is, $SAMP_c(L)$ consists of the cth element of L followed by the $(2c)$th element of L, and so on.

The algorithm for constructing $U(v)$ for each $v \in T$ proceeds in stages. Intuitively, in each stage we will be performing a portion of the merge of $U(lchild\,(v))$ and $U(rchild\,(v))$ to give the list $U(v)$. After performing a portion of this merge we will gain some insight into how to perform the merge at v's parent. Consequently, we will pass some of the elements formed in the merge at v to v's parent, so we can begin performing the merge at v's parent.

Specifically, we denote the list stored at a node v in T at stage s by $U_s(v)$. Initially, $U_0(v)$ is empty for every node except the leaf nodes of T, in which case $U_0(v)$ contains the item stored at the leaf node v (if there is such an item). We say that an internal node v is *active at stage s* if $\lfloor s/3 \rfloor \leqq alt(v) \leqq s$, and we say v is *full at stage s* if $alt(v) = \lfloor s/3 \rfloor$. As will become apparent below, if a node v is full, then $U_s(v) = U(v)$. For each active node $v \in T$ we define the list $U'_{s+1}(v)$ as follows:

$$U'_{s+1}(v) = \begin{cases} SAMP_4(U_s(v)) & \text{if } alt(v) \geqq s/3, \\ SAMP_2(U_s(v)) & \text{if } alt(v) = (s-1)/3, \\ SAMP_1(U_s(v)) & \text{if } alt(v) = (s-2)/3. \end{cases}$$

At stage $s+1$ we perform the following computation at each internal node v that is currently active.

Per-stage computation $(v, s+1)$. Form the two lists $U'_{s+1}(lchild\,(v))$ and $U'_{s+1}(rchild\,(v))$, and compute the new list

$$U_{s+1}(v) := U'_{s+1}(lchild\,(v)) \cup U'_{s+1}(rchild\,(v)).$$

This formalizes the notion that we pass information from the merges performed at the children of v in stage s to the merge being performed at v in stage $s+1$. Note that until v becomes full, $U'_{s+1}(v)$ will be the list consisting of every fourth element of $U_s(v)$. This continues to be true about $U'_{s+1}(v)$ up to the point that v becomes full. If s_v is the stage at which v becomes full (and $U_s(v) = U(v)$), then at stage s_v+1, $U'_{s+1}(v)$ is the two-sample of $U_s(v)$, and, at stage s_v+2, $U'_{s+1}(v) = U_s(v)$ $(=U(v))$. Thus, at stage s_v+3, $parent(v)$ is full. Therefore, after $3 * height(T)$ stages every node has become full and the algorithm terminates. We have yet to show how to perform each stage in $O(1)$ time using n processors.

We begin by showing that the number of items in $U_{s+1}(v)$ can be only a little more than twice the number of items in $U_s(v)$, a property that is essential to the construction.

LEMMA 2.1. *For any stage $s \geqq 0$ and any node $v \in T$, $|U_{s+1}(v)| \leqq 2|U_s(v)| + 4$.*

Proof. The proof is by induction on s.

Basis $(s=0)$. The claim is clearly true for $s=0$.

Induction step $(s>0)$. Assume the claim is true for stage $s-1$. If v is full (i.e., $alt(v) = \lfloor s/3 \rfloor$), then the claim is obviously true, since $U_{s+1}(v) = U_s(v) = U(v)$. Consider the case where either the children of v were not full at stage s or had just become full at stage s. We know that $U_{s+1}(v) = U'_{s+1}(x) \cup U'_{s+1}(y)$, where $x = lchild(v)$ and

$y = rchild\,(v)$. In addition, we have the following:

$$\begin{aligned}|U_{s+1}(v)| &= \left\lfloor \frac{|U_s(x)|}{4} \right\rfloor + \left\lfloor \frac{|U_s(y)|}{4} \right\rfloor \quad \text{(from definitions)}\\ &\leqq \left\lfloor \frac{2|U_{s-1}(x)|+4}{4} \right\rfloor + \left\lfloor \frac{2|U_{s-1}(y)|+4}{4} \right\rfloor \quad \text{(by induction hypothesis)}\\ &\leqq 2\left(\left\lfloor \frac{|U_{s-1}(x)|}{4} \right\rfloor + \left\lfloor \frac{|U_{s-1}(y)|}{4} \right\rfloor \right) + 4\\ &= 2|U_s(v)| + 4.\end{aligned}$$

The case when the children of v are full at stage $s-1$ is similar (except that one divides by 2 or 1 instead of 4). Actually, it is simpler, since in this case the children of v were full in stage $s-1$; hence, the step using the induction hypothesis can be replaced by a simple algebraic substitution step. □

In the next lemma we show that the way in which the $U_s(v)$'s grow is "well behaved."

LEMMA 2.2. *Let $[a, b]$ be an interval with $a, b \in (-\infty, U'_s(v), \infty)$. If $[a, b]$ intersects $k+1$ items in $(-\infty, U'_s(v), \infty)$, then it intersects at most $8k+8$ items in $U_s(v)$ for all $k \geqq 1$ and $s \geqq 1$.*

Proof. The proof is by induction on s. The claim is initially true (for $s = 1$). Actually, for any stage s, if $U'_s(v)$ is empty, then $U_{s-1}(v)$ contains at most three items, hence, $U_s(v)$ contains at most ten elements, by the previous lemma. Also, if $U'_s(v)$ contains one item, then $U_{s-1}(v)$ contains at most seven items, hence, $U_s(v)$ contains at most 18 items, by the previous lemma. At most 15 of these items can be between any two adjacent items in $(-\infty, U'_s(v), \infty)$, since the item in $U'_s(v)$ was the fourth item in $U_{s-1}(v)$ by definition.

Inductive step (assume true for stage s). Let $[a, b]$ be an interval with a, b both in the list $(-\infty, U'_{s+1}(v), \infty)$, and suppose $[a, b]$ intersects $k+1$ items in $(-\infty, U'_{s+1}(v), \infty)$. The lemma is immediately true if v was full stage s, since the smallest sample we take is a four-sample. So, next, suppose that either the children of v are not full or have just become full in stage s. Let g be the number of items in $(-\infty, U_s(v), \infty)$ intersected by $[a, b]$. Recall that $U_s(v) = U'_s(lchild\,(v)) \cup U'_s(rchild\,(v))$. Let $[a_1, b_1]$ (respectively, $[a_2, b_2]$) be the smallest interval containing $[a, b]$ such that $a_1, b_1 \in (-\infty, U'_s(lchild\,(v)), \infty)$ (respectively, $a_2, b_2 \in (-\infty, U'_s(rchild\,(v)), \infty)$). Suppose the interval $[a_1, b_1]$ intersects $h+1$ items in the list $(-\infty, U'_s(lchild\,(v)), \infty)$ and $[a_2, b_2]$ intersects $j+1$ items in $(-\infty, U'_s(rchild\,(v)), \infty)$. Note that $h + j = g$. By the induction hypothesis, $[a_1, b_1]$ intersects at most $8h+8$ items in $U_s(lchild\,(v))$, and hence at most $(8h+8)/4 = 2h+2$ items in $U'_{s+1}(lchild\,(v))$. Likewise, $[a_2, b_2]$ intersects at most $2j+2$ items in $U'_{s+1}(rchild\,(v))$. The definition of $U'_{s+1}(v)$ implies that $g \leqq 4k+1$. Therefore, since $U_{s+1}(v) = U'_{s+1}(lchild\,(v)) \cup U'_{s+1}(rchild\,(v))$, $[a, b]$ intersects at most $(2h+2)+(2j+2)$ items in $U_{s+1}(v)$, where $(2h+2)+(2j+2) \leqq (2h+2)+(2(4k-h+1)+2) \leqq 8k+8$.

The proof for the case when the children of v were full in stage $s-1$ is similar. Actually, it is simpler, since the induction steps can be replaced by algebraic substitution steps in this case. □

COROLLARY 2.3. *The list $(-\infty, U'_s(v), \infty)$ is a four-cover for $U'_{s+1}(v)$, for all $s \geqq 0$.* □

This corollary is used in showing that we can perform each stage of the merge procedure in $O(1)$ time. In addition to this corollary, we also need to maintain the following rank information at the start of each stage s:

(1) For each item in $U'_s(v)$: its rank in $U'_s(\mathit{sibling}(v))$.

(2) For each item in $U'_s(v)$: its rank in $U_s(v)$ (and hence, implicitly, its rank in $U'_{s+1}(v)$).

The lemma that follows shows that the above information is sufficient to allow us to merge $U'_{s+1}(\mathit{lchild}\ (v))$ and $U'_{s+1}(\mathit{rchild}\ (v))$ into the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors.

LEMMA 2.4 (THE MERGE LEMMA) [13]. *Suppose we are given sorted lists A_s, A'_{s+1}, B'_s, B'_{s+1}, C'_s, and C'_{s+1}, where the following (input) conditions are true:*

(1) $A_s = B'_s \cup C'_s$;

(2) A'_{s+1} *is a subset of* A_s;

(3) B'_s *is a* c_1*-cover for* B'_{s+1};

(4) C'_s *is a* c_2*-cover for* C'_{s+1};

(5) B'_s *is ranked in* B'_{s+1};

(6) C'_s *is ranked in* C'_{s+1};

(7) B'_s *and* C'_s *are cross-ranked.*

Then in $O(1)$ time using $|B'_{s+1}|+|C'_{s+1}|$ processors in the CREW PRAM *model, we can compute the following (output computations):*

(1) *the sorted list* $A_{s+1} = B'_{s+1} \cup C'_{s+1}$;

(2) *the ranking of* A'_{s+1} *in* A_{s+1};

(3) *the cross-ranking of* B'_{s+1} *and* C'_{s+1}. □

We apply this lemma by setting $A_s = U_s(v)$, $A'_{s+1} = U'_{s+1}(v)$, $A_{s+1} = U_{s+1}(v)$, $B'_s = U'_s(x)$, $B'_{s+1} = U'_{s+1}(x)$, $C'_s = U'_s(y)$, and $C'_{s+1} = U'_{s+1}(y)$, where $x = \mathit{lchild}\ (v)$ and $y = \mathit{rchild}\ (v)$. Note that assigning the lists of Lemma 2.4 in this way satisfies input conditions (1)-(4) from the definitions. The ranking information we maintain from stage to stage satisfies input conditions (5)-(7). Thus, in each stage s, we can construct the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors. Also, the new ranking information (of output computations (2) and (3)) gives us the input conditions (5)-(7) for the next stage. By Corollary 2.3 we have that the constants c_1 and c_2 (of input conditions (3) and (4)) are both equal to four. Note that in stage s it is only necessary to store the lists for $s-1$; we can discard any lists for stages previous to that.

The method for performing all these merges with a total of $|T|$ processors is basically to start out with $O(1)$ virtual processors assigned to each leaf node, and each time we pass k elements from a node v to the parent of v (to perform the merge at the parent), we also pass $O(k)$ virtual processors to perform the merge. When v's parent becomes full, then we no longer "store" any processors at v. (See [17] for details.) There can be at most $O(n)$ elements present in active nodes of T for any stage s (where n is the number of leaves of T), since there are n elements present on the full level, at most $n/2$ on the level above that, $n/8$ on the level above that, and so on. Thus, we can perform the entire generalized cascading procedure using $O(n)$ virtual processors, or n actual processors (by a simple simulation argument). This also implies that we need only $O(n)$ storage for this computation, in addition to that used for the output, since once a node v becomes full we can consider the space used for $U(v)$ to be part of the output. Equivalently, if we are using the generalized merging procedure in an algorithm that does not need a $U(v)$ list once v's parent becomes full, then we can implement that algorithm in $O(n)$ space by deallocating the space for a $U(v)$ list once it is no longer needed (this is in fact what we will be doing in § 6).

It will often be more convenient to relax the condition that there be at most one item stored at each leaf. So, suppose there is an unsorted set $A(v)$ (which may be empty) stored at each leaf. In this case we can construct a tree T' from T by replacing each leaf v of T with a complete binary tree with $|A(v)|$ leaves, and associating each item in $A(v)$ with one of these leaves. T' would now satisfy the conditions of the method outlined above. We incorporate this observation in the following theorem, which summarizes the discussion of this section.

THEOREM 2.5. *Suppose we are given a binary tree T such that there is an unsorted set $A(v)$ (which may be empty) stored at each leaf. Then we can compute, for each node $v \in T$, the list $U(v)$, which is the union of all items stored at descendents of v, sorted in an array. This computation can be implemented in $O(height(T) + \log(\max_v |A(v)|))$ time using a total of $n + N$ processors in the* CREW PRAM *computational model, where n is the number of leaves of T and N is the total number of items stored in T.*

Proof. The complexity bounds follow from the fact that the tree T' described above would have height at most $O(height(T) + \log(\max_v |A(v)|))$ and $|T'|$ is $O(|T| + N)$. □

The above method comprises one of the main building blocks of the algorithms presented in this paper. We present another important building block in the following section.

3. Fractional cascading in parallel.

Given a directed graph $G = (V, E)$, such that every node v contains a sorted list $C(v)$, the fractional cascading problem is to construct an $O(n)$ space data structure that, given a walk $(v_1, v_2, \cdots, v_m)$ in G and an arbitrary element x, enables a single processor to locate x quickly in each $C(v_i)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. Fractional cascading problems arise naturally from a number of computational geometry problems. As a simple example of a fractional cascading problem, suppose we have five different English dictionaries and would like to build a data structure that would allow us to look up a word w in all the dictionaries. Chazelle and Guibas [12] give an elegant $O(n)$ time sequential method for constructing a fractional cascading data structure from any graph G, as described above, achieving a search time of $O(\log n + m \log d(G))$, where $d(G)$ is the maximum degree of any node in G. However, their approach does not appear to be "parallelizable."

In this section we show how to construct a data structure achieving the same performance as that of Chazelle and Guibas in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors. Our method begins with a preprocessing step similar to one used by Chazelle and Guibas where we "expand" each node of G into two binary trees—one for its in-edges and one for its out-edges—so that each node in our graph has in-degree and out-degree at most 2. We then perform a cascading merge procedure in stages on this graph. Each catalogue $C(v)$ is "fed into" the node v in samples that double in size with each stage and these lists are in turn sampled and merged along the edges of G. Lists continue to be sampled and "pushed" across the edges of G (even in cycles) for a logarithmic number of stages, at which time we stop the computation and add some links between elements in adjacent lists. We conclude this section by showing that this gives us a fractional cascading data structure, and that the computation can be implemented in $O(\log n)$ time and $O(n)$ space using $\lceil n/\log n \rceil$ processors.

We show below how to perform the computations in $O(\log n)$ time and $O(n)$ space using n processors. We will show later how to get the number of processors down to $\lceil n/\log n \rceil$ by a careful application of Brent's theorem [11].

Define $In(v, G)$ (respectively, $Out(v, G)$) to be the set of all nodes w in V such that $(w, v) \in E$ (respectively, $(v, w) \in E$). The *degree* of a vertex v, denoted $d(v)$, is defined as $d(v) = \max\{|In(v, G)|, |Out(v, G)|\}$. The *degree* of G, denoted $d(G)$, is defined as $d(G) = \max_{v \in V}\{d(v)\}$. A sequence $(v_1, v_2, \cdots, v_m)$ of vertices is a *walk* if $(v_i, v_{i+1}) \in E$ for all $i \in \{1, 2, \cdots, m-1\}$.

As mentioned above, we begin the construction by preprocessing the directed graph G to convert it into a directed graph $\hat{G} = (\hat{V}, \hat{E})$ such that $d(\hat{G}) \leqq 2$ and such that an edge (v, w) in G corresponds to a path in $\hat{G}$ of length at most $O(\log d(G))$. Specifically, for each node $v \in V$ we construct two complete binary trees T_v^{in} and T_v^{out}. Each leaf in T_v^{in} (respectively, T_v^{out}) corresponds to an edge coming into v (respectively, going out of v). So there are $|In(v, G)|$ leaves in T_v^{in} and $|Out(v, G)|$ leaves in T_v^{out}. (See Fig. 2.) We call T_v^{in} the *fan-in tree* for v and T_v^{out} the *fan-out tree* for v. An edge $e = (v, w)$ in G corresponds to a node e in $\hat{G}$ such that e is a leaf of the fan-out tree for v and e is also a leaf of the fan-in tree for w. The edges in T_v^{in} are all directed up towards the root of T_v^{in}, and the edges in T_v^{out} are all directed down towards the leaves of T_v^{out}. For each $v \in V$ we create a new node v' and add a directed edge from v' to v, a directed edge from the root of T_v^{in} to v', and an edge from v' to the root of T_v^{out}. We call v' the *gateway* for v. (See Fig. 2). Note that $d(\hat{G}) = 2$. We assume that for each node v we have access to the nodes in $In(v, \hat{G})$ as well as those in $Out(v, \hat{G})$. We structure fan-out trees so that a processor needing to go from v to w in $\hat{G}$, with $(v, w) \in E$, can correctly determine the path down T_v^{out} to the leaf corresponding to (v, w). More specifically, the leaves of each fan-out tree are ordered so that they are listed from left to right by increasing destination name, i.e., if the leaf in T_v^{out} for $e = (v, u)$ is to the left of the leaf for $f = (v, w)$, then $u < w$. (The leaves of T_v^{in} need not be sorted, since all edges are directed towards the root of that tree.) If we are not given the $Out(v, G)$ sets in sorted order, then we must perform a sort as a part of the T_v^{out} construction, which can be done in $O(\log d(G))$ time using n processors using Cole's merge sorting algorithm [13]. We also store in each internal node z of T_v^{out} the leaf node u that has the smallest name of all the descendants of z.

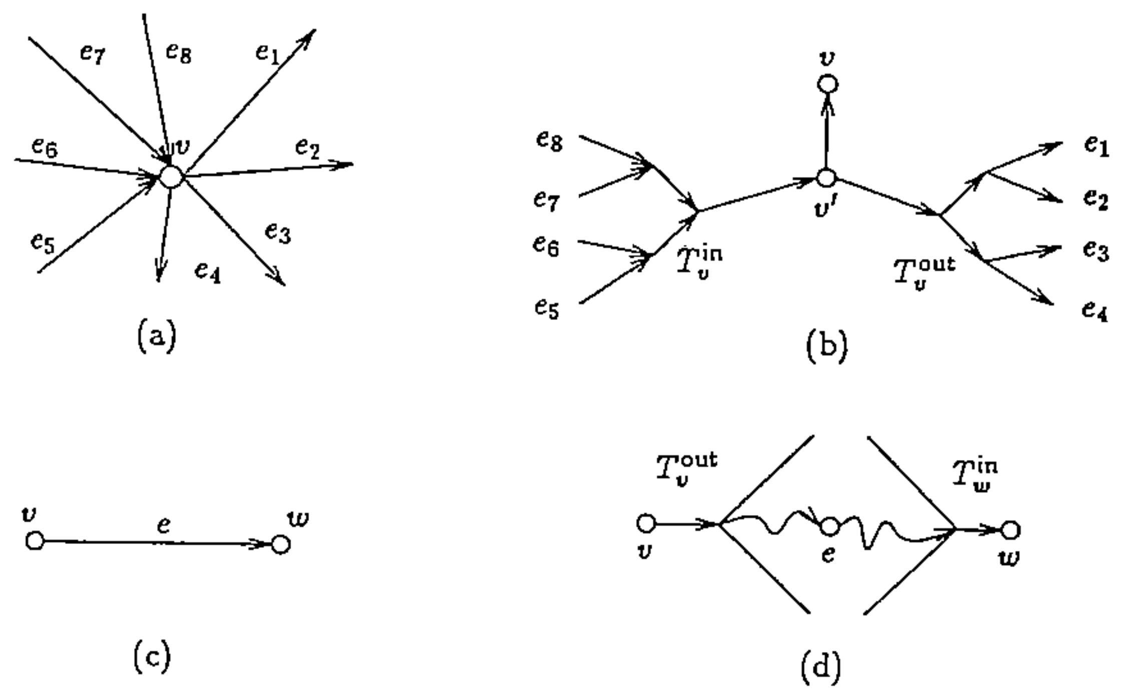


FIG. 2. *Converting G into a bounded degree graph $\hat{G}$. A node v in G* (a) *corresponds into a node v adjacent to its gateway v′, which is connected to the fan-in tree and the fan-out tree for v* (b). *An edge e in G* (c) *is converted into a node in $\hat{G}$ which corresponds to a leaf node of two trees* (d).

The above preprocessing step is similar to a preprocessing step used in the sequential fractional cascading algorithm of Chazelle and Guibas [12]. This is where the resemblance to the sequential algorithm ends, however.

The goal for the rest of the computation is to construct a special sorted list $B(v)$, which we call the *bridge list*, for every node $v \in \hat{V}$. We shall define these bridge lists so that $B(v) = C(v)$ if v is in V; if v is in $\hat{V}$ but not in V, then for every $(v, w) \in \hat{E}$, if a single processor knows the position of a search item x in $B(v)$, it can find the position of x in $B(w)$ in $O(1)$ time.

The construction of the $B(v)$'s proceeds in stages. Let $B_s(v)$ denote the bridge list stored at node $v \in \hat{V}$ at the end of stage s. Initially, $B_0(v) = \varnothing$ for all v in $\hat{V}$. Intuitively, the per-stage computation is designed so that if v came from the original graph G (i.e., $v \in V$), then v will be "feeding" $B_s(v)$ with samples of the catalogue $C(v)$ that double in size with each stage. These samples are then cascaded back into the gateway v' for v and from there back through the fan-in tree for v. We will also be merging any samples "passed back" from the fan-out tree for v with $B_s(v')$, and cascading these values back through the fan-in tree for v as well. We iterate the per-stage computation for $\lceil \log N \rceil$ stages, where N is the size of the largest catalogue in G. We will show that after we have completed the last stage, and updated some ranking pointers, $\hat{G}$ will be a fractional cascading data structure for G. The details follow.

Recall that $B_0(v) = \varnothing$ for all $v \in \hat{V}$. For stage $s \geqq 0$, we define $B'_{s+1}(v)$ and $B_{s+1}(v)$ as follows:

$$B'_{s+1}(v) = \begin{cases} SAMP_4(B_s(v)) & \text{if } v \in \hat{V} - V, \\ SAMP_{c(s)}(C(v)) & \text{if } v \in V, \end{cases}$$

$$B_{s+1}(v) = \begin{cases} B'_{s+1}(w_1) \cup B'_{s+1}(w_2) & \text{if } Out(v, \hat{G}) = \{w_1, w_2\}, \\ B'_{s+1}(w) & \text{if } Out(v, \hat{G}) = \{w\}, \\ \varnothing & \text{if } Out(v, \hat{G}) = \varnothing, \end{cases}$$

where $c(s) = 2^{\lceil \log N \rceil - s}$ and N is the size of the largest catalogue. The per-stage computation, then, is as follows.

Per-stage computation $(v, s+1)$. Using the above definitions, construct $B_{s+1}(v)$ for all $v \in \hat{V}$ in parallel (using $|B_{s+1}(v)|$ processors for each v).

The function $c(s)$ is defined so that if $v \in V$, then as the computation proceeds the list $B'_{s+1}(v)$ will be empty for a while. Then at some stage $s+1$, it will consist of a single element of $C(v)$ (the $(2^{\lceil \log N \rceil - s})$th element), in stage $s+2$ at most three elements (evenly sampled), in stage $s+3$ at most five elements, in stage $s+4$ at most nine elements, and so on. This continues until the final stage (stage $\lceil \log N \rceil$), when $B'_{s+1}(v) = C(v)$. Intuitively, the $c(s)$ function is a mechanism for synchronizing the processes of "feeding" the $C(v)$ lists into the nodes of $\hat{G}$ so that all the processes complete at the same time. We show below that each stage can be performed in $O(1)$ time, resulting in a running time of the cascading computations that is $O(\log N)$ (plus the time it takes time to compute the value of N, namely, $O(\log n)$). The following important lemma is similar to Lemma 2.1 in that it guarantees that the bridge lists do not grow "too much" from one stage to another.

LEMMA 3.1. *For any stage $s \geqq 0$ and any node $v \in T$, $|B_{s+1}(v)| \leqq 2|B_s(v)| + 4$.*

Proof. The proof is by induction on s.

Basis $(s = 0)$. The claim is clearly true for $s = 0$.

Induction step $(s > 0)$. Assume the claim is true for stage $s-1$. If $v \in V$, then the claim follows immediately from the definition of $c(s)$, since in this case $B_{s+1}(v)$ and

$B_s(v)$ are both samples of $C(v)$ with $B_{s+1}(v)$ being twice as fine as $B_s(v)$, i.e., $|B_{s+1}(v)| \leqq 2|B_s(v)| + 1$.

Consider the case when $v \in \hat{V} - V$, and $Out(v, \hat{G}) = \{w_1, w_2\}$. We know in this case $B_{s+1}(v) = B'_{s+1}(w_1) \cup B'_{s+1}(w_2)$. Thus, we have the following:

$$\begin{aligned}|B_{s+1}(v)| &= \left\lfloor \frac{|B_s(w_1)|}{4} \right\rfloor + \left\lfloor \frac{|B_s(w_2)|}{4} \right\rfloor \quad \text{(from definitions)}\\ &\leqq \left\lfloor \frac{2|B_{s-1}(w_1)|+4}{4} \right\rfloor + \left\lfloor \frac{2|B_{s-1}(w_2)|+4}{4} \right\rfloor \quad \text{(by induction hypothesis)}\\ &\leqq 2\left(\left\lfloor \frac{|B_{s-1}(w_1)|}{4} \right\rfloor + \left\lfloor \frac{|B_{s-1}(w_2)|}{4} \right\rfloor\right) + 4\\ &= 2|B_s(v)| + 4.\end{aligned}$$

For the case when $v \in \hat{V} - V$ and $Out(v, \hat{G})$ contains only one node, w, the argument is similar and, in fact, simpler. We simply repeat the above argument, replacing w_1 with w and eliminating those terms that contain w_2. □

In the next lemma we show that the way in which the $B_s(v)$'s grow is "well behaved," much as we did in Lemma 2.2.

LEMMA 3.2. *Let $[a, b]$ be an interval with $a, b \in (-\infty, B'_s(v), \infty)$. If $[a, b]$ intersects $k+1$ items in $(-\infty, B'_s(v), \infty)$, then it intersects at most $8k+8$ items in $B_s(v)$ for all $k \geqq 1$ and $s \geqq 1$.*

Proof. The proof is structurally the same as that of Lemma 2.2, since that lemma was based on a merge definition similar to that for $B_{s+1}(v)$. □

COROLLARY 3.3. *The list $(-\infty, B'_s(v), \infty)$ is a four-cover for $B'_{s+1}(v)$, for $s \geqq 0$.*

COROLLARY 3.4. *The list $(-\infty, B_s(v), \infty)$ is a 16-cover for $B_s(w)$, for $s \geqq 0$ and $(v, w) \in \hat{E}$.*

The first of these two corollaries implies that we can satisfy all the c-cover input conditions for the Merge Lemma (Lemma 2.4) for performing the merge operations for the computation at stage s in $O(1)$ time using n_s processors, where $n_s = \sum_{v \in \hat{V}} |B_s(v)|$. We use the second corollary to show that when the computation is completed we will have a fractional cascading data structure (after adding the appropriate rank pointers). We maintain the following rank information at the start of each stage s.

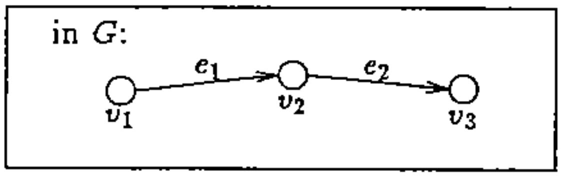
(1) For each item in $B'_s(v)$: its rank in $B'_s(w)$ if $In(v, \hat{G}) \cap In(w, \hat{G})$ is nonempty, i.e., if there is a vertex u such that $(u, v) \in \hat{E}$ and $(u, w) \in \hat{E}$.

(2) For each item in $B'_s(v)$: its rank in $B_s(v)$ (and thus, implicitly, its rank in $B'_{s+1}(v)$).

By having this rank information available at the start of each stage s, we satisfy all the ranking input conditions of the Merge Lemma. Thus, we can perform each stage in $O(1)$ time using n_s processors. Moreover, the output computations of the Merge Lemma allow us to maintain all the necessary rank information into the next stage. Note that in stage s it is only necessary to store the lists for $s-1$; we can discard any lists for stages previous to that, as in the generalized cascading merge.

Recall that we perform the computation for $\lceil \log N \rceil$ stages, where N is the size of the largest catalogue. When the computation completes, we take $B(v) = B_s(v)$ for all $v \in \hat{V}$, and for each $(v, w) \in \hat{E}$ we rank $B(v)$ in $B(w)$. We can perform this ranking step by the following method. Assign a processor to each element b in $B(v)$ for all $v \in \hat{V}$ in parallel. The processor for b can find the rank of b in each $B'_s(w)$ such that $w \in Out(v, \hat{G})$ in $O(1)$ time because $B_s(v)$ contains $B'_s(w)$ as a proper subset ($B'_s(w)$ was one of the lists merged to make $B_s(v)$). This processor can then determine the rank of b in $B(w) = B_s(w)$ for each $w \in Out(v, \hat{G})$ in $O(1)$ time by using the ranking information we maintained (from $B'_s(w)$ to $B_s(w)$) for stage s (rank condition (2) above).

Given a walk $W = (v_1, \cdots, v_m)$, and an arbitrary element x, the query that asks for locating x in every $C(v_i)$ is called the *multilocation* of x in $(v_1, \cdots, v_m)$. To perform a multilocation of x in a walk $(v_1, \cdots, v_m)$, we extend the walk W in G to its corresponding walk $\hat{W} = (\hat{v}_1, \cdots, \hat{v}'_m)$ in $\hat{G}$ and perform the corresponding multilocation in $\hat{G}$, similar to the method given by Chazelle and Guibas [12] for performing multilocations in their data structure. The multilocation begins with the location of x in $B(\hat{v}_1) = B(v'_1)$, the gateway bridge list for v_1, by binary search. For each other vertex in this walk we can locate the position of x in $B(\hat{v}_i)$ given its position in $B(\hat{v}_{i-1})$ in $O(1)$ time. The method is to follow the pointer from x's predecessor in $B(\hat{v}_{i-1})$ to the predecessor of that element in $B(\hat{v}_i)$ and then locate x in $B(\hat{v}_i)$ by a linear search from that position (which will require at most 15 comparisons by Corollary 3.4). In addition, if $\hat{v}_i$ corresponds to a gateway v', then we can locate x in $C(v)$ in $O(1)$ time given its position in $B(v')$ by a similar argument. (See Fig. 3.) Since each edge in the walk W corresponds to a path in $\hat{W}$ of length at most $O(\log d(G))$, this implies that we can perform the multilocation of x in $(v_1, \cdots, v_m)$ in $O(\log |B(v'_1)| + m \log d(G))$ time. In other words, $\hat{G}$ is a fractional cascading data structure. We show that $\hat{G}$ uses $O(n)$ space in the following lemma.



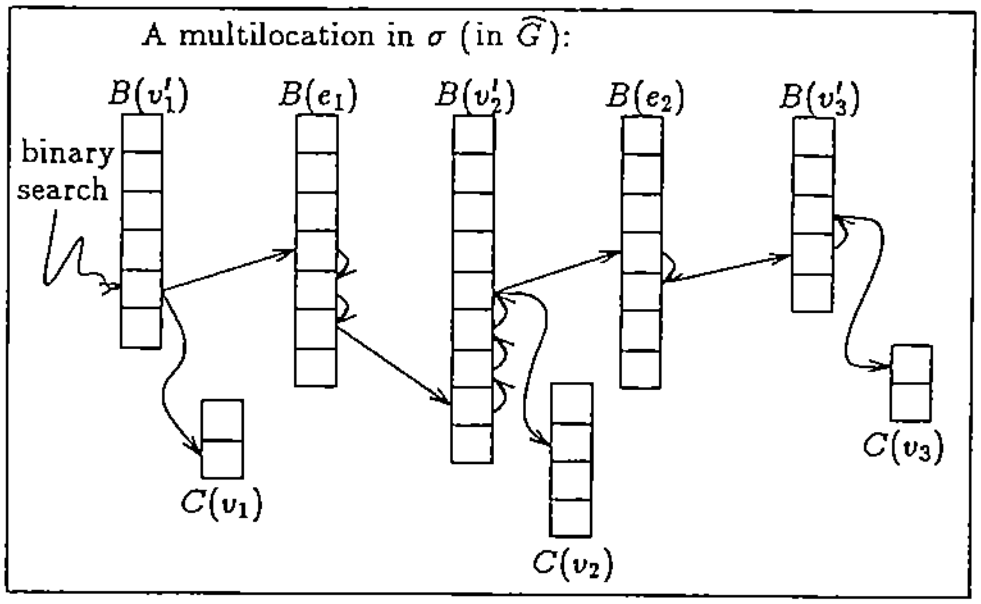


FIG. 3. *Multilocating an element x in (v_1, v_2, v_3).*

LEMMA 3.5. *Let n_v denote the amount of space that is added to $\hat{G}$ because of the presence of a particular catalogue $C(v)$, $v \in V$. Then $n_v \leqq 2|C(v)|$.*

Proof. Recall that while constructing the bridge lists in $\hat{G}$ we copy one-fourth of the elements in each bridge list to at most two of its neighbors. Thus, we have the following:

$$n_v \leqq |C(v)| + 2\lfloor |C(v)|/4 \rfloor + 2^2\lfloor |C(v)|/4^2 \rfloor + 2^3\lfloor |C(v)|/4^3 \rfloor + \cdots$$
$$\leqq 2|C(v)|.$$

(This is obviously an overestimate, but it is good enough for the purposes of the analysis.) □

COROLLARY 3.6. *The total amount of space used by the fractional cascading data structure is $O(n)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$.*

Proof. The total amount of space used by the fractional cascading data structure is $O(|\hat{V}| + |\hat{E}| + \sum_{v \in \hat{V}} |B(v)|)$. Since all the bridge lists start out empty, $\sum_{v \in \hat{V}} |B(v)|) = \sum_{v \in V} n_v$. The previous lemma implies that $\sum_{v \in V} n_v \leqq \sum_{v \in V} 2|C(v)|$. Therefore, since $|\hat{V}| + |\hat{E}|$ is $O(|V| + |E|)$ by the definition of $\hat{G}$, the total amount of space used by the fractional cascading data structure is $O(n)$. □

Note that the upper bound on the space of the fractional cascading data structure holds even if $\hat{G}$ contains cycles. This corollary, then, implies that we can construct a fractional cascading data structure $\hat{G}$ from any catalogue graph G in $O(\log n)$ time and $O(n)$ space using n processors, even if G contains cycles. We have not shown, however, how to assign these n processors to their respective jobs.

The method for performing the processor allocation is as follows. Initially, we assign $2|C(v)|$ virtual processors to each node $v \in V$ and no processors to each node $v \in \hat{V} - V$. This requires at most $2n$ virtual processors; hence, can be easily simulated with n actual processors. Each time we pass k elements from a node v to a node w (in performing the merge at node w) we also pass along (exactly) k virtual processors to go with them. When we say that we are passing a virtual processor from some node v to some node w, all we are actually changing is the node to which that processor is assigned. Since, by Lemma 3.5, $n_v \leqq 2|C(v)|$, we know that there are enough virtual processors assigned to $v \in V$ to do this. To see that this also suffices for $v \in \hat{V} - V$ note that at the beginning of stage s node v has $|B_{s-1}(v)|$ elements (and processors). We "give away" at most $2\lfloor |B_{s-1}(v)|/4 \rfloor$ elements (and processors) from $B_{s-1}(v)$ in stage s and receive $|B_s(v)|$ elements (and processors). Consequently, there are enough processors to perform the merge to construct $B_s(v)$ and repeat the give-away procedure for the next stage. In addition, since we pass a processor for each item we pass to another node, each processor p_i can maintain not only which node it is assigned but p_i can also maintain m_v, the number of other processors that are assigned to that node, as well as maintaining a unique integer identification for itself in the range $[1, m_v]$. Thus, we have the following lemma.

LEMMA 3.7. *Given any catalogue graph G, we can construct a fractional cascading data structure for G in $O(\log n)$ time and $O(n)$ space using n processors in the* CREW PRAM *model.* □

Thus, we can solve the fractional cascading problem in $O(\log n)$ time using n processors. For the applications we study in this paper, however, we can do even better. The following lemma enumerates two important situations where the method just described can be improved.

LEMMA 3.8. *Given any catalogue graph G, if $d(G)$ is $O(1)$ or if we are given $Out(v, G)$ in sorted order for each $v \in V$, then the total number of operations performed by the fractional cascading algorithm is $O(n)$.*

Proof. If $d(G)$ is $O(1)$ or we are given $Out(v, G)$ in sorted order, then the construction of the graph $\hat{G}$ (without any bridge lists) requires only $O(n)$ operations, since we do not have to perform any sorting. Let us account for the total work performed by computing the total number of other operations that are performed because of the fact that the catalogue for each node v contains $|C(v)|$ elements (we will only charge vertices in V). Let s_v be the first stage that $B_s(v')$ becomes nonempty. In this stage $B_s(v')$ receives one element of $C(v)$ from v, and hence we charge one operation in stage s_v for the node v. In stage s_v+1 we will then perform at most 3 operations, at most 7 in stage s_v+2, at most 15 in stage s_v+3, and so on. As soon as $B_s(v')$ contains at least four elements from v (as early as stage s_v+2), then we will perform one more operation, passing one element to the fan-in tree for v. In the next stage, s_v+3, we will perform at most two additional operations, then at most four additional operations in stage s_v+4, and so on. This pattern will "ripple" back through the fan-in tree for v and on through the graph $\hat{G}$ for as long as the computation proceeds. Specifically, the number of operations charged to a node $v \in V$ is, at most, the sum of the following $k_v = \lceil \log_4 |C(v)| \rceil$ rows:

$$\begin{array}{ccccccccc}
1 & 3 & 7 & 15 & 31 & 63 & 127 & \cdots & |C(v)| \\
 & & 2*1 & 2*3 & 2*7 & 2*15 & 2*31 & \cdots & 2\lfloor |C(v)|/4 \rfloor \\
 & & & & 2^2*1 & 2^2*3 & 2^2*7 & \cdots & 2^2\lfloor |C(v)|/4^2 \rfloor \\
 & & & & & & \vdots & & \vdots \\
 & & & & & & & & 2^{k_v}*1
\end{array}$$

where the number in row i and column j corresponds to the maximum number of operations performed in stage s_v+j-1 at nodes at distance i from v because of the fact that the catalogue at node v contains $|C(v)|$ elements. (This is actually an overestimate, since not all nodes in $\hat{G}$ have out-degree 2). Summing the number of operations for each row, and then summing the rows, we get that the number of operations charged to $v \in V$ is at most $2(|C(v)|+2\lfloor |C(v)|/4 \rfloor + 2^2\lfloor |C(v)|/4^2 \rfloor + \cdots + 2^{k_v})$, which is at most $4|C(v)|$. Thus, the total number of operations performed by the fractional cascading algorithm is $O(n)$. $\square$

This lemma immediately suggests that we may be able to apply Brent's theorem to the fractional cascading algorithm so that it runs in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors.

THEOREM 3.9 ([11]). *Any synchronous parallel algorithm taking time T that consists of a total of N operations can be simulated by P processors in $O(\lfloor N/P \rfloor + T)$ time.*

Proof of Brent's theorem. Let N_i be the number of operations performed at step i in the parallel algorithm. The P processors can simulate step i of the algorithm in $O(\lceil N_i/P \rceil)$ time. Thus, the total running time is $O(\lfloor N/P \rfloor + T)$:

$$\sum_{i=1}^{T} \lceil N_i/P \rceil \leq \sum_{i=1}^{T} (\lfloor N_i/P \rfloor + 1) \leq \lfloor N/P \rfloor + T. \qquad \square$$

There are two qualifications we must make to Brent's theorem before we can apply it in the PRAM model, however. The first is that we must be able to compute N_i at the beginning of step i in $O(\lceil N_i/P \rceil)$ time using P processors. And, second, we must know how to assign each processor to its job. Thus, in order to apply Brent's theorem to our problem of doing fractional cascading, we must deal with these processor allocation problems.

Let $\Gamma = \{p_1, p_2, \cdots, p_m\}$ be the set of virtual processors used in the fractional cascading algorithm (with $m \leqq 2n$), and let $\Gamma' = \{p'_1, p'_2, \cdots, p'_{\lceil n/\log n \rceil}\}$ be the set of processors we will be using to simulate the fractional cascading algorithm. Assuming that $d(G)$ is constant or we are given the list of vertices in $Out(v, G)$ in sorted order, we can compute the graph $\hat{G}$ and the initial assignment of processors from Γ, so that we assign $2|C(v)|$ virtual processors to each node $v \in V$, in $O(\log n)$ time using the processors in Γ' by a parallel prefix computation. (Recall that the problem of computing all prefix sums $c_k = \sum_{i=1}^{k} a_i$ of a sequence of integers $(a_1, a_2, \cdots, a_n)$ can be done in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors [21], [22].) Let $v(p_i)$ denote the vertex in $\hat{G}$ to which $p_i \in \Gamma$ is assigned. Recall that we will be "passing" the processor p_i around $\hat{G}$ during the computation, so the value of $v(p_i)$ can change from one stage to the next. Once a processor p_i becomes active, it stays active for the remainder of the computation. So, the only thing left to show is how to compute the number of processors active in stage s, and to assign the processors in Γ' to their respective tasks of simulating the processors in Γ. We do this by sorting the set of processors in Γ by the stage in which they become active. It is easy to compute the stage in which a processor p_i becomes active in $O(1)$ time, because this depends only on the initial value of $v(p_i)$ and the size of $C(v(p_i))$ relative to N (the size of the largest catalogue). We can sort the processors in Γ by the stage in which they become active in $O(\log n)$ time using the $\lceil n/\log n \rceil$ processors in Γ', by using an algorithm from Reif [23] (since the stage numbers fall in the range $[1, \lceil \log N \rceil]$). Thus, by performing a parallel prefix computation on this ordered list of processors, we can determine the number of processors active in each stage s, and also know how to assign the processors in Γ' so that they optimally simulate the activities of the processors in Γ during stage s. We thus have established the following theorem.

THEOREM 3.10. *Given a catalogue graph $G = (V, E)$, such that $d(G)$ is $O(1)$ or given each $Out(v, G)$ set in sorted order, we can build a fractional cascading data structure for G in $O(\log n)$ time and $O(n)$ space using $\lceil n/\log n \rceil$ processors in the* CREW PRAM *model, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. This bound is optimal.* □

4. The plane-sweep tree data structure. In this section we define a data structure, which we call the plane-sweep tree, and show how to use it and the fractional cascading procedure of the previous section to solve the trapezoidal decomposition problem and the planar-point location problem in $O(\log n)$ time using n processors. Since the construction of this data structure is quite involved, we merely define the data structure now, and show how to construct it in these same bounds in § 5.

Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of nonintersecting line segments in the plane, and let $X(S) = (\alpha_1, \alpha_2, \cdots, \alpha_{2n})$ be the (nondecreasing) sorted list of the x-coordinates of the endpoints of the segments in S. To simplify the exposition, we assume that no two endpoints in S have the same x-coordinate, i.e., $\alpha_i < \alpha_{i+1}$. Let $X' = (x_1, x_2, \cdots, x_m)$ be some subsequence of $X(S)$ and let T be the complete binary tree whose $m+1$ leaves, in left to right order, correspond to the intervals $(-\infty, x_1]$, $[x_1, x_2]$, $[x_2, x_3]$, $\cdots$, $[x_{m-1}, x_m]$, $[x_m, +\infty)$, respectively. Associated with each internal node $v \in T$ is the interval I_v which is the union of the intervals associated with the descendants of v. Let Π_v denote the vertical strip $I_v \times (-\infty, +\infty)$. We say a segment s_i *covers a node* $v \in T$ if it spans Π_v but not $\Pi_{parent(v)}$. No segment covers more than two nodes of any level of T; hence, every segment covers at most $O(\log m)$ nodes of T. For each node $v \in T$ we let $Cover(v)$ denote the set of all segments in S that cover v.

The idea of using a tree data structure such as this to parallelize plane-sweeping is due to Aggarwal et al. [1] and is itself based on the "segment tree" of Bentley and Wood [8]. The data structure of Aggarwal et al. consists of the tree T described above with $X' = X(S)$ (i.e., it has $2n+1$ leaves). Aggarwal et al. store the list $Cover(v)$ at each node v sorted by the "above" relation for line segments. They construct these lists by first collecting the segments in each $Cover(v)$ and then sorting all the $Cover(v)$'s in parallel, an operation that requires $\Theta(\log^2 n)$ time using n processors [13], since there are a total $\Theta(n \log n)$ items to sort. Once these lists are constructed the data structure can then be used to solve various problems by performing certain searches on the nodes of T. These searches are of the following nature: given a set of $O(n)$ input points, for each point p locate the segment in $Cover(v)$ that is directly above (or below) p, for all $v \in T$ such that $p \in \Pi_v$. Notice that for the leaf-to-root walk starting with the leaf v such that $p \in \Pi_v$, this search can be solved by the multilocation of p in that walk. Aggarwal et al. [1] perform all $O(n)$ multilocations in $O(\log^2 n)$ time using n processors by assigning a processor to each point p and doing a binary search for p in all the $Cover(v)$ lists such that $p \in \Pi_v$ (there are $O(\log n)$ such lists for each p).

Although based on the structure of Aggarwal et al., the plane-sweep tree differs from it in some important ways. One such difference is that the plane-sweep tree allows us to perform $O(n)$ multilocations in $O(\log n)$ time using n processors, after a preprocessing step that takes $O(\log n)$ time using n processors. Also, instead of taking X' to be the entire $X(S)$ list, we define X' to be the list consisting of every $\lceil \log n \rceil$th element of $X(S)$, i.e., $X' = SAMP_{\lceil \log n \rceil}(X(S))$. Thus, each vertical strip Π_v associated with a leaf of T in our construction contains $O(\log n)$ segment endpoints. Like Aggarwal et al., we also store each $Cover(v)$ list sorted by the "above" relation. In addition, for every node v of T we define the set $End(v)$ as follows:

$$End(v) = \{s_i | s_i \in S, \text{ has an endpoint in } \Pi_v, \text{ and does not span } \Pi_v\}.$$

Although $End(v)$ is defined for each node of T we only construct a copy of $End(v)$ if v is a leaf node. We do not store the elements of any $End(v)$ in any particular order. This is due to the fact that $End(v)$ contains $O(\log n)$ segments for any leaf node; hence a single processor can search the entire list in $O(\log n)$ time.

Note that all the segments in the $Cover(v)$'s of any root-to-leaf path in T are comparable by the "above" relation. Thus, if we direct all the edges in T so that each edge goes from a child to its parent, then the elements stored in any directed walk in T are all comparable by the "above" relationship. Therefore, we can apply the fractional cascading technique of the previous section to T (with each $Cover(v)$ playing the role of the catalogue $C(v)$). Since T has bounded degree and has $O(n \log n)$ space, we can, by Theorem 3.10, construct a fractional cascading data structure $\hat{T}$ for T in $O(\log n)$ time and $O(n \log n)$ space using n processors. This data structure allows us to perform the multilocation of any point p (in a leaf-to-root walk) in $O(\log n)$ time ($O(\log n)$ for the binary search at the leaf, and an additional $O(1)$ for each internal node on the path to the root). We also store the set $End(v)$ in each leaf v of $\hat{T}$. The plane-sweep tree data structure, then, consists of the tree $\hat{T}$ constructed from T by fractional cascading, where T is defined with $X' = SAMP_{\lceil \log n \rceil}(X(S))$, has $Cover(v)$ stored in sorted order for every node $v \in T$, and the set $End(v)$ stored (unsorted) for each leaf node $v \in T$ (see Fig. 4).

In § 5 we show how to construct this data structure efficiently in parallel. Since the construction is rather involved, before giving the details of the construction, we give two applications of this data structure. We begin with the trapezoidal decomposition problem.

4.1. The trapezoidal decomposition problem. Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of nonintersecting line segments in the plane. For any endpoint p of a segment in S a *trapezoidal segment* for p is a segment of S that is directly above or below p such that the vertical line segment from p to this edge is not intersected by any other segment in S. The trapezoidal decomposition problem is to find the trapezoidal segment(s) for each endpoint of the segments in S. Even in the parallel setting, this problem is often

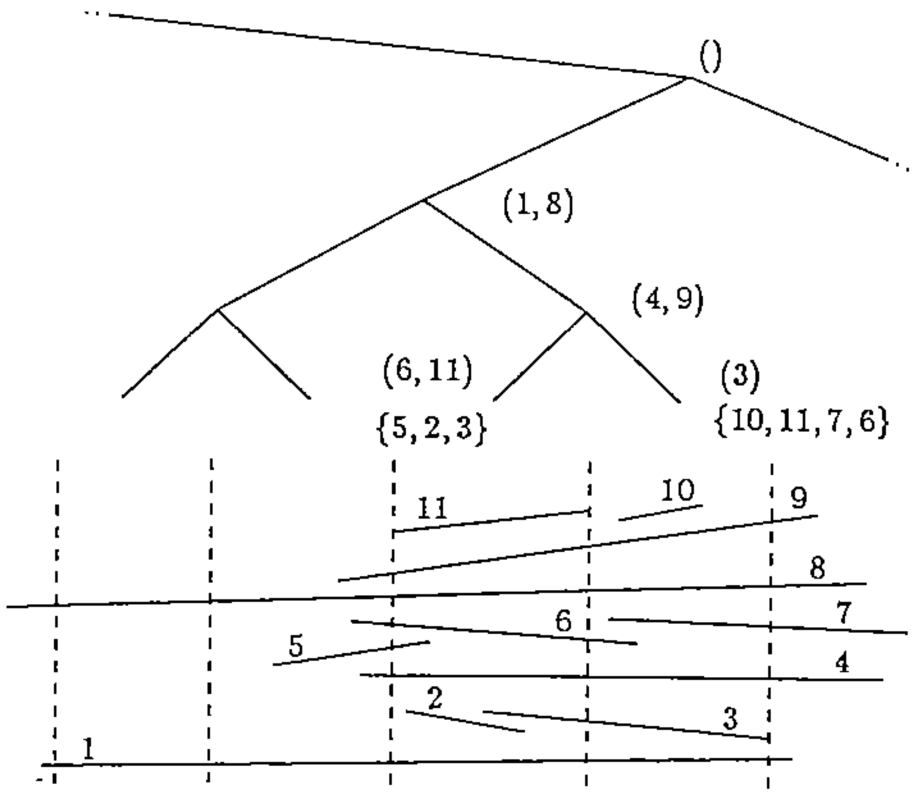


FIG. 4. *A portion of a plane-sweep tree. The segments are numbered in this example by embedding the "above" relation of* § 2 *in the total order* 1, 2, · · · , 11. *For simplicity we denote the list Cover*(*v*) *by parentheses and the set End*(*v*) *by set braces.*

used as a building block to solve other problems, such as polygon triangulation [1], [19], [28] or shortest paths in a polygon [16].

THEOREM 4.1. *A trapezoidal decomposition of a set S of n nonintersecting segments in the plane can be constructed in $O(\log n)$ time using n processors in the* CREW PRAM *model, and this is optimal.*

Proof. Construct the plane-sweep tree data structure T for S. Theorem 5.2 (to be given later, in § 5) shows that this structure can be constructed in $O(\log n)$ time using n processors. And we already know that T can be made into a fractional cascading data structure $\hat{T}$ in these same bounds. We assign a single processor to every segment endpoint (there are $2n$ such points). Let us concentrate on computing the trapezoidal segment below a single segment endpoint p. Let $(v, \cdots, root(T))$ be the leaf-to-root path in $\hat{T}$ that starts with the leaf v such that $p \in \Pi_v$. We first search through $End(v)$ to see if there are any segments in this set that are below p, and take the one that is closest to p (recall that $End(v)$ contains $O(\log n)$ segments). We then perform the multilocation of p in the leaf-to-root walk starting at v, giving us for each w such that $p \in \Pi_w$ the segment in $Cover(w)$ directly below p. We choose among these $\lceil \log n \rceil$ segments the segment that is closest to p. Comparing this segment to the one (possibly) found in $End(v)$, we get the segment in S, if there is one, that is directly below p. Since the length of the walk from v to $root(T)$ is at most $\lceil \log n \rceil$, by the method outlined at the end of § 3 [12], this computation can be done in $O(\log n)$ time using n processors. Since the two-dimensional maxima problem can be reduced to trapezoidal decomposition in $O(1)$ time using n processors [17], and the two-dimensional maxima problem has a sequential lower bound of $\Omega(n \log n)$ in the algebraic computation tree model [7], [20], we cannot do better than $O(\log n)$ time using n processors. □

Solving the trapezoidal decomposition problem efficiently in parallel has proven to be an important step in triangulating a polygon efficiently in parallel [1], [2], [5], [17], [28]. In fact, Theorem 4.1 is used in the algorithms of Goodrich [19] and Yap [28] to achieve an $O(\log n)$ time solution to polygon triangulation using only n processors. We next point out that the plane-sweep tree can also be used to solve the planar point location problem.

4.2. The planar point location problem. The planar point location problem is the following: Given a planar subdivision S consisting of n edges, construct a data structure that, once constructed, enables one processor to determine for a query point p the face in S containing p. This problem has applications in several other parallel computational geometry problems, such as Voronoi diagram construction.

THEOREM 4.2. *Given a planar subdivision S consisting of n edges, we can construct a data structure that can be used to determine for any query point p the face in S containing p in $O(\log n)$ serial time. This construction takes $O(\log n)$ time using n processors in the* CREW PRAM *model.*

Proof. The solution to this problem is to build the plane-sweep tree data structure for S (with fractional cascading) and associate with each edge s_i the name of the face above s_i. As already mentioned, Theorem 5.2 (to be given later, in § 5) shows that the tree T can be constructed in $O(\log n)$ time using n processors. Also recall that T can be made a fractional cascading data structure $\hat{T}$ in these bounds. Let a query point p be given. A planar point location query for p can be solved in $O(\log n)$ serial time by performing a multilocation like that used in the proof of Theorem 4.1 to find the segment in S directly below p. After we have determined the segment s_i in S that is directly below p, we then can read off the face of S containing p by looking up which face is directly above s_i. □

Incidentally, Theorem 4.2 immediately implies that the running time of the Voronoi diagram algorithm of Aggarwal et al. [1] can be improved from $O(\log^3 n)$ to $O(\log^2 n)$, still using only n processors. (We have recently learned that in the final version of their paper [2], they reduce the time bound of their algorithm to $O(\log^2 n)$ using a substantially different technique.)

The results of §§ 4.1 and 4.2 are conditional: they hold if we can construct the plane-sweep tree data structure efficiently in parallel. We next show how to construct the plane-sweep tree in $O(\log n)$ time using only n processors.

5. Cascading with line segment partial orders. In this section we show how to modify the cascading divide-and-conquer technique of § 2 to solve some geometric problems in which the elements being merged belong to the partial order defined by a set of nonintersecting line segments. Recall that in this partial order a segment s_1 is "above" a segment s_2 if there is a vertical line that intersects both segments, and its intersection with s_1 is above its intersection with s_2. We apply this technique to the problems of constructing the plane-sweep tree data structure and of detecting if any two of n segments in the plane intersect.

We now give a brief overview of the problems encountered and our solutions to them. The essential computation is as follows: we have a binary tree with lists stored in its leaves, and we wish to combine them in pairs (up the tree) to construct lists at internal nodes. The main difficulty is that the list stored at some node v is not defined as a simple merge of the lists stored at the children of v. Instead, its definition involves deleting elements from lists stored at children nodes before performing a merge. These deletions are quite troublesome, because if we try to perform these deletions while cascading, then the rank information will become corrupted, and the cascade will fail.

On the other hand, if we try to postpone the deletions to some postprocessing step, then there will be nondeleted elements that are not comparable to others at the same node; hence, there will be instances when processors try to compare two elements that are not comparable, and the cascade will fail. The main idea of our method for getting around these problems is to embed partial orders in total orders "on the fly" while we are cascading up the tree. That is, we change the identity of segments as they are being passed up the tree, so that the segments in any list are always linearly ordered. To be able to do this, however, we must do some preprocessing that involves simultaneously performing a number of cascading merges in parallel. We complete the computation by performing a purging postprocessing step to remove the segments that "changed identity" (as an alternative to being deleted).

For the intersection detection problem, we need to dovetail the detection of intersections with the cascading. That is, we cascade the results of intersection checks along with the segments being passed up the tree. The complication here is that if we should ever detect an intersection on the way up the tree we cannot stop and answer "yes" as this would require $O(\log n)$ time (to "fan-in" all the possible answers). Thus we are forced to proceed with the merging until we reach the root, even though in the case of an intersection the segments being merged no longer even belong to a partial order. We show that in this case we can replace the segment with a special place holder symbol so that the cascades can proceed. After the cascading merge completes we perform some postprocessing to then check if any intersections are present.

The next two subsections give the details.

5.1. Plane-sweep tree construction. In this subsection we describe how to construct the *Cover*(v) lists for each node v in the plane-sweep tree T. We begin by making a few definitions and observations. We let left (Π_v) (respectively, right (Π_v)) denote the left (right) vertical boundary line for Π_v. We define the *dominator node* of a segment s_i, denoted $dom(s_i)$, to be the deepest node v (i.e., farthest from the root) in T such that s_i is completely contained in Π_v. That is, the dominator of s_i is the node v such that s_i does not intersect left (Π_v) or right (Π_v), but s_i does intersect the vertical boundary separating $\Pi_{lchild(v)}$ and $\Pi_{rchild(v)}$. In addition, we define the following sets for each node $v \in T$:

$$L(v)=\{s_i | s_i \in End(v) \text{ and } s_i \cap \text{left}(\Pi_v) \neq \varnothing\},$$

$$R(v)=\{s_i | s_i \in End(v) \text{ and } s_i \cap \text{right}(\Pi_v) \neq \varnothing\},$$

$$l(v,d)=\{s_i | s_i \in L(v) \text{ and } d = depth(dom(s_i))\},$$

$$r(v,d)=\{s_i | s_i \in R(v) \text{ and } d = depth(dom(s_i))\}.$$

Note that $l(v,d)$ and $r(v,d)$ are only defined for $0 \leqq d < depth(v)$. Any time we construct one of these sets it will be ordered by the "above" relation, so for the remainder of this section we represent these sets as sorted lists. In the following lemma we make some observations concerning the relationships between the various lists defined above.

LEMMA 5.1. *Let v be a node in T with left child x and right child y. Then we have the following* (*see Fig.* 5):

(1) $l(v,d)=l(x,d)\cup l(y,d)$ *for* $d<depth(v)$,
(2) $r(v,d)=r(x,d)\cup r(y,d)$ *for* $d<depth(v)$,
(3) $L(v)=l(v,0)\cup l(v,1)\cup\cdots\cup l(v,depth(v)-1)$,
(4) $R(v)=r(v,0)\cup r(v,1)\cup\cdots\cup r(v,depth(v)-1)$,
(5) $L(v)=L(x)\cup(L(y)-l(y,depth(v)))$,

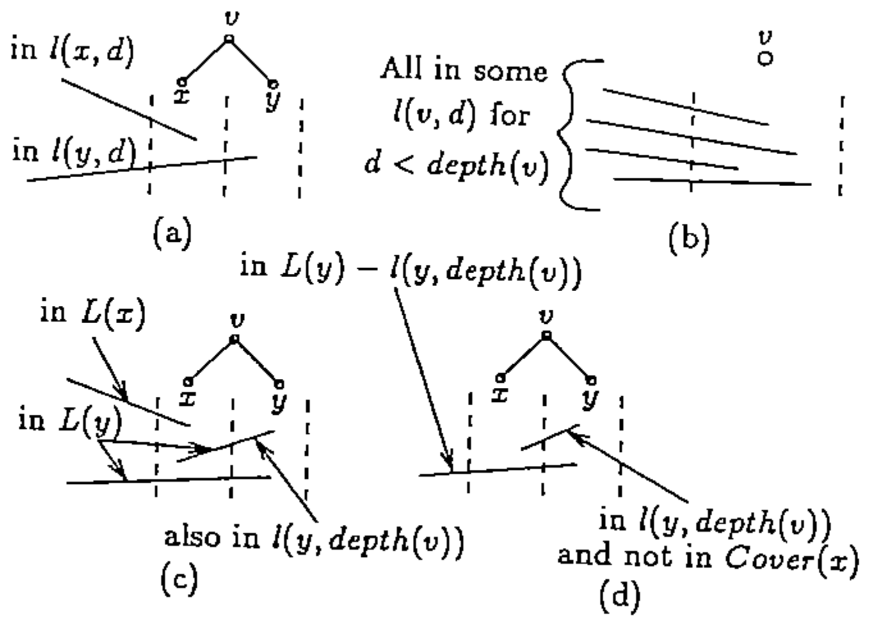

in $l(x,d)$
in $l(y,d)$
v
x
y
(a)
All in some $l(v,d)$ for $d < depth(v)$
(b)
in $L(y) - l(y, depth(v))$
in $L(x)$
in $L(y)$
also in $l(y, depth(v))$
(c)
in $l(y, depth(v))$ and not in $Cover(x)$
(d)

FIG. 5. *The plane-sweep tree equations.* (a) $l(v,d) = l(x,d) \cup l(y,d)$; (b) $L(v) = l(v,0) \cup l(v, depth(v)-1)$; (c) $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$; (d) $Cover(x) = L(y) - l(y, depth(v))$.

(6) $R(v) = (R(x) - r(x, depth(v))) \cup R(y)$,

(7) $Cover(x) = L(y) - l(y, depth(v))$,

(8) $Cover(y) = R(x) - r(x, depth(v))$.

Proof. The proof follows from the definitions. □

Lemma 5.1 essentially states that the lists l, r, L, R, and *Cover* for the nodes on a particular level of T can be defined in terms of lists for nodes on the next lower level of T. We could use this lemma and the parallel merge technique of Valiant [26], as implemented by Borodin and Hopcroft [10], to construct a sorted copy of each $Cover(v)$ list in $O(\log n \log\log n)$ time using n processors, improving on the previous bound of $O(\log^2 n)$ time using the same number of processors, due to Aggarwal et al. [1]. We can do even better, however, by exploiting the structure of the L and R lists. We describe how to do this below, in order to achieve a running time of $O(\log n)$ still using n processors. Before going into the details of the plane-sweep tree construction, we give a brief overview of the algorithm.

High-Level Description of Plane-Sweep Tree Construction.

The construction consists of the following four steps:

Step 1. Construct $l(v,d)$ and $r(v,d)$ for every $v \in T$. To implement this step, we perform $\lceil \log n \rceil$ generalized cascading merges in parallel (one for each d) based on (1) and (2) of Lemma 5.1 (starting with the leaf nodes of T). We implement this step in $O(\log n)$ time using n processors in total for all the merges.

Step 2. Let $d_v = depth(parent(v))$. Compute for each segment in $l(v, d_v)$ (respectively, $r(v, d_v)$) its predecessor segment in $L(v) - l(v, d_v)$ (respectively, $R(v) - r(v, d_v)$) based on (3) and (4). We do this, for each $v \in T$, by making d_v copies of $l(v, d_v)$ and $r(v, d_v)$, and merging $l(v, d_v)$ (respectively, $r(v, d_v)$) with all the $l(v,d)$ (respectively, $r(v,d)$) such that $d < d_v$. Note: we perform this step without actually constructing $L(v)$ or $R(v)$.

Step 3. Construct $L(v)$ and $R(v)$ for every $v \in T$. To implement this step we perform a generalized cascading merge procedure based on (5) and (6) and the information computed in Step 2 (starting with the leaf nodes of T). We never actually perform the set difference operations of (5) and (6), however. Instead, at the point in the merge that a segment in, say, $l(v, d_v)$, should be deleted we "change the identity" of that segment to its predecessor in $L(v) - l(v, d_v)$ (which we know from Step 2). That is, from this point on in the cascading merge this segment is indistinguishable from its predecessor in $L(v) - l(v, d_v)$. We show below that (i) the cascading merge will not be corrupted by doing this, (ii) the lists never contain too many duplicate entries (that would require us to use more than n processors), and (iii) after the merge completes, we can construct $L(v)$ and $R(v)$ for each node by removing duplicate segments in $O(\log n)$ time using n processors.

Step 4. Construct $Cover(v)$ for every $v \in T$ using (7) and (8) and the lists constructed in Step 3. The implementation of this step amounts to compressing each $L(v)$ (respectively, $R(v)$) so as to delete all the segments in $l(v, d_v)$ (respectively, $r(v, d_v)$), and then copying the list of segments so computed to the sibling node in T.

END OF HIGH-LEVEL DESCRIPTION.

We now describe how to perform each of these high-level steps.

5.2. Step 1: Constructing $l(v, d)$ and $r(v, d)$. We construct the $l(v, d)$ and $r(v, d)$ lists as follows. We make $\lceil \log n \rceil$ copies of T, and let $T(d)$ denote tree number d. Note that by our definition of T the space needed to store the "skeleton" of each $T(d)$ is $O(n/\log n)$. This of course results in a total of $O(n)$ space for all the $T(d)$'s. For each node v of $T(d)$ such that $depth(v) > d$ we wish to construct the lists $l(v, d)$ and $r(v, d)$, as given by (1) and (2) of Lemma 5.1. This implies that if we store $l(v, d)$ (respectively, $r(v, d)$) in every leaf node v of $T(d)$, then for any node $v \in T(d)$, $l(v, d)$ is precisely the sorted merge of the lists stored in the descendants of v. We start with the elements belonging to $l(v, d)$ (respectively, $r(v, d)$) stored (unsorted) in a list $A(v)$ for each leaf v in $T(d)$, and construct each $l(v, d)$ and $r(v, d)$ by the generalized cascading merge technique of Theorem 2.5 (using the $A(v)$'s as in the theorem). Note: since $l(v, d)$ and $r(v, d)$ are only defined for $d < depth(v)$, we only proceed up any tree $T(d)$ as far as nodes at depth $d + 1$, terminating the cascading merge at that point. We allocate $\lceil n/\log n \rceil + N_d$ processors to each tree $T(d)$, where N_d denotes the number of segments stored initially in the leaves of $T(d)$. Thus, since $\sum_{d=1}^{\lceil \log n \rceil} N_d = n$, we have shown how to construct all the $l(v, d)$ and $r(v, d)$ lists in $O(\log n)$ time and $O(n \log n)$ space using n processors.

5.3. Step 2: Computing predecessors. In Step 2 we wish to compute for each segment in the list $l(v, d_v)$ (respectively, $r(v, d_v)$) its predecessor segment in $L(v) - l(v, d_v)$ (respectively, $R(v) - r(v, d_v)$), where $d_v = depth(parent(v))$. Without loss of generality, we restrict our attention to the segments in $l(v, d_v)$ (the treatment for the segments in $r(v, d_v)$ is similar). Recall that (3) and (4) state that $L(v) = l(v, 0) \cup l(v, 1) \cup \cdots \cup l(v, d_v)$ and that $R(v) = r(v, 0) \cup r(v, 1) \cup \cdots \cup r(v, d_v)$. We make d_v copies of $l(v, d_v)$ and, using the merging procedure of Shiloach and Vishkin [25] or that of Bilardi and Nicolau [9], we merge a copy of $l(v, d_v)$ with each of $l(v, 0), \cdots, l(v, d_v - 1)$. This takes $O(\log n)$ time using $\lceil |L(v)|/\log n \rceil + \lceil d_v |l(v, d_v)|/\log n \rceil$ processors for each $v \in T$. Since (i) there are $O(n/\log n)$ nodes in each $T(d)$; (ii) each segment appears exactly once in some $l(v, d_v)$; and (iii) $\sum_{v \in T} |L(v)|$ is $O(n \log n)$, we can implement all these merges in parallel using n processors. Once we have completed all the merges, we assign a single processor to each segment s_i and compare the predecessors of s_i in $l(v, 0), \cdots, l(v, d_v - 1)$ so as to find the predecessor of s_i in $L(v) - l(v, d_v)$ $(= l(v, 0) \cup \cdots \cup l(v, d_v - 1))$. This amounts to $O(\log n)$ additional work for each s_i; thus Step 2 can be implemented in $O(\log n)$ time using n processors.

5.4. Step 3: Constructing $L(v)$ and $R(v)$. In this step we perform another cascading merge on T; this time to construct $L(v)$ and $R(v)$ for each $v \in T$ based on (5) and (6) of Lemma 5.1. Initially, we have $L(v)$ and $R(v)$ constructed only for the leaves. We then merge these lists up the tree based on (5) and (6) as in Theorem 2.5. The computation for this step differs from the cascading merge of Step 2, however, in that we need to be performing set-difference operations as well as list merges as we are cascading up the tree. Unfortunately, it is not clear how to perform these difference operations on-line any faster than $O(\log n)$ time per level, which would result in a running time that is $O(\log^2 n)$. We get around this problem by never actually performing the difference operations. That is, we do not actually delete segments from any lists. Instead, we change the identity of a segment s_i in say, $I(y, d_v)$, to its predecessor in $L(y) - I(y, d_v)$ when we are performing the merge as node v, where $y = rchild\ (v)$ (see Fig. 6). We do this instead of simply marking s_i as "deleted" in $L(v)$, because segments in $I(y, d_v)$ may not be comparable to segments in $L(x)$ (the list with which we wish to merge $L(y) - I(y, d_v)$). Simply marking a segment as being "deleted" could thus result in a processor attempting to compare two incomparable segments.

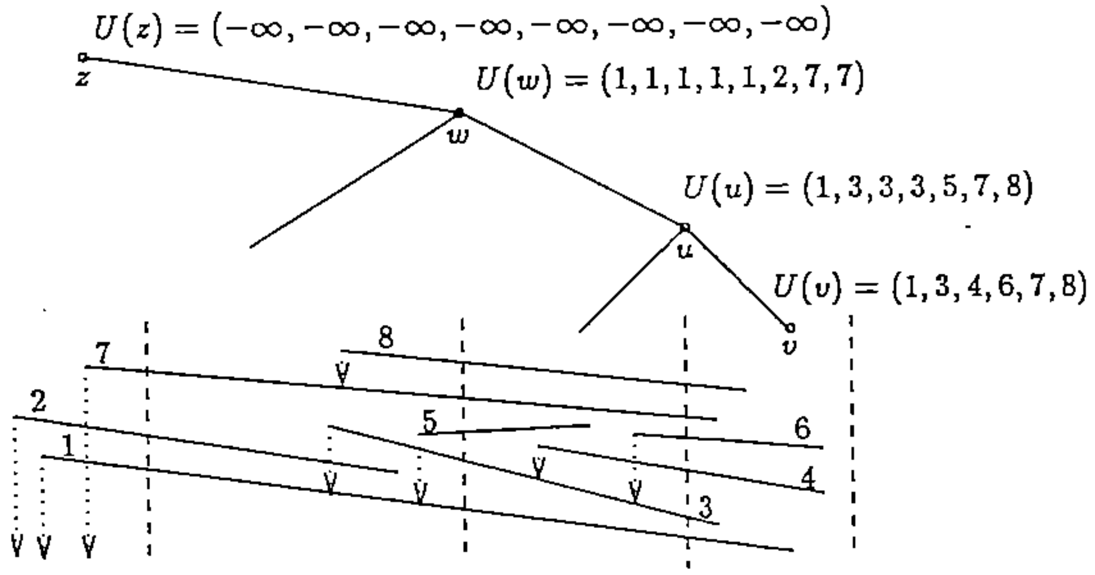

$U(z) = (-\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)$
z
$U(w) = (1,1,1,1,1,2,7,7)$
w
$U(u) = (1,3,3,3,5,7,8)$
u
$U(v) = (1,3,4,6,7,8)$
v
7
8
2
1
5
6
4
3

FIG. 6. *Segment identity changing during the cascading merge. We illustrate the way segment names change identity to that of their predecessor as we are performing the cascading merge. In this case we are constructing the $L(v)$'s. We denote the predecessor of each segment by a dotted arrow.*

Clearly, the fact that we change the identity of a segment in $I(y, d_v)$ to its predecessor in $L(y) - I(y, d_v)$ means that there will be multiple copies of some segments. This will not corrupt the cascading merge, however, because one of the properties of the "above" relation for segments is that all duplicate copies of a segment will be contiguous. Moreover, they will remain contiguous as the cascading merge proceeds up the tree. In addition, even though we will have multiple copies of segments in lists as they are merging up the tree, we can still implement this step with a total of n processors, because there will never be more items present in any $L(v)$ than the total number of items stored in the (leaf) descendants of v. At the end of this step we assign $\lceil |L(v)|/\log n \rceil$ processors to each v and compress out the duplicate entries in $L(v)$ in $O(\log n)$ time. Thus, we can construct $L(v)$ and $R(v)$ (compressed and sorted) for each $v \in T$ in $O(\log n)$ time using n processors.

5.5. Step 4: Constructing *Cover*(v). In this step we construct $Cover(v)$ for every v in T, based on (7) and (8) of Lemma 5.1. We implement this step by first compressing each $L(v)$ (respectively, $R(v)$) so as to delete all the segments in $I(v, d_v)$ (respectively,

$r(v, d_v)$), and then by copying the list of segments so computed to the sibling of v in T. This can all be done in $O(\log n)$ time using n processors.

Thus, summarizing the entire previous section, we have the following theorem.

THEOREM 5.2. *Given a set S of nonintersecting line segments in the plane, we can construct the plane-sweep tree T for S in $O(\log n)$ time using n processors in the* CREW PRAM *model, and this is optimal.*

Proof. We have already established the correctness and complexity bounds. To see that our construction is optimal, note that the plane-sweep tree requires $\Omega(n \log n)$ space. □

In the previous sections we assumed that segments did not intersect. Indeed, T is defined only if they do not intersect. We show in the next section that we can detect an intersection, if there is one, by constructing T while simultaneously checking for intersections.

5.6. The segment intersection detection problem. The problem we solve in this section is the following: given a set S of n line segments in the plane, determine if any two segments in S intersect. We begin by stating the conditions that we use to test for an intersection.

LEMMA 5.3 [1]. *The segments in S are nonintersecting if and only if we have the following for the plane-sweep tree T of S:*

(1) *For every $v \in T$ all the segments in $Cover(v)$ intersect* left (Π_v) *in the same order as they intersect* right (Π_v).

(2) *For every $v \in T$ no segment in $End(v)$ intersects any segment in $Cover(v)$.* □

Aggarwal et al. [1] used this lemma and their data structure to solve the intersection detection problem in $O(\log^2 n)$ time using n processors. Their method consisted of constructing the $Cover(v)$ lists independently of one another, basing comparisons on segment intersections with left (Π_v), and then testing for condition (1) by checking if each list $Cover(v)$ would be in the same order if they based comparisons on segment intersections with right (Π_v). If no intersection was detected by this step, then they tested for condition (2) by performing $O(n)$ multilocations of segment endpoints. This entire process took $O(\log^2 n)$ time using n processors.

We use this lemma by testing for condition (1) while we are constructing the plane-sweep tree for S (instead of waiting until after it has been built) and in so doing we achieve an $O(\log n)$ time bound for this test (since our construction takes only $O(\log n)$ time). We test condition (2) in the same fashion as Aggarwal et al., that is, by doing $O(n)$ multilocations after the plane-sweep tree has been built. Since with our data structure the multiplications can all be performed in $O(\log n)$ time, the entire intersection-detection process takes $O(\log n)$ time using n processors.

Since we do not construct the $Cover(v)$ lists independently of one another, but instead construct them by performing several cascading merges, we must be very careful in how we base segment comparisons, and in how we test for condition (1). For if two segments intersect, then determining which segment is above the other depends on the vertical line upon which we base the comparison.

We consider each step of the construction in turn, beginning with Step 1. Recall that in Step 1 we construct all the $l(v, d)$ and $r(v, d)$ lists for each $v \in T$. In the following lemma we show that if we base segment comparisons on appropriate vertical lines, Step 1 can be performed just as before.

LEMMA 5.4. *Let $v \in T$ and $0 \leqq d < depth(v)$ be given, and let s_1 and s_2 be two segments such that $s_1 \in l(w, d)$ and $s_2 \in l(z, d)$ (or $s_1 \in r(w, d)$ and $s_2 \in r(z, d)$), where $w, z \in Desc(v)$. Then $dom(s_1) = dom(s_2)$.*

Proof. Let $v \in T$ and $0 \leqq d \leqq \lceil \log n \rceil$ be given. Recall that $l(v, d)$ (respectively, $r(v, d)$) is defined to be the list of all segments in $L(v)$ ($R(v)$) that have a dominator node at depth d in T. Note that the dominator node for any segment s_i in $l(w, d)$, $r(w, d)$, $l(z, d)$, or $r(z, d)$, where $w, z \in Desc(v)$, must be an ancestor of v, since $d < depth(v)$ and, by definition, $s_i \in End(v)$ and $s_i \in End(dom(s_i))$. There is only one node that is an ancestor of v and is at depth d in T. □

Thus, we can perform the merges based on (1) and (2) of Lemma 5.1 (e.g., $l(v, d) = l(x, d) \cup l(y, d)$) by basing all segment comparisons on the intersection of the segments with the vertical boundary separating the two children of their dominator node. That is, if s_1 and s_2 are two segments to be compared in Step 1, then we say that s_1 is "above" s_2 if and only if the intersection of s_1 with L is above the intersection of s_2 with L, where L is the vertical boundary line separating the two children of $dom(s_1)$ $(= dom(s_2))$.

In Step 2 we computed for each segment in $l(v, d_v)$ (respectively, $r(v, d_v)$) its predecessor segment in $L(v) - l(v, d_v)$ (respectively, $R(v) - r(v, d_v)$), where $d_v = depth(parent(v))$. Recall that we did this by merging $l(v, d_v)$ with each of $l(v, 0), \cdots, l(v, d_v - 1)$. A similar computation was done for $r(v, d_v)$; without loss of generality, we concentrate on the computation involving $l(v, d_v)$. Also recall that all the segments in $l(v, 0), \cdots, l(v, d_v)$ belong to $L(v)$; hence they intersect left (Π_v). After Step 1 finishes, each list $l(v, d)$ will be sorted based on segment intersections with the vertical boundary line separating the two children of the ancestor of v at depth d (the dominator of all the segments in $l(v, d)$). In $O(\log n)$ time we can check if this order is preserved in each of $l(v, 0), \cdots, l(v, d_v)$ if we base segment intersections on left (Π_v), instead. If the order changed in any $l(v, d)$, then we have detected an intersection, and we are done. Otherwise, we proceed with Step 2 just as before, basing comparisons on segment intersections with left (Π_v).

In Step 3 we performed a cascading merge up the tree T, constructing $L(v)$ and $R(v)$ for every node $v \in T$. Recall that this cascading merge was based on (5) and (6) of Lemma 5.1 (e.g., $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$). Let us concentrate on the testing procedure for the $L(v)$'s, since the method for the $R(v)$'s is similar. Initially, let us start with each $L(v)$ constructed at the leaves of T sorted by segment intersections with left (Π_v). Thus, before we perform the merge based on the equation $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$, we must first check to see if the segments in the sample of $L(y) - l(y, depth(v))$ (to be merged with the sample of $L(x)$) have the same order independent of whether comparisons are based on segment intersections with left (Π_y) or left (Π_v). Unfortunately, to do this completely would require $O(\log n)$ time at every level of the tree, resulting in an $O(\log^2 n)$ time algorithm. So, instead of broadcasting at each level whether an intersection has occurred or not, we cascade that information up along with the merges. More precisely, before doing the merge at a node v, we test if every consecutive pair of items in the sample of $L(y) - l(y, depth(v))$ would remain in the same order independent of whether comparisons were based on segment intersections with left (Π_y) or with left (Π_v). If we detect that an intersection has occurred, then we will have two elements that are out of order. If this should occur, we replace both items by the distinguished symbol S. Then, as the merges continue up the tree, any time we compare an item with S, we replace that item with S and proceed just as before. This keeps the merging process consistent, and after the cascading merge completes we can then in $O(\log n)$ time test if any of the items in any $L(v)$ or $R(v)$ contain a S symbol, by assigning $\lceil |L(v)|/\log n \rceil$ processors to each $v \in T$.

In Step 4 we constructed $Cover(v)$ for each $v \in T$. Recall that we did this by simply performing compressing and copying operations on lists constructed in Step 3. Thus, assuming that no intersection was detected in Step 3, we can perform Step 4 just as before. After Step 4 completes we can assign $\lceil |Cover(v)|/\log n \rceil$ processors to each $v \in T$ and test condition (1) directly in $O(\log n)$ time, checking if the items in $Cover(v)$ would be in the same order independent of whether comparisons were based on left (Π_v) or on right (Π_v).

If we have not discovered an intersection after Step 4, then the only computation left is to perform fractional cascading on the plane-sweep tree T, constructing a fractional cascading data structure $\hat{T}$. In directing all the edges in T to the root, and performing the fractional cascading preprocessing on T to construct $\hat{T}$, we associate a vertical strip with each node in $\hat{T}$. Since T is a tree then $\hat{T}$ is also a tree (recall the preprocessing step of the fractional cascading algorithm). For each node v in $\hat{T}$ if v is also in T, then we take Π_v for v in $\hat{T}$ to be the same as Π_v for v in T. Then, for any v that is in $\hat{T}$ but not in T (i.e., v is a gateway or a node in a fan-in or fan-out tree), we take Π_v to be the union of all the vertical strips that are descendents of v. Every time we perform the per-stage merge computation we compare adjacent entries in each bridge list $B(v)$ to see if they would be in the same order independent of whether we base comparisons on segment intersections with left (Π_v) or right (Π_v). If we detect that two adjacent segments intersect, then we replace both with the special symbol \$. Then, as before, any time we compare a segment with \$ we replace that segment by \$. Finally, when we complete the computation for Step 5, we assign $\lceil |B(v)|/\log n \rceil$ processors to each node v and check if there are any \$ symbols present in any $B(v)$ list.

If there are no intersections detected during the fractional cascading, then we perform $O(n)$ multilocations of all the segment endpoints as in [1] to test condition (2). Let p be an endpoint of some segment s_i. We perform the multilocation of p in the plane-sweep tree for S, and check if s_i intersects the segment directly above p or the segment directly below p in each $Cover(v)$ list such that $p \in \Pi_v$. This test is sufficient, since if s_i intersects any segment in $Cover(v)$, it must intersect the segment directly above p in $Cover(v)$ or the segment directly below p in $Cover(v)$. Thus, by performing a multilocation for p, we can test for condition (2) in $O(\log n)$ time using n processors. We summarize this discussion in the following theorem.

THEOREM 5.5. *Given a set of n line segments in the plane, we can detect if any two intersect in* $O(\log n)$ *time using n processors in the* CREW PRAM *model.* □

So far in this paper we have restricted ourselves to applications involving line segments. In the next section we show how to apply the cascading divide-and-conquer technique to other geometric problems as well.

6. Cascading with labeling functions.

In this section we show how to solve several different geometric problems by combining the merging procedure of § 2 with divide-and-conquer strategies based on merging lists with labels defined on their elements. For most of these problems our divide-and-conquer approach gives an efficient sequential alternative to the known sequential algorithms (which use the plane-sweeping paradigm) and gives rise to efficient parallel algorithms as well. We begin with the three-dimensional maxima problem.

6.1. The three-dimensional maxima problem.

Let $V = \{p_1, p_2, \cdots, p_n\}$ be a set of points in $\Re^3$. For simplicity, we assume that no two input points have the same x (respectively, y, z) coordinate. We denote the x, y, and z coordinates of a point p by $x(p)$, $y(p)$, and $z(p)$, respectively. We say that a point p_i *one-dominates* another point p_j if $x(p_i) > (p_j)$, *two-dominates* p_j *if* $x(p_i) > x(p_j)$ and $y(p_i) > y(p_j)$, and *three-dominates*

p_i if $x(p_i) > x(p_j)$, $y(p_i) > y(p_j)$, and $z(p_i) > z(p_j)$. A point $p_i \in V$ is said to be a *maximum* if it is not three-dominated by any other point in V. The three-dimensional maxima problem, then, is to compute the set, M, of maxima in V. We show how to solve the three-dimensional maxima problem efficiently in parallel in the following algorithm.

Our method is based on cascading a divide-and-conquer strategy in which the subproblem merging step involves the computation of two labeling functions for each point. The labels we use are motivated by the optimal sequential plane-sweeping algorithm of Kung, Luccio, and Preparata [20]. Specifically, for each point p_i we compute the maximum z-coordinate from among all points that one-dominate p_i and use that label to also compute the maximum z-coordinate from among all points that two-dominate p_i. We can then test if p_i is a maximum point by comparing $z(p_i)$ to this latter label. The details follow.

Without loss of generality, we assume the input points are given sorted by increasing y-coordinates, i.e., $y(p_i) < y(p_{i+1})$, since if they are not given in this order we can sort them in $O(\log n)$ time using n processors [13]. Let T be a complete binary tree with leaf nodes $v_1, v_2, \cdots, v_n$ (in this order). In each leaf node v_i we store the list $B(v_i) = (-\infty, p_i)$, where $-\infty$ is a special symbol such that $x(-\infty) < x(p_j)$ and $y(-\infty) < y(p_j)$ for all points p_j in V. Initializing T in this way can be done in $O(\log n)$ time using n processors. We then perform a generalized cascading merge from the leaves of T as in Theorem 2.5, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). Using the notation of § 2, we let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store two labels: $zod(p_i, v)$ and $ztd(p_i, v)$, where $zod(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that one-dominate p_i, and $ztd(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that two-dominate p_i. Initially, zod and ztd labels are only defined for the leaf nodes of T. That is, $zod(p_i, v_i) = ztd(p_i, v_i) = -\infty$ and $zod(-\infty, v_i) = ztd(-\infty, v_i) = z(p_i)$ for all leaf nodes v_i in T (where $U(v_i) = (-\infty, p_i)$). In order to be more explicit in how we refer to various ranks, we let $\text{pred}(p_i, v)$ denote the predecessor of p_i in $U(v)$ (which would be $-\infty$ if the x-coordinates of the points in $U(v)$ are all larger than $x(p_i)$) (see Fig. 7). As we are performing the cascading merge, we update the labels zod and ztd based on the equations in the following lemma.

LEMMA 6.1. *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following*:

$$(9)\qquad zod(p_i, v) = \begin{cases} \max\{zod(p_i, u), zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u), \\ \max\{zod(\text{pred}(p_i, u), u), zod(p_i, w)\} & \text{if } p_i \in U(w), \end{cases}$$

$$(10)\qquad ztd(p_i, v) = \begin{cases} \max\{ztd(p_i, u), zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u), \\ ztd(p_i, w) & \text{if } p_i \in U(w). \end{cases}$$

Proof. Consider (9). If $p_i \in U(u)$, then every point that one-dominates p_i's predecessor in $U(w)$ also one-dominates p_i, since p_i's predecessor in $U(w)$ is the point with largest x-coordinate less than $x(p_i)$ (or $-\infty$ if every point in $U(w)$ has larger x-coordinate than p_i). Thus $zod(p_i, v)$ is the maximum of $zod(p_i, u)$ and $zod(\text{pred}(p_i, w), w)$ in this case. The case when $p_i \in U(w)$ is similar. Next, consider (10). We know that every point in $U(w)$ has y-coordinate greater than every point in $U(u)$, by our construction of T. Therefore, if $p_i \in U(u)$, then every point in $U(w)$ that one-dominates p_i's predecessor in $U(w)$ must two-dominate p_i. Thus, $ztd(p_i, v)$ is the

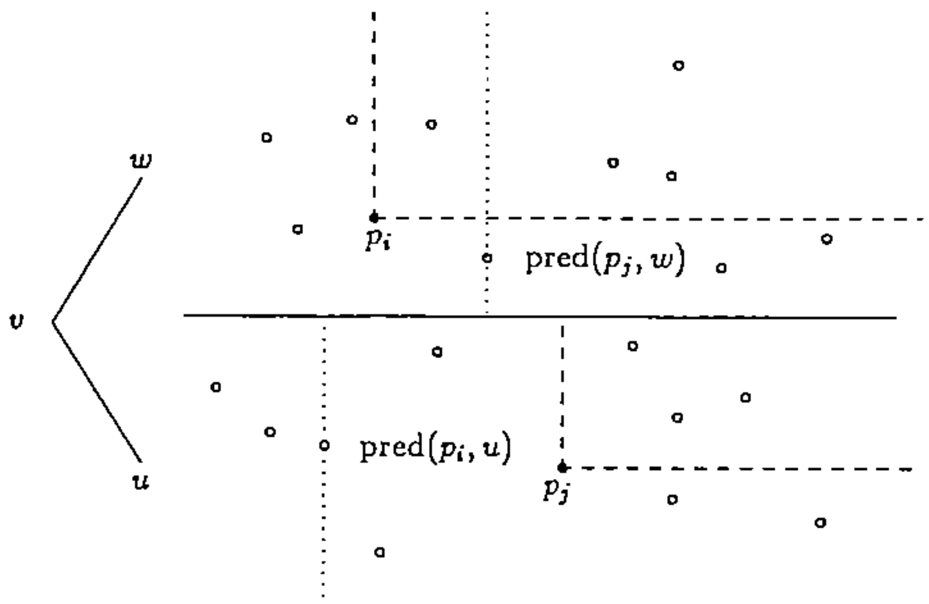


FIG. 7. *The combining step for three-dimensional maxima. Points to the right of the dotted line one-dominate p_i (respectively, p_j), and points enclosed in the dashed lines two-dominate p_i (p_j).*

maximum of $ztd(p_i, u)$ and $zod(\text{pred}(p_i, w), w)$. On the other hand, if $p_i \in U(w)$ then no point in $U(u)$ can two-dominate p_i; thus, $ztd(p_i, v) = ztd(p_i, w)$. □

We use these equations during the cascading merge to maintain the labels for each point. By Lemma 6.1, when v becomes full (and we have $U(u)$, $U(w)$, and $U(u) \cup U(w)$ available), we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of the cascading merge algorithm, even with these additional label computations, is still $O(\log n)$ using n processors. Moreover, after v's parent becomes full we no longer need $U(v)$, and can deallocate the space it occupies, resulting in an $O(n)$ space algorithm, as outlined in § 2. After we complete the merge, and have computed $U(root(T))$, along with all the labels for the points in $U(root(T))$, note that a point $p_i \in U(root(T))$ is a maximum if and only if $ztd(p_i, root(T)) \leqq z(p_i)$ (there is no point that two-dominates p_i and has z-coordinate greater than $z(p_i)$). Thus, after completing the cascading merge we can construct the set of maxima by compressing all the maximum points into one contiguous list using a simple parallel prefix computation. We summarize in the following theorem.

THEOREM 6.2. *Given a set V of n points in $\Re^3$, we can construct the set M of maxima points in V in $O(\log n)$ time and $O(n)$ space using n processors in the* CREW PRAM *model, and this is optimal.*

Proof. We have established the correctness and complexity bounds for parallel three-dimensional maxima finding in the discussion above. Kung, Luccio, and Preparata [20] have shown that this problem has an $\Omega(n \log n)$ sequential lower bound (in the comparison model). Thus, we can do no better than $O(\log n)$ time using n processors. □

It is worth noting that we can use roughly the same method as that above as the basis step of a recursive procedure for solving the general k-dimensional maxima problem. The resulting time and space complexities are given in the following theorem. We state the theorem for $k \geqq 3$ (since the two-dimensional maxima problem can easily be solved in $O(\log n)$ time and $O(n)$ space by a sorting step followed by a parallel prefix step).

THEOREM 6.3. *For $k \geqq 3$ the k-dimensional maxima problem can be solved in $O((\log n)^{k-2})$ time using n processors in the* CREW PRAM *model.*

Proof. The method is a straightforward parallelization of the algorithm by Kung, Luccio, and Preparata [20], using a procedure very similar to that described above as the basis for the recursion. We leave the details to the reader. □

Next, we address the two-set dominance counting problem. We also show how the multiple range-counting problem and the rectilinear segment intersection counting problem can be reduced to two-set dominance problems efficiently in parallel.

6.2. The two-set dominance counting problem. In the two-set dominance counting problem we are given a set $A = \{q_1, q_2, \cdots, q_m\}$ and a set $B = \{r_1, r_2, \cdots, r_l\}$ of points in the plane, and wish to know for each point r_i in B the number of points in A that are two-dominated by r_i. For simplicity, we assume that the points have distinct x (respectively, y) coordinates. Our approach to this problem is similar to that of the previous subsection, in that we will be performing a cascading merge procedure while maintaining two labeling functions for each point. In this case the labels maintain for each point p_i(from A or B) how many points of A are one-dominated by p_i and also how many points of A are two-dominated by p_i. As in the previous solution, the first label is used to maintain the second. The details follow.

Let $Y = \{p_1, p_2, \cdots, p_{l+m}\}$ be the union of A and B with the points listed by increasing y-coordinate, i.e., $y(p_i) < y(p_{i+1})$. We can construct Y in $O(\log n)$ time using n processors [13], where $n = l + m$. Our method for solving the two-set dominance counting problem is similar to the method used in the previous subsection. As before, we let T be a complete binary tree with leaf nodes $v_1, v_2, \cdots, v_n$, in this order, and in each leaf node v_i we store the list $U(v_i) = (-\infty, p_i)$ ($-\infty$ still being a special symbol such that $x(-\infty) < x(p_i)$ and $y(-\infty) < y(p_i)$ for all points p_i in Y). We then perform a generalized cascading merge from the leaves of T as in Theorem 2.5, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). We let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinate. For each point p_i in $U(v)$ we store two labels: $nod(p_i, v)$ and $ntd(p_i, v)$. The label $nod(p_i, v)$ is the number of points in $U(v)$ that are in A and are one-dominated by p_i, and the label $ntd(p_i, v)$ is the number of points in $U(v)$ that are in A and are two-dominated by p_i. Initially, the *nod* and *ntd* labels are only defined for the leaf nodes of T. That is, $nod(p_i, v_i) = nod(-\infty, v_i) = ntd(p_i, v_i) = ntd(-\infty, v_i) = 0$. For each $p_i \in Y$ we define the function $\chi_A(p_i)$ as follows: $\chi_A(p_i) = 1$ if $p_i \in A$, and $\chi_A(p_i) = 0$ otherwise. (We also use pred (p_i, v) to denote the predecessor of p in $U(v)$. As we are performing the cascading merge, we update the labels *nod* and *ntd* based on the equations in the following lemma (see Fig. 8).

LEMMA 6.4. *Let p_i be an element of $U(v)$ and let $u = lchild\ (v)$ and $w = rchild\ (v)$. Then we have the following*:

$$(11)\quad nod(p_i, v) = \begin{cases} nod(p_i, u) + nod(\text{pred}\,(p_i, w), w) + \chi_A(\text{pred}\,(p_i, w)) & \textit{if } p_i \in U(u), \\ nod(\text{pred}\,(p_i, u), u) + nod(p_i, w) + \chi_A(\text{pred}\,(p_i, u)) & \textit{if } p_i \in U(w), \end{cases}$$

$$(12)\quad ntd(p_i, v) = \begin{cases} ntd(p_i, u) & \textit{if } p_i \in U(u), \\ nod(\text{pred}\,(p_i, u), u) + ntd(p_i, w) + \chi_A(\text{pred}\,(p_i, u)) & \textit{if } p_i \in U(w). \end{cases}$$

Proof. Consider (11). For any point $p_i \in U(u)$ the number of points one-dominated by p_i is equal to the number of points in $U(u)$ that are in A and one-dominated by

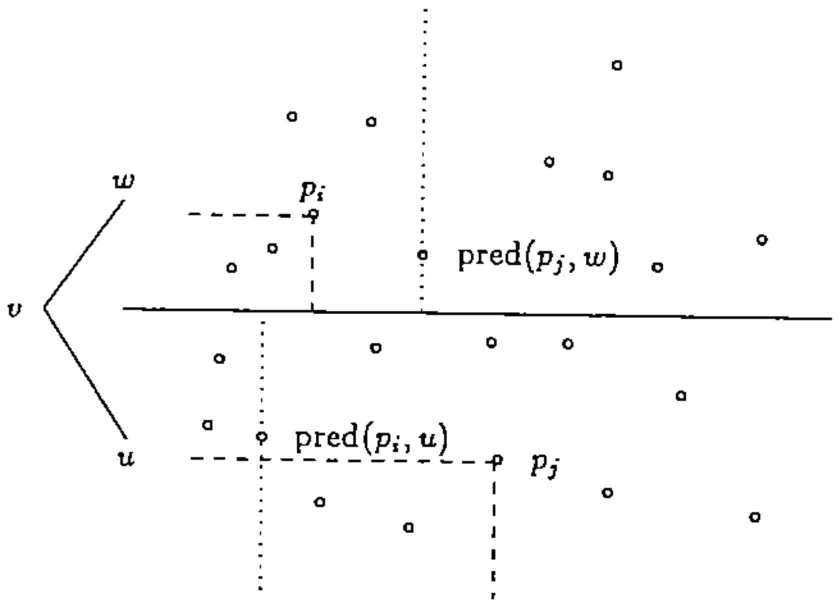


FIG. 8. *The combining step for dominance counting. Points to the left of the dotted line are one-dominated by p_i (respectively, p_j), and points enclosed in dashed lines are two-dominated by p_i (p_j).*

p_i, plus the number of points in $U(w)$ that are in A and one-dominated by pred(p_i, w), plus one if pred(p_i, w) is in A (since the predecessor of p_i is one-dominated by p_i). Thus, we have the equation for the case when $p_i \in U(u)$. The case when $p_i \in U(w)$ is similar. Next, consider (12). By our construction, every point in $U(u)$ has y-coordinate less than the y-coordinate of every point in $U(w)$. So if $p_i \in U(u)$, then the number of points in $U(v)$ that are in A and are two-dominated by p_i is precisely $ntd(p_i, u)$, since p_i cannot two-dominate any points in $U(w)$. If $p_i \in U(w)$, on the other hand, then the number of points in $U(v)$ that are in A and two-dominated by p_i is the number of points in $U(u)$ that are in A and one-dominated by pred(p_i, u), plus the number of points in $U(w)$ that are in A and two-dominated by p_i, plus one if pred(p_i, u) is in A. This is exactly (12) in this case. □.

By Lemma 6.4, when v becomes full (and we have $U(u)$, $U(w)$, and $U(v) = U(u) \cup U(w)$ available), we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of the cascading merge algorithm, even with these additional label computations, is still $O(\log n)$ using n processors. After we complete the merge, and have computed $U(\text{root}(T))$, along with all the labels for the points in $U(\text{root}(T))$, then we are done. We summarize in the following theorem.

THEOREM 6.5. *Given a set A of l points in the plane and a set B of m points in the plane, we can compute for each point p in B the number of points in A two-dominated by p in $O(\log n)$ time and $O(n)$ space using n processors in the* CREW PRAM *model, where $n = l + m$, and this is optimal.*

Proof. The correctness and complexity bounds should be apparent from the discussion above. To prove the lower bound note that the two-dimensional maxima problem can be reduced to dominance counting in $O(1)$ time using n processors (see [17]). Since the maxima problem has an $\Omega(n \log n)$ lower bound [20] in the comparison model, we conclude that we can do no better than $O(\log n)$ time using n processors in the CREW PRAM model. □

There are a number of other problems that can be reduced to two-set dominance counting. We mention two here, the first being the multiple range-counting problem:

given a set V of l points in the plane and a set R of m isothetic rectangles (ranges) the multiple range-counting problem is to compute the number of points interior to each rectangle.

COROLLARY 6.6. *Given a set V of l points in the plane and a set R of m isothetic rectangles, we can solve the multiple range-counting problem for V and R in $O(\log n)$ time and $O(n)$ space using n processors, where $n = l + m$.*

Proof. Let $d(p)$ be the number of points in V two-dominated by a point p. Edelsbrunner and Overmars [15] have shown that counting the number of points interior to a rectangle can be reduced to dominance counting. That is, given a rectangle $r = (p_1, p_2, p_3, p_4)$ (where vertices are listed in counterclockwise order starting with the upper right-hand corner), the number of points in V interior to r is $d(p_1) - d(p_2) + d(p_3) - d(p_4)$. Therefore, it suffices to solve the two-set dominance counting problem. □

Another problem that reduces to two-set dominance counting is rectilinear segment intersection counting: given a set S of n rectilinear line segments in the plane, determine for each segment the number of other segments in S that intersect it.

COROLLARY 6.7. *Given a set S of n rectilinear line segments in the plane, we can determine for each segment the number of other segments in S that intersect it in $O(\log n)$ time and $O(n)$ space using n processors in the* CREW PRAM *model.*

Proof. Let U_1 (U_2) be the set of left (right) endpoints of horizontal segments, and let $d_1(p)$ ($d_2(p)$) denote the number of points in U_1 (U_2) two-dominated by p. For any vertical segment s, with upper endpoint p and lower endpoint q, the number of horizontal segments that intersect s is $d_1(p) - d_1(q) + d_2(q) - d_2(p)$. This is because $d_1(p) - d_1(q)$ (respectively, $d_2(p) - d_2(q)$) counts the number of horizontal segments with a left (right) endpoint to the left of s and y-coordinate in the interval $[y(q), y(p)]$. Thus, $d_1(p) - d_1(q) - (d_2(p) - d_2(q))$ counts the number of horizontal segments with left endpoint to the left of s, right endpoint to the right of s, and y-coordinate in the interval $[y(q), y(p)]$ (i.e., the set of horizontal segments that intersect s). □

The final problem we address at is visibility from a point.

6.3. The visibility from a point problem. Given a set of line segments $S = \{s_1, s_2, \cdots, s_n\}$ in the plane that do not intersect, except possibly at endpoints, and a point p, the visibility from a point problem is to determine the part of the plane that is visible from p assuming every s_i is opaque. Intuitively, we can think of the point p as a specular light source, the segments as walls, and the problem to determine all the parts of the plane that are illuminated. We can use the cascading divide-and-conquer technique to solve this problem in $O(\log n)$ time and $O(n)$ space using n processors. Without loss of generality, we assume that the point p is at negative infinity below all the segments. The algorithm is essentially the same if p is a finite point, except that the notion of segment endpoints being ordered by x-coordinate is replaced by the notion that they are ordered radially around p. In other words, it suffices to compute the *lower envelope* of the n segments to give a method for computing the visibility from a point. For simplicity of expression, we also assume that the x-coordinates of the endpoints are distinct.

In the previous two subsections the set of objects consisted of points, but in the visibility problem we are dealing with line segments. The method is slightly different in this case. In this case, we store the segments in the leaves of a binary tree and perform a cascading merge of the x-coordinates of intervals of the x-axis determined by segment endpoints. We maintain a single label for each interval which represents the segment which is visible from $-\infty$ on that interval. The details follow.

Let T be a complete binary tree with leaf nodes $v_1, v_2, \cdots, v_n$ ordered from left to right. We associate the segment s_i with the leaf v_i and at v_i store the list $U(v_i) = (-\infty, p_1, p_2)$, where p_1 and p_2 are the two endpoints of s_i, with $x(p_1) < x(p_2)$, and $-\infty$ is defined such that $x(-\infty) < x(p)$ and $y(-\infty) < y(p)$ for all points p. We then perform a generalized cascading merge from the leaves of T as in Theorm 2.5, basing comparisons on increasing x-coordinates of the points. For each internal node v we let $U(v)$ denote an array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store a label $vis(p_i, v)$ which stores the segment with endpoints in $U(v)$ that is visible in the interval $(x(p_i), x(succ(p_i, v)))$, where $succ(p_i, v)$ denotes the successor of p_i in $U(v)$ (based on x-coordinates). Initially, the vis labels are only defined for the leaf nodes of T. That is, if $U(v) = (-\infty, p_1, p_2)$, where $s_i = p_1 p_2$, then $vis(-\infty) = +\infty$, $vis(p_1) = s_i$, and $vis(p_2) = +\infty$. We use pred (p_i, v) to denote the predecessor of p_i in $U(v)$. As we are performing the cascading merge, we update the vis labels based on the equation in the following lemma (see Fig. 9).

LEMMA 6.8. *Let p_i be an element of $U(v)$ and let $u = lchild\ (v)$ and $w = rchild\ (v)$. Then we have the following (if two segments s_i and s_j are comparable by the "above"*

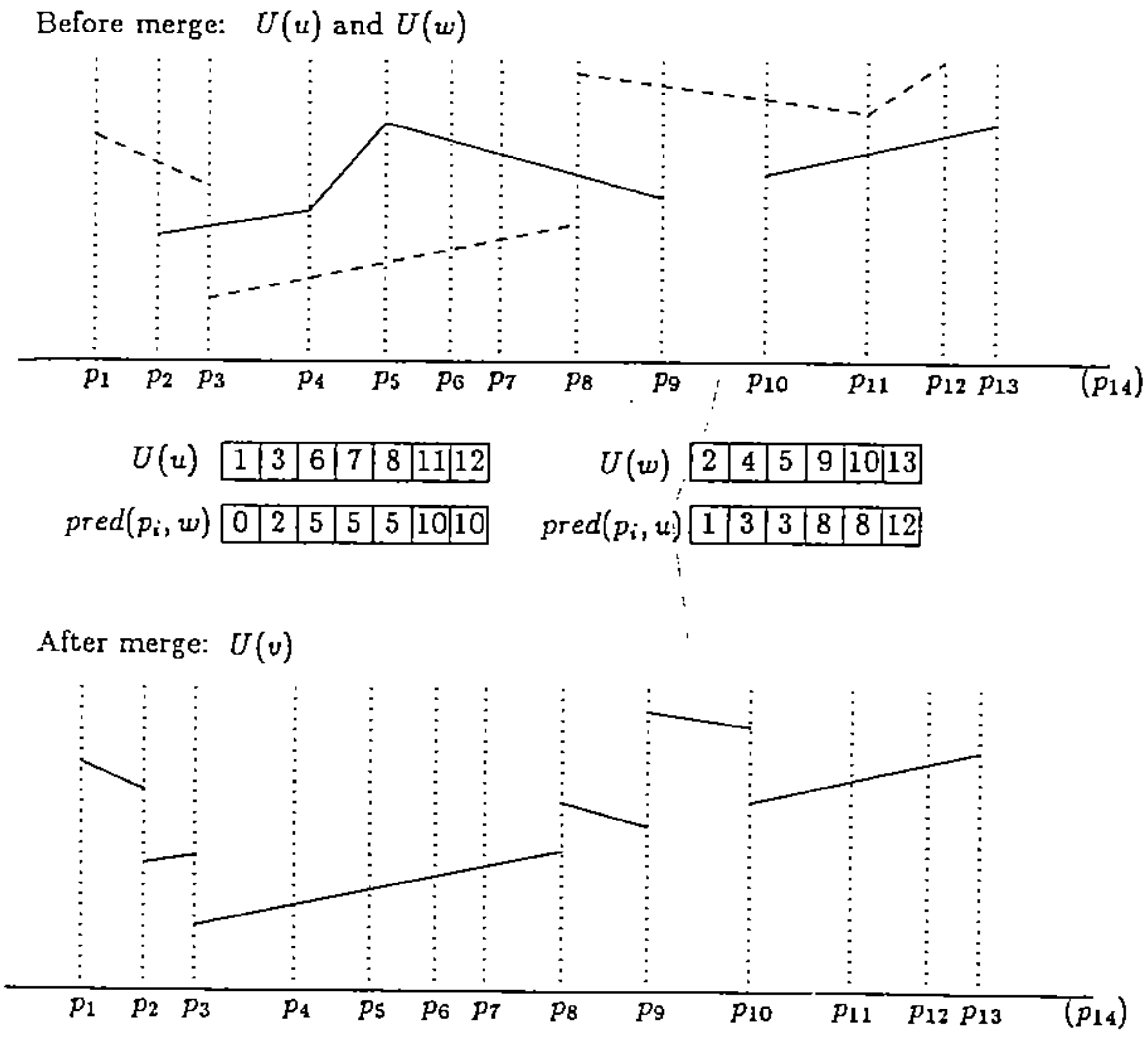

Before merge: $U(u)$ and $U(w)$
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} (p_{14})
$U(u)$ 1 3 6 7 8 11 12
$pred(p_i, w)$ 0 2 5 5 5 10 10
$U(w)$ 2 4 5 9 10 13
$pred(p_i, u)$ 1 3 3 8 8 12
After merge: $U(v)$
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} (p_{14})

FIG. 9. *An example of visibility merging. The dashed segments correspond to the visible region for $X(u)$ and the solid segments correspond to the visible region for $X(w)$. For simplicity, we store the pointers* pred (p_i, u) *and* pred (p_i, w) *in arrays and denote each point p_i by its index i. Note that points are never removed, even if the same segment defines the visible region for many consecutive intervals (e.g., p_3 through p_7).*

relation, then we let $\min\{s_i, s_j\}$ *denote the lower of the two*):

$$vis(p_i, v) = \begin{cases} \min\{vis(p_i, u), vis(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u), \\ \min\{vis(\text{pred}(p_i, u), u), vis(p_i, w)\} & \text{if } p_i \in U(w). \end{cases}$$

Proof. If we restrict our attention to the segments with an endpoint in $U(v)$, then for any point $p_i \in U(u)$ the segment visible (from $-\infty$) on the interval $(x(p_i), x(succ(p_i, v)))$ is the minimum of the segment visible on the interval $(x(p_i), x(succ(p_i, u)))$ and the segment that is visible on the interval $(x(\text{pred}(p_i, w)), x(succ(\text{pred}(p_i, w), w)))$. This is because the interval $(x(p_i), x(succ(p_i, v)))$ is exactly the intersection of the interval $(x(p_i), x(succ(p_i, u)))$ and the interval $(x(\text{pred}(p_i, w)), x(succ(\text{pred}(p_i, w), w)))$, and there is no segment in $U(v)$ with an endpoint interior to the interval $(x(p_i), x(succ(p_i, v)))$. Thus, $vis(p_i, v)$ is equal to minimum of $vis(p_i, u)$ and $vis(\text{pred}(p_i, w), w)$. The case when $p_i \in U(v)$ is similar. □

By Lemma 6.8, after merging the lists $U(u)$ and $U(w)$, we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of this generalized cascading merge algorithm is still $O(\log n)$ using n processors. After we complete the merge and have computed $U(root(T))$, along with all the vis labels for the points in $U(root(T))$, then we can compress out duplicate entries in the list $(vis(p_1, root(T)), vis(p_2, root(T)), \cdots, vis(p_{2n}, root(T)))$ using a parallel prefix computation to construct a compact representation of the visible portion of the plane. We summarize in the following theorem.

THEOREM 6.9. *Given a set S of n nonintersecting segments in the plane, we can find the lower envelope of S in* $O(\log n)$ *time and* $O(n)$ *space using n processors in the* CREW PRAM *model, and this is optimal.*

Proof. The correctness and complexity bounds follow from the discussion above. Since we require that the points in the description of the lower envelope be given by increasing x-coordinates, we can reduce sorting to this problem, and thus can do no better than $O(\log n)$ time using n processors. □

7. EREW PRAM implementations. In this section we briefly note that the same techniques as employed by Cole in [13] to implement his merging procedure in the EREW PRAM model (no simultaneous reads) can be applied to our algorithms for generalized merging, fractional cascading, constructing the plane-sweep tree, three-dimensional maxima, two-set dominance counting, and visibility from a point, resulting in EREW PRAM algorithms for these problems. Apparently, we cannot apply his techniques to our algorithms for trapezoidal decomposition and segment intersection detection, however, since our algorithms for these problems explicitly use concurrent reads (in the multilocation steps).

Applying his techniques to our algorithms results in EREW PRAM algorithms with the same asymptotic bounds as the ones presented in this paper, except that the space bounds for the problems addressed in § 6 all become $O(n \log n)$. The reason that his techniques increase the space complexity of these problems is because of our use of labeling functions. Specifically, it is not clear how to perform the merges on-line and still update the labels in $O(1)$ time after a node becomes full. This is because a label whose value changes on level l may have to be broadcasted to many elements in level $l-1$ to update their labels, which would require $\Omega(\log n)$ time in this model if there were $O(n)$ such elements.

We can get around the problem arising from the labeling functions, however. For the three-dimensional maxima problem and the two-set dominance counting problem, we separate the computation of the $U(v)$ lists and computation of the labeling functions into two separate steps, rather than "dovetailing" the two computations as before. Each of the labeling functions we used for these two problems can be redefined so as to be EREW-computable. Specifically, the label for an element p in $U(v)$, on level l, can be expressed in terms of a label $pref(p, v)$ and a label $up(p, v)$, where $pref(p, v)$ can be computed by performing a parallel prefix computation [21], [22] in $U(v)$ and $up(p, v)$ can be defined in terms of $pref(pred(p, lchild(v)), lchild(v))$, $pref(pred(p, rchild(v)), rchild(v))$, and the up label p had on level $l+1$ (say, in $U(rchild(v))$ if $p \in U(rchild(v))$). In particular, for the three-dimensional maxima problem $pref(p, v) = zod(p, v)$ and $up(p, v) = ztd(p, v)$, and for the two-set dominance counting problem $pref(p, v) = nod(p, v)$ and $up(p, v) = ntd(p, v)$. We can compute all the $pref(p, v)$ labels in $O(\log n)$ time using n processors by assigning $\lceil |U(v)|/\log n \rceil$ processors to each node v [21]. We can then broadcast each $pref(p, v)$ label to the successor of v in $sibling(v)$, which takes $O(\log n)$ time using n processors by assigning $\lceil |U(v)|/\log n \rceil$ processors to each node v. Finally, we can compute all the $up(p, v)$ labels in $O(\log n)$ additional time by assigning a single processor to each point p and tracing the path in the tree from the leaf node that contains p up to the root. This is an EREW operation because computing all the $up(p, v)$ labels only depends upon accessing memory locations associated with the point p.

The EREW solution to the visibility from a point problem requires $O(n \log n)$ space for a different reason, namely, because we can solve it by constructing the plane-sweep tree for the segments (we need not have the $Cover(v)$'s in sorted order, however), computing the lowest segment in each $Cover(v)$, and then performing a top-down parallel min-finding computation to find the segment visible on each interval (p_i, p_{i+1}). Since these are all straightforward computations, given the discussion presented earlier in this paper, we leave the details to the reader.

8. Conclusion.

In this paper we gave several general techniques for solving problems efficiently using parallel divide-and-conquer. Our techniques are based on nontrivial generalizations of the merge-sorting approach of Cole [13]. It is interesting to note that Cole's algorithm improved the previous results by a constant factor, whereas our algorithms improve the previous results asymptotically.

Two of our techniques involved methods for performing fractional cascading and a generalized version of the merge-sorting problem optimally in parallel. Our method for doing fractional cascading runs in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors, and, if implemented as a sequential algorithm, results in a sequential alternative to the method of Chazelle and Guibas [12] for fractional cascading.

We also showed how to apply the generalized merging procedure and fractional cascading to efficiently solve several problems by "cascading" the divide-and-conquer paradigm. For three of the problems—trapezoidal decomposition, planar point location, and segment intersection detection—the method involved merging in the line segment partial order, and required considerable care to avoid situations in which the algorithm would halt because it attempted to compare two incomparable segments. All three of these algorithms ran in $O(\log n)$ time using n processors, which is optimal for all but the point location problem. In addition, since our algorithm for doing planar point location results in a query time of $O(\log n)$, our result immediately implies an $O(\log^2 n)$ time, n processor solution to the problem of constructing the Voronoi diagram of n planar points, using the algorithm of Aggarwal et al. [1].

We showed how to apply the cascading divide-and-conquer technique to problems that can be solved by merging with labeling functions. We used this approach to solve the three-dimensional maxima problem, the two-set dominance counting problem, the rectilinear segment intersection counting problem, and the visibility from a point problem. Our algorithms for these problems all ran in $O(\log n)$ time using n processors, which is optimal.

REFERENCES

[1] A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C. Ó'DÚNLAING, AND C. YAP, *Parallel computational geometry*, Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 468-477.

[2] ———, *Parallel computational geometry*, Algorithmica, 3 (1988), pp. 293-328.

[3] M. J. ATALLAH, R. COLE, AND M. T. GOODRICH, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 151-160.

[4] M. J. ATALLAH AND M. T. GOODRICH, *Efficient parallel solutions to some geometric problems*, J. Parallel and Distributed Computing, 3 (1986), pp. 492-507.

[5] ———, *Efficient plane sweeping in parallel*, Proc. 2nd ACM Symposium on Computational Geometry, 1986, pp. 216-225.

[6] ———, *Parallel algorithms for some functions of two convex polygons*, Algorithmica, 3 (1988), pp. 535-548.

[7] M. BEN-OR, *Lower bounds for algebraic computation trees*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 80-86.

[8] J. L. BENTLEY AND D. WOOD, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., C-29 (1980), pp. 571-576.

[9] G. BILARDI AND A. NICOLAU, *Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines*, TR 86-769, Department of Computer Science, Cornell University, August 1986.

[10] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130-145.

[11] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. ACM, 21 (1974), pp. 201-206.

[12] B. CHAZELLE AND L. J. GUIBAS, *Fractional cascading: I. A data structuring technique*, Algorithmica, 1 (1986), pp. 133-162.

[13] R. COLE, *Parallel merge sort*, Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 511-516; SIAM J. Comput., 17 (1988), pp. 770-785.

[14] N. DADOUN AND D. KIRKPATRICK, *Parallel processing for efficient subdivision search*, Proc. 3rd ACM Symposium on Computational Geometry, 1987, pp. 205-214.

[15] H. EDELSBRUNNER AND M. H. OVERMARS, *On the equivalence of some rectangle problems*, Inform. Process. Lett., 14 (1982), pp. 124-127.

[16] H. ELGINDY AND M. T. GOODRICH, *Parallel algorithms for shortest path problems in polygons*, The Visual Computer: Internat. J. Comput. Graphics, 3 (1988), pp. 371-378.

[17] M. T. GOODRICH, *Efficient parallel techniques for computational geometry*, Ph.D. thesis, Department of Computer Science, Purdue University, W. Lafayette, IN, 1987.

[18] ———, *Finding the convex hull of a sorted point set in parallel*, Inform. Process. Lett., 26 (1987), pp. 173-179.

[19] ———, *Triangulating a polygon in parallel*, J. Algorithms, to appear.

[20] H. T. KUNG, F. LUCCIO, AND F. P. PREPARATA, *On finding the maxima of a set of vectors*, J. ACM, 22 (1975), pp. 469-476.

[21] C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *The Power of Parallel Prefix*, Proc. 1985 IEEE Internat. Conference on Parallel Processing, St. Charles, IL, 1985, pp. 180-185.

[22] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. ACM (1980), pp. 831-838.

[23] J. H. REIF, *An optimal parallel algorithm for integer sorting*, Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 496-504.

[24] J. H. REIF AND S. SEN, *Optimal Randomized Parallel Algorithms for Computational Geometry*, Proc. 1987 IEEE Internat. Conference on Parallel Processing, 1987, pp. 270-277.

[25] Y. SHILOACH AND U. VISHKIN, *Finding the maximum merging, and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88-102.

[26] L. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348-355.

[27] H. WAGENER, *Optimally parallel algorithms for convex hull determination*, manuscript, 1985.

[28] C. K. YAP, *Parallel triangulation of a polygon in two calls to the trapezoidal map*, Algorithmica, 3 (1988), pp. 279-288.