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

by

Mikhail J. Atallah[*]
Richard Cole[†]
Michael T. Goodrich[‡]

Ultracomputer Note #127
Computer Science Technical Report #317
September, 1987

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012

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

by

Mikhail J. Atallah[*]
Richard Cole[†]
Michael T. Goodrich[‡]

Ultracomputer Note #127
Computer Science Technical Report #317
September, 1987

ABSTRACT

We present techniques for parallel divide-and-conquer, resulting in improved parallel algorithms for a number of problems. The problems for which we give improved algorithms include intersection detection, trapezoidal decomposition, and planar point location. We also give efficient parallel algorithms for fractional cascading, 3-dimensional maxima, 2-set dominance counting, and visibility from a point. All of our algorithms run in $O(\log n)$ time with either a linear or sub-linear number of processors in the CREW PRAM model.

---

[*] This work was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-84-51393, with matching funds from AT&T. Current Address: Department of Computer Science, Purdue University, West Lafayette, IN 47907

[†] This research was supported in part by the NSF Grants DCR-84-01633 and CCR-8702271, and by ONR grant N00014-85-K-0046. Current Address: Courant Institute, New York University, New York, NY 10012

[‡] This research was supported by the ONR under Grants N00014-84-K-0502 and N00014-86-K-0689, the NSF under Grant DCR-84-51383 (with matching funds from AT&T), and a David Ross Grant from the Purdue Research Foundation. Current Address: Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218

1 Introduction

This paper presents a number of general techniques for parallel divide-and-conquer. These techniques are based on a nontrivial generalizations of Cole's recent parallel merge sort result [11], and enable us to achieve improved complexity bounds for a large number of geometric problems.

The framework is one in which we want to design efficient parallel algorithms for the CREW PRAM computation model. Recall that this is the synchronous shared memory model in which processors may simultaneous read from any memory location but simultaneous writes are not allowed. 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 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 optimal algorithms. Previous optimal algorithms in this area have been for the convex hull problem [1,3,5,16,25] and 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. In this paper 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. In [11], the list at a node was the merge of the two lists of its children. In our problems, however, the lists at a node of the tree depend on the lists of the children in more complex ways, involving more than just simple merge operations. 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 in [11]. Such operations arise here because the lists at a node contain segments ordered by the "above" relationship (which is obviously not a total order). One may be tempted to delay performing these set difference operations until the end, as a postprocessing "purging" step. 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 is it possible to explicitly perform

the set difference operations without sacrificing the time-efficiency of the cascading method. Our solution avoids both of these problems.

Another significant contribution of this paper is an optimal parallel construction of the "fractional cascading" data structure of Chazelle and Guibas [10]. This too is based on a generalization of Cole's method [11] in the sense that instead of 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 (as 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 resulted 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.

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 any edge (v, w) in E and the position of an arbitrary element x in $C(v)$, enables a single processor to locate x in $C(w)$ in $(\log nd(G))$ time, where $d(G)$ is the maximum degree of any node in G. In [10] 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 in [1], later improved to $(\log n \log\log n, n)$ in [4]. 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 which, 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 in [1], later improved to $(\log n \log\log n, n)$, $Q(n) = O(\log n)$ in [4]. In [12] this is further improved 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 [4]. We improve this to $(\log n, n)$.

3-Dimensional Maxima: Given a set S of n points in 3-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 [4]. We improve this to $(\log n, n)$.

Two-Set Dominance Counting: Given a set $V = \{p_1, p_2, \ldots, p_l\}$ and a set $U = \{q_1, q_2, \ldots, q_m\}$ of points in the plane, determine for each point q_i in U the number of points in V whose x and y coordinates are both less than the corresponding coordinates of q_i. The problem size is $n = l + m$. A $(\log n \log\log n, n)$ solution was given in [4]. 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 [4]. We improve this to $(\log n, n)$.

We recently learned that Reif and Sen [22] can solve planar point location, trapezoidal decomposition, segment intersection, and visibility in randomized $O(\log n)$ time using $O(n)$ processors. All of our algorithms are deterministic.

This paper is organized as follows. In Section 2 we present a generalized version of the cascading merge procedure and in Section 3 we give our method for doing fractional cascading in parallel. In Section 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 Section 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 Section 6 we use the cascading divide-and-conquer technique to compute labeling functions, and show how to use this approach to solve 3-dimensional maxima, 2-set dominance counting, and visibility from a point.

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 $U(v)$ which consists of all the items stored in descendant nodes of v. (See Figure ??.) In this section we show how to construct $U(v)$ 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 [11], 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 only has one child then we

can add a child to v (a leaf node) which 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.

Let a, b, and c be three items, with $a \leq b$. We say c is *between* a and b if $a < c \leq b$. Let two sorted (nondecreasing) lists $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ be given. Given an element a we define the *predecessor* of a in B to be the greatest element in B which 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 0). 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 the elements in A or B. If B is a subset of A, then we define $A - B$ to be the sorted list of the elements in A which 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, sibling of v, left child of v, right child of v, and $depth(v)$ (the root is at depth 0), 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. As in [11], 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 c-th element of L, and call this set the *c-sample* of L. That is, $SAMP_c(L)$ consists of the c-th 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. 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 $\lfloor s/3 \rfloor \leq alt(v) \leq s$, and we say v is *full at stage* s if $alt(v) = \lfloor s/3 \rfloor$. It will become apparent below that 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:

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

At stage $s+1$ we perform the following computation at each internal node v which 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 $U_{s+1}(v) := U'_{s+1}(lchild(v)) \cup U'_{s+1}(rchild(v))$.

Thus, until v becomes full $U'_{s+1}(v)$ will be the list consisting of every fourth element of $U_s(v)$. We continue defining $U'_{s+1}(v)$ in this way up to the point that v becomes full. If s_v is the stage at which v becomes full, then at stage $s_v + 1$, $U'_{s+1}(v)$ is the two-sample of $U_s(v)$ $(= U(v))$, and, at stage $s_v + 2$, $U'_{s+1}(v) = U_s(v)$ $(= U(v))$. Therefore, at stage $s_v + 3$, $parent(v)$ is full. After $3 * height(T)$ stages every node is 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 which is essential to the construction.

Lemma 2.1: *For any stage $s \geq 0$ and any node $v \in T$, $|U_{s+1}(v)| \leq 2|U_s(v)| + 4$.*

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 when $alt(v) > \lfloor s/3 \rfloor + 1$. We know that $U_{s+1}(v) = U'_{s+1}(x) \cup U'_{s+1}(y)$, where $x = lchild(v)$ and $y = rchild(v)$. Thus, we have the following:

$$|U_{s+1}(v)| = \left\lfloor \frac{|U_s(x)|}{4} \right\rfloor + \left\lfloor \frac{|U_s(y)|}{4} \right\rfloor \qquad \text{(from definitions)}$$

$$\leq \left\lfloor \frac{2|U_{s-1}(x)| + 4}{4} \right\rfloor + \left\lfloor \frac{2|U_{s-1}(y)| + 4}{4} \right\rfloor \qquad \text{(by induction hyp.)}$$

$$\leq 2\left(\left\lfloor \frac{|U_{s-1}(x)|}{4} \right\rfloor + \left\lfloor \frac{|U_{s-1}(y)|}{4} \right\rfloor \right) + 4$$

$$= 2|U_s(v)| + 4$$

The case when $alt(v) = \lfloor s/3 \rfloor + 1$ is similar (actually, it is simpler). ∎

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 $k \geq 1$ and $s \geq 1$.*

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 3 items, hence, $U_s(v)$ contains at most 10 elements. Also, if $U'_s(v)$ contains one item, then $U_{s-1}(v)$ contains at most 7 items, hence, $U_s(v)$ contains at most 18 items (at most 15 of which can be between any two adjacent items in $(-\infty, U'_s(v), \infty)$).

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 $(-\infty, U'_{s+1}(v), \infty)$. We first suppose that $alt(v) > \lfloor s/3 \rfloor$. 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 $[a_1, b_1]$ (resp. $[a_2, b_2]$) be the smallest interval containing $[a,b]$ such that $a_1, b_1 \in (-\infty, U'_s(lchild(v)), \infty)$ (resp. $a_2, b_2 \in (-\infty, U'_s(rchild(v)), \infty)$). 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 $(-\infty, U'_s(rchild(v)), \infty)$. Clearly, $h + j = g$. 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 $g \leq 4k+1$. 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 $(2h+2)+(2j+2) \leq (2h+2)+(2(4k-h+1)+2) \leq 8k+8$. The proof for the case when $alt(v) = \lfloor s/3 \rfloor$ is similar (actually it is simpler). ■

Corollary 2.3: *The list $(-\infty, U'_s(v), \infty)$ is a 4-cover for $U'_{s+1}(v)$, for all $s \geq 0$.*

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

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

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

Lemma 2.4: (The Merge Lemma) [11]. *Suppose we are given sorted lists A_s, A'_{s+1}, B'_s, B'_{s+1}, C'_s, and C'_{s+1}, where we have the following (input conditions):*

(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 $C'_{s+1} = U'_{s+1}(y)$. Note that assigning the sets of Lemma 2.4 in this way satisfies input conditions 1–4 from definitions. The ranking information we maintain from stage to stage satisfies input conditions 5–7. Thus, in each stage s we can construct the set $U_{s+1}(v)$ 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 4. Note that in stage s it is only necessary to store the sets for $s-1$; we can discard any sets for stages previous to that.

The method for performing all these merges with a total of $|T|$ processors is basically to start out with 1 processor 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 k processors to perform the merge. When v's parent becomes full, then we no longer "store" any processors at v. (See [15] for details.) Thus, since there are at most $O(|T|)$ elements present in active nodes of T for any stage s, we can perform the entire generalized cascading procedure using $|T|$ processors.

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. Construct a tree T' from T by replacing each leaf v of T with a complete binary tree with $|A(v)|$ leaves, and associate each item in $A(v)$ with one of these leaves. T' now satisfies the conditions of the method outlined above. We summarize the above discussion in the following theorem:

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. We can compute, for each node* $v \in T$, *the list* $U(v)$, *which is the*

union of all items stored at descendants of v, sorted in an array, in $O(height(T)+\log(\max_v |A(v)|))$ time using a total of $|T|+N$ processors in the CREW PRAM computational model, where N is the total number of items stored in T.

Proof: The complexity bounds from the fact that T' has 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 problem is to construct an $O(n)$ space data structure that, given any edge $(v,w)\in E$ and the position of an arbitrary element x in $C(v)$, enables a single processor to locate x in $C(w)$ in $O(\log d(G))$ time, where $d(G)$ is the maximum degree of any node in G and $n=|V|+|E|+\sum_{v\in V}|C(v)|$. Chazelle and Guibas [10] give an elegant $O(n)$ time sequential construction of such a data structure. However, their approach does not appear to be "parallelizable." Here we construct such a data structure in $O(\log n)$ time using $O(n/\log n)$ processors.

We define $In(v,G)$ (resp. $Out(v,G)$) to be the set of all nodes w in V such that $(w,v)\in E$ (resp. $(v,w)\in E$). 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 $d(G)=\max_{v\in V}\{d(v)\}$. A sequence $(v_1,v_2,\ldots,v_m)$ of vertices is a *walk* if $(v_i,v_{i+1})\in E$ for all $i\in\{1,2,\ldots,m-1\}$.

We begin by showing how to do fractional cascading 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 [9].

We begin the fractional cascading construction by preprocessing the directed graph G, converting it into a directed graph $\hat{G}=(\hat{V},\hat{E})$ such that $d(\hat{G})\leq 3$ 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} (see Figure ??). Each leaf in T_v^{in} (resp., T_v^{out}) corresponds to an edge coming into v (resp., going out of v). So there are $|In(v,G)|$ leaves in T_v^{in} and $|Out(v,G)|$ leaves in T_v^{out}. 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 Figure ??.) 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 $u < w$. 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 $O(\log n)$ time using n processors [11]. We also store in each internal node z of T_v^{out} the leaf node u which has the smallest name of all the descendents of z.

The above preprocessing step is similar to a preprocessing step used in the sequential fractional cascading algorithm of Chazelle and Guibas [10]. This is where the resemblance to the sequential algorithm ends, however.

The goal for the rest of the computation is to construct special sorted lists $B(v)$, which we call *bridge lists*, for every node $v \in \hat{V}$. We shall define these bridge lists so that $B(v) = C(v)$ if v is in V, and, 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 $B(v)$, then 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, $B_0(v) = \emptyset$ 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 bigger and bigger samples of the catalogue $C(v)$ with each stage, the sample at stage $s + 1$ being twice the size of the sample at stage s. These samples are then cascaded back into the gateway v' for v and from there back through the fan-in tree for v. Of course, we will 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 are as follows. Recall that $B_0(v) = \emptyset$ for all $v \in \hat{V}$. For stage $s \geq 0$ we define $B'_{s+1}(v)$ as follows:

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

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

where $c(s) = 2^{\lceil \log N \rceil - s}$ and N is the size of the largest catalogue. The **Per-Stage Computation**, then, is to compute $B_{s+1}(v)$ 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 we go from one stage to the next $B'_{s+1}(v)$ will be empty for a while, then it will consist of a single element of $C(v)$, then at most three elements, then at most five elements, then at most nine elements, and so on. This continues until the final stage (stage $\lceil \log N \rceil$), when $B'_{s+1}(v) = C(v)$. We show below that each stage can be performed in $O(1)$ time, resulting in a total running time of $O(\log n)$ (it takes $O(\log n)$ time to compute the value of N). The following 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 3.6: *For any stage $s \geq 0$ and any node $v \in T$, $|B_{s+1}(v)| \leq 2|B_s(v)| + 4$.*

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 is obviously true, from the definition of $c(s)$. If $v \in \hat{V} - V$ and $Out(v, \hat{G})$ contains only one node, then the claim is clearly true as well. So consider the case when $v \in \hat{V} - V$, and $Out(v, \hat{G}) = \{w_1, w_2\}$. We know that $B_{s+1}(v) = B'_{s+1}(w_1) \cup B'_{s+1}(w_2)$. Thus, we have the following:

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

■

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

Lemma 3.7: *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 $8k+6$ items in $B_s(v)$ for all $k \geq 1$ and $s \geq 1$.*

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 3.8: *The list $(-\infty, B'_s(v), \infty)$ is a 4-cover for $B'_{s+1}(v)$, for $s \geq 0$.*

Corollary 3.9: *The list $(-\infty, B_s(v), \infty)$ is an 14-cover for $B_s(w)$, for $s \geq 0$ 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 completes 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 non-empty, 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 sets for $s-1$; we can discard any sets 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 sets 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).

By Corollary 3.9, $B(v)$ is an 14-cover for $B(w)$ if $(w,v) \in \hat{E}$. This implies that $\hat{G}$ is a fractional cascading data structure. For, given the position of an item x in $B(w)$ we can locate x in $B(v)$ in $O(1)$ time, for each $(w,v) \in \hat{E}$. Thus, given the position of an item x in $C(w)$ we can locate x in $C(v)$, where $(w,v) \in E$, by the following the path in $\hat{G}$ from w', which stores $B(w')(= C(w))$, to v, which stores $C(v)$, locating x in each bridge list along the way in $O(1)$ time per vertex. Since this path contains at most $\lceil \log d(G) \rceil$ vertices, this takes $O(\log d(G))$ time. We show that $\hat{G}$ uses $O(n)$ space in the following lemma.

Lemma 3.10: *Let n_s denote the space used for the data structure in the fractional cascading algorithm in stage s, that is, $n_s = \sum_{v \in \hat{V}} |B_s(v)|$. Then n_s is $O(n)$ for any $s \in [0, \lceil \log N \rceil]$.*

Proof: Let us analyze the amount of space, n_v, that is added to $\hat{G}$ for any particular catalogue $C(v)$. Since we started with all the bridge lists being empty, $n_s = \sum_{v \in \hat{V}} n_v$. Recall that while constructing $\hat{G}$ we copy one fourth of the elements in each bridge list to at most two of its neighbors. Thus, we have the following for any $s \in [0, \lceil \log N \rceil]$:

$$\begin{aligned} n_v &\leq |C(v)| + 2\lfloor |C(v)|/4 \rfloor + 2^2 \lfloor |C(v)|/4^2 \rfloor + 2^3 \lfloor |C(v)|/4^3 \rfloor + \ldots \\ &\leq 2|C(v)|. \end{aligned}$$

(This is obviously an over-estimate; but it is good enough for the purposes of the analysis.) Therefore, $n_s \leq 2n$. Note that the upper bound for n_v given above holds even if $\hat{G}$ contains cycles. ∎

We have just shown 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. We have not, however, shown how to assign these n processors to their respective jobs. Initially, we assign $2|C(v)|$ processors to each node $v \in V$ and no processors to each node $v \in \hat{V} - V$. Then, 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 processors to go with them. Since $n_v \leq 2|C(v)|$, we know that there are enough 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). There are clearly enough processors to perform the merge to construct $B_s(v)$ and repeat the give-away procedure for the next stage. Thus, we have the following:

Lemma 3.11: *Given any catalogue graph G we can construct a fractional cascading data structure for G which uses $O(n)$ space in $O(\log n)$ time using n processors in the CREW PRAM model.* ∎

Thus, we can do fractional cascading in $O(\log n)$ time using n processors. For the applications we study in this paper we can do even better, however.

Lemma 3.12: *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) clearly requires only $O(n)$ operations. We will charge all other operations to nodes $v \in V$, and compute the total number of operations that are performed because the node v initially stores $|C(v)|$ elements. In the first stage, s_v, that $B_s(v')$ becomes non-empty it will receive one element of $C(v)$ from v, hence we will perform 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 8 elements from v (as early as stage $s_v + 3$), then we will perform one more operation, passing one element to the fan-in tree for v. In the next stage, $s_v + 4$, we will perform at most 2 additional operations, then at most 4 additional operations in stage $s_v + 5$, 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:

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

(This is actually an over-estimate, since not all nodes in $\hat{G}$ have out-degree two.) 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 $2(|C(v)| + 2\lfloor |C(v)|/4 \rfloor + 2^2\lfloor |C(v)|/4^2 \rfloor + \ldots + 2^{k_v})$, which is at most $4|C(v)|$. Thus, the total number of operations performed by the fractional cascading algorithm is $O(n)$. ∎

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 3.13: *[9] 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 $O(\lfloor N/P \rfloor + T)$:

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

There are two qualifications one must make to Brent's theorem before one can apply it in the PRAM model, however. The first is one 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 exactly 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 $\Gamma = \{p_1, p_2, \ldots, p_{2n}\}$ be the set of processors used in the fractional cascading algorithm, and let $\Gamma' = \{p'_1, p'_2, \ldots, p'_{\lceil n/\log n \rceil}\}$ 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, it is easy to compute the graph $\hat{G}$ and the initial assignment of processors from Γ, so that we assign $2|C(v)|$ 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 $(a_1, a_2, ..., a_n)$ can be done in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors [19,20].) 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 assigning 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, because this depends only on the value of $v(p_i)$ and the size of $C(v)$ 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 due to Reif [21] (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 3.14: *Given a catalogue graph $G = (V, E)$, such that $d(G)$ is $O(1)$ or we are 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 $n = |V| + |E| + \sum_{v \in V} |C(v)|$, which is optimal. ∎

Given a walk $(v_1, ..., v_m)$ and an arbitrary element x, the query which asks for locating x in every $C(v_i)$ is called the *multilocation* of x in $(v_1, ..., v_m)$. Thus, to perform a multilocation of x in a walk $(v_1, ..., v_m)$ we first locate x in $C(v_1)$ by binary search and then traverse each other vertex in the walk in $O(\log d(G))$ time per vertex (by Theorem 3.14). Therefore, we can perform the multilocation of x in $(v_1, ..., v_m)$ in $O(\log |C(v_1)| + m \log d(G))$ time. (See Figure ??.) In the section which follows we give some applications of fractional cascading in parallel.

4 The Plane-Sweep Tree Data Structure

In this section we define a data structure, and show how to turn it into a fractional cascading data structure so that it can be used to solve the trapezoidal decomposition problem and the planar-point location problems 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 Section 5.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of non-intersecting line segments in the plane, and let $X(S) = (\alpha_1, \alpha_2, \ldots, \alpha_{2n})$ be the (non-decreasing) 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., $x_i < x_{i+1}$. Let $X' = (x_1, x_2, \ldots, x_m)$ 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 $(-\infty, x_1]$, $[x_1, x_2]$, $[x_2, x_3]$, ..., $[x_{m-1}, x_m]$, $[x_m, +\infty)$ (respectively). Associated with each internal node $v \in T$ is a closed interval $I_v = [x_i, x_j]$ 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 $\Pi_{parent(v)}$. Clearly, no segment covers more than 2 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 which 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 [7]. 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). They store the set $Cover(v)$ at each node v sorted by the "above" relation for line segments. They construct these sets by first collecting the segments in each $Cover(v)$ and then sorting all the $Cover(v)$'s in parallel, an operation which requires $\Theta(\log^2 n)$ time using n processors,

since there are a total of $\Theta(n \log n)$ items to sort [11]. Once these sets 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 $Cover(v)$ which 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 is exactly 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 $Cover(v)$ sets such that $p \in \Pi_v$ (there are $O(\log n)$ such lists for each p).

As mentioned earlier, our plane-sweep tree data structure shares some of the structure of the data structure of Aggarwal *et al.*, but differs from it in some important ways. One important difference is that it allows us to perform $O(n)$ multilocations in $O(\log n)$ time using n processors, after a preprocessing step which takes $O(\log n)$ time using n processors. Also, instead of taking X' to be the entire set $X(S)$, 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:

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

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 3.14, 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 $\hat{T}$. Although $End(v)$ is defined for each node of T we only construct a copy of $End(v)$ for leaf nodes v. $End(v)$ is not sorted in any particular order. 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 Figure ??).

In section 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 $S = \{s_1, s_2, \ldots, s_n\}$ be a set of non-intersecting line segments in the plane. For any endpoint p of a segment in S a *trapezoidal segment* for p is a segment of S which 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 used as a building block to solve other problems, such as polygon triangulation [1,17,26] or shortest paths in a polygon [14].

Theorem 4.15: *A trapezoidal decomposition of a set S of n 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: We first construct the plane-sweep tree data structure T for S. Theorem 5.18 (to be given later, in Section 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 $(v, ..., root(T))$ be the leaf-to-root path in $\hat{T}$ which 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 which are below p, and take the one which is closest to p. 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 $\lceil \log n \rceil$ answers the segment which 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, which is directly below p. Since the length of the walk from v to $root(T)$ is at most $\lceil \log n \rceil$, by Theorem 3.14 this computation done in $O(\log n)$ time using n processors. Since the 2-dimensional maxima problem can be reduced to trapezoidal decomposition in $O(1)$ time using n processors [15], and the 2-dimensional maxima problem has a sequential lower bound of $\Omega(n \log n)$ in the algebraic computation tree model [6,18], 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 [1,2,4,15,26]. In fact, the method of Theorem 4.15 can be used in the algorithms of Goodrich [17] and Yap [26] 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 which, 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 [1].

Theorem 4.16: *Given a planar subdivision S consisting of n edges, we can construct a data structure which can be used to determine for any query point p the face in S containing p in $O(\log n)$ time. The 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 5.18 (to be given later, in Section 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 4.15 to find the segment in S directly below p. After we have determined the segment s_i in S which 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 4.16 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 Subsections 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 Section 2 to solve some geometric problems in which the elements being merged belong to the partial order defined by a set of non-intersecting line segments. Recall that in this partial order a segment s_1 is "above" a segment s_2 if there is a vertical line which 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 briefly give an overview of the problems encountered and our solutions to them. The essential computation is as follows: we have a binary tree with sets stored in its leaves, and we wish to combine them in pairs (up the tree) to construct sets at internal nodes. The main difficulty is that the set stored at some node v is not defined as a simple merge of the sets stored at the children of v. Instead, its definition involves deleting elements from sets 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 non-deleted elements which are not comparable to others at the same node; hence, there will be instances when processors try to compare two elements which 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 which involves simultaneously performing a number of cascading merges in parallel. We complete the computation by performing a purging post-processing step to remove the segments that "changed identity" (as an alternative to being deleted).

For the intersection detection problem, we need to dove-tail 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 post-processing to then check if any intersections are present.

The next two subsections give the details.

5.1 Plane-Sweep Tree Construction

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 $v \in T$:

$$\begin{aligned} L(v) &= \{s_i \mid s_i \in End(v) \text{ and } s_i \cap \text{left}(\Pi_v) \neq \emptyset\}, \\ R(v) &= \{s_i \mid s_i \in End(v) \text{ and } s_i \cap \text{right}(\Pi_v) \neq \emptyset\}, \\ l(v,d) &= \{s_i \mid s_i \in L(v) \text{ and } d = depth(dom(s_i))\}, \\ r(v,d) &= \{s_i \mid s_i \in R(v) \text{ and } d = depth(dom(s_i))\}. \end{aligned}$$

Note that $l(v,d)$ and $r(v,d)$ are only defined for $0 \leq d < depth(v)$. 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 5.17: *Let v be a node in T with left child x and right child y. Then we have the following (see Figure ??):*

(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) $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$;

(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: Easily follows from definitions. ∎

Lemma 5.17 essentially states that the sets l, r, L, R, and $Cover$ for the nodes on a particular level of T can be defined in terms of sets for nodes on the next lower level of T. We could use this lemma and the parallel merge technique of Valiant [24], as implemented by Borodin and Hopcroft [8], to construct a sorted copy of each $Cover(v)$ set 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 [1]. We can do even better, however, by exploiting the structure of the L and R sets. 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 five 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 Equations 1 and 2 of Lemma 5.17 (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 $l(v,d_v)$ (resp. $r(v,d_v)$) its predecessor segment in $L(v) - l(v,d_v)$ (resp. $R(v) - r(v,d_v)$). 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 $l(v,d_v)$ (resp. $r(v,d_v)$) with all the $l(v,d)$ (resp. $r(v,d)$) 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 Equations 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 Equations 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 sets 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 Equations 7 and 8 and the sets constructed in Step 3. The implementation of this step amounts to compressing each $L(v)$ (resp. $R(v)$) so as to delete all the segments in $l(v,d_v)$ (resp. $r(v,d_v)$), and then copying the set 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 $r(v,d)$ sets 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 sets $l(v,d)$ and $r(v,d)$, as given by Equations 1 and 2 of Lemma 5.17. This implies that if we store $l(v,d)$ (resp. $r(v,d)$) in every leaf node v of $T(d)$, then for any node $v \in T(d)$, $l(v,d)$ is precisely the sorted union of the sets stored in the descendants of v. We start with the elements belonging to $l(v,d)$ (resp. $r(v,d)$) stored (unsorted) in a set $A(v)$ 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 $A(v)$'s 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 $r(v,d)$ sets in $O(\log n)$ time using n processors.

5.3 Step 2: Computing predecessors

In Step 2 we wish to compute for each segment in the set $l(v,d_v)$ (resp. $r(v,d_v)$) its predecessor segment in $L(v) - l(v,d_v)$ (resp. $R(v) - r(v,d_v)$), 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 Equations 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 [23] we merge a copy of $l(v,d_v)$ with each of $l(v,0),\ldots,l(v,d_v-1)$. This takes $O(\log n)$ time using $\lceil |L(v)|/\log n \rceil$ 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 clearly implement all these merges in parallel using n processors. Once we have completed all the merges it is an easy matter to assign a single processor to each segment s_i and compare the predecessors of s_i in $l(v,0),\ldots,l(v,d_v-1)$ 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, 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 Equations 5 and 6 of Lemma 5.17. Initially, we have $L(v)$ and $R(v)$ constructed only for the leaves. We then merge these sets up the tree based on the Equations 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 difference operations as well as list merges as we are cascading up the tree. Unfortunately, performing these difference operations explicitly at each level would require $\Theta(\log^2 n)$ time. We get around this problem by never actually performing the difference operations. That is, we don't actually delete segments from any lists. Instead, we change the identity of a segment s_i in say, $l(y, d_y)$, to its predecessor in $L(y) - l(y, d_y)$ when we are performing the merge at node v. (See Figure ??.) We do this instead of deleting it from $L(y)$, because segments in $l(y, d_y)$ are not comparable to segments in $L(x)$ (the list we wish to merge $L(y) - l(y, d_y)$ with).

Clearly, the fact that we change the identity of a segment in $l(y, d_y)$ to its predecessor in $L(y) - l(y, d_y)$ 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 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 it is an easy matter to 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 Equations 7 and 8 of Lemma 5.17. We implement this step by first compressing each $L(v)$ (resp. $R(v)$) so as to delete all the segments in $l(v, d_v)$ (resp. $r(v, d_v)$), and then by copying the set of segments so computed to the sibling of v in T. This can clearly be done in $O(\log n)$ time using n processors.

Thus, summarizing the entire previous section, we have the following:

Theorem 5.18: *Given a set S of non-intersecting 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 T requires $\Omega(n \log n)$ space. ■

In the previous sections we assumed that segments did not intersect. Indeed, T is defined only if they don't intersect. We show in the next section that the following idea is useful when we want to check whether any two of the given segments intersect: we can, in effect, pretend we are constructing T and check for intersections while doing so.

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 which we use to test for an intersection.

Lemma 5.19: *[1] The segments in S are non-intersecting 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 in our data structure the multilocations 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 $Cover(v)$ sets 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 on 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 $r(v,d)$ sets 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 5.20: *Let $v \in T$ and $0 \leq d < depth(v)$ be given, and let s_1 and s_2 be two segments such that $s_1 \in l(w,d)$ and $s_2 \in l(z,d)$ (resp. $s_1 \in r(w,d)$ 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 $0 \leq d \leq \lceil \log n \rceil$ be given. Recall that $l(v,d)$ (resp., $r(v,d)$) is defined to be the set of all segments in $L(v)$ ($R(v)$) which 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 which is an ancestor of v and is at depth d in T. ∎

Thus, we can perform the merges based on Equations 1 and 2 of Lemma 5.17 (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 intersect 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 $l(v,d_v)$ (resp. $r(v,d_v)$) its predecessor segment in $L(v) - l(v,d_v)$ (resp. $R(v) - r(v,d_v)$), where $d_v = depth(parent(v))$. Recall that we did this by merging $l(v,d_v)$ with each of $l(v,0),\ldots,l(v,d_v-1)$. 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 $l(v,0),\ldots,l(v,d_v)$ belong to $L(v)$; hence, intersect left(Π_v). After Step 1 finishes each set $l(v,d)$ 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 $l(v,0),\ldots,l(v,d_v)$ 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 Equations 5 and 6 of Lemma 5.17 (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 we will have two elements which are out of order. We replace both items by the distinguished symbol \$. Then, as the merges continue up the tree, any time we compare an item with \$ we replace that item with \$ 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 \$ 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 sets 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 intersections 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 in the construction 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 which 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 which 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 $Cover(v)$ set 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 5.21: *Given a set of n line segments in the plane it is possible to 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 Section 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 alternative to the known sequential algorithms (which use the plane-sweeping paradigm), as well as giving rise to efficient parallel algorithms. We begin with the 3-dimensional maxima problem.

6.1 The 3-Dimensional Maxima Problem

Let $V = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\Re^3$. For simplicity, we assume that no two input points have the same x (resp., 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 *1-dominates* another point p_j if $x(p_i) > x(p_j)$, *2-dominates* p_j if $x(p_i) > x(p_j)$ and $y(p_i) > y(p_j)$, and *3-dominates* p_j 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 3-dominated by any other point in V. The 3-dimensional maxima problem, then, is to compute the set, M, of maxima in V. We show how to solve the 3-dimensional maxima problem efficiently in parallel in the following algorithm. The labels we use are motivated by the optimal sequential plane-sweeping algorithm of Kung, Luccio, and Preparata [18].

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 [11]. Let T be a complete binary tree with leaf nodes $v_1, v_2, \ldots, v_n$ (from left to right). In each leaf node v_i we store the set $B(v_i) = (-\infty, p_i)$, 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 clearly 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 the previous section, 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 $U(v)$ which 1-dominate p_i, and $ztd(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ which 2-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 input points are all larger than $x(p_i)$). (See Figure ??.) As we are performing the cascading-merge, we update the labels *zod* and *ztd* based on the equations in the following lemma:

Lemma 6.22: *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following:*

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

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

Proof: Consider Equation 9. If $p_i \in U(u)$, then it is clear that every point which 1-dominates p_i's predecessor in $U(w)$ also 1-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 Equation 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 $U(w)$ which 1-dominates p_i's predecessor in $U(w)$ must 2-dominate p_i. Thus, $ztd(p_i, v)$ is the 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 2-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 6.22 it should be clear that at the stage 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 $U(root(T))$, 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 $ztd(p_i, root(T)) \leq z(p_i)$ (there is no point which 2-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 6.23: *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 using n processors in the CREW PRAM model, and this is optimal.*

Proof: We have established the correctness and complexity bounds for parallel 3-dimensional maxima finding in the discussion above. Kung, Luccio, and Preparata [18] 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 $k \geq 3$ (since the 2-dimensional maxima problem can easily be solved in $O(\log n)$ time and $O(n)$ space by sorting followed by a parallel prefix computation).

Theorem 6.24: *For $k \geq 3$ the k-dimensional maxima problem can be solved in $O((\log n)^{k-2})$ time*

and $O(n)$ space using n processors on a CREW PRAM.

Proof: The method is a straightforward parallelization of the algorithm by Kung, Luccio, and Preparata [18], 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 $A = \{q_1, q_2, \ldots, q_m\}$ and a set $B = \{p_1, p_2, \ldots, p_l\}$ of points in the plane, and wish to know for each point p_i in B the number of points in A which are 2-dominated by p_i. For simplicity, we assume that the points have distinct x (resp. y) coordinates. In the algorithm which follows we show how to solve this problem efficiently in parallel.

Let $Y = \{p_1, p_2, \ldots, p_{l+m}\}$ be the union of A and B with the points listed by increasing y-coordinate, $y(p_i) < y(p_{i+1})$. Clearly, we can construct Y in $O(\log n)$ time using n processors [11], 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 $v_1, v_2, \ldots, v_n$ ordered from left to right, and in each leaf node v_i we store the set $U(v_i) = (-\infty, p_i)$, ($-\infty$ is 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 $U(v)$ which are in A and are 1-dominated by p_i, and the label $ntd(p_i, v)$ is the number of points in $U(v)$ which are in A and are 2-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 $\text{pred}(p_i, v)$ 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 Figure ??):

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. Let T

be a complete binary tree with leaf nodes $v_1, v_2, \ldots, v_n$ ordered from left to right. We associate the segment s_i with the leaf v_i and at v_i store the set $U(v_i) = (-\infty, p_1, p_2)$, 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 Theorem 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 whose endpoints are in $U(v)$ which is visible in the interval $(x(p_i), x(\text{succ}(p_i, v)))$, where $\text{succ}(p_i, v)$ denotes the successor of p_i in $U(v)$ (based on x-coordinates). Initially, the vis labels are initially 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 $\text{pred}(p_i, v)$ 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 Figure ??):

Lemma 6.25: *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" 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 $U(u)$, then for any point $p_i \in U(u)$ the segment visible (from $-\infty$) on the interval $(x(p_i), x(\text{succ}(p_i, v)))$ is the minimum of the segment visible on the interval $(x(p_i), x(\text{succ}(p_i, u)))$ and the segment which is visible on the interval $(x(\text{pred}(p_i, w)), x(\text{succ}(\text{pred}(p_i, w), w)))$. 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 $(x(\text{pred}(p_i, w)), x(\text{succ}(\text{pred}(p_i,$ 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 the 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 6.25 it should be clear that after merging the lists $U(lchild(v))$ and $U(rchild(v))$ 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

$(vis(p_1, root(T)), vis(p_2, root(T)), \ldots, vis(p_{2n}, root(T)))$ using a parallel prefix computation to construct a compact representation of the visible portion of the plane. We summarize in the following theorem:

Theorem 6.26: *Given a set S of n non-intersecting segments in the plane, we can find the lower envelope of S in $O(\log n)$ time 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, thus can do no better than $O(\log n)$ time using n processors. ∎

7 Conclusion

In this paper we gave several general techniques for solving various geometric problems efficiently in parallel. In particular, we gave optimal algorithms for doing fractional cascading and a generalized version of the merge sorting problem. Our method for doing fractional cascading ran in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors, and if implemented as a sequential algorithm results in an sequential alternative to the method of Chazelle and Guibas [10] 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 in the CREW PRAM model, 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 a $O(\log^2 n)$ 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 also applied the cascading divide-and-conquer technique to problems which can be solved by merging with labeling functions. We used this approach to optimally solve the 3-dimensional maxima problem, the 2-set dominance counting problem, the rectilinear segment intersection counting problem, and the visibility from a point problem. All of our algorithms for these problems ran in $O(\log n)$ time using n processors in the CREW PRAM model, which is optimal for all of the problems. Incidentally, the same technique as employed by Cole in [11] to implement his merging procedure in the EREW PRAM model (no simultaneous reads) can be applied to the algorithms

for generalized merging, fractional cascading, 3-dimensional maxima, 2-set dominance counting, rectilinear segment intersection counting, and visibility from a point, resulting in EREW PRAM algorithms with the same asymptotic bounds as the ones presented in this paper. Apparently, one cannot apply his techniques to our algorithms for the other problems, however, since the algorithms for those problems explicitly use concurrent reads.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Proc. 25th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 468–477.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," manuscript, 1987.

[3] M.J. Atallah and M.T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *Journal of Parallel and Distributed Computing*, Vol. 3, No. 4, December 1986, pp. 492–507.

[4] M.J. Atallah and M.T. Goodrich, "Efficient Plane Sweeping in Parallel," *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, pp. 216–225.

[5] M.J. Atallah and M.T. Goodrich, "Parallel Algorithms for Some Functions of Two Convex Polygons," to appear in *Algorithmica*; a preliminary version of this work appeared in *Proc. 24th Allerton Conference on Communication, Control, and Computing*, pp. 758–767, 1986.

[6] M. Ben-Or, "Lower Bounds for Algebraic Computation Trees," *Proc. 15th ACM Symp. on Theory of Computing*, 1983, pp. 80–86.

[7] J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, Vol. C-29, No. 7, July 1980, 571–576.

[8] A. Borodin and J.E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *Journal of Computer and System Sciences*, Vol. 30, No. 1, February 1985, pp. 130–145.

[9] R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM*, Vol. 21, No. 2, 1974, pp. 201–206.

[10] B. Chazelle and L.J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica*, Vol. 1, No. 2, pp. 133–162.

[11] R. Cole, "Parallel Merge Sort," to appear *SIAM Journal on Computing*; a preliminary version of this work appeared in *Proc. 27th IEEE Symp. on Foundations of Computer Science*, 1986, pp. 511–516.

[12] N. Dadoun and D. Kirkpatrick, "Parallel Processing for Efficient Subdivision Search," *3rd ACM Symp. Computational Geom.*, 1987, 205–214.

[13] H. Edelsbrunner and M.H. Overmars, "On the Equivalence of Some Rectangle Problems," *Information Processing Letters*, Vol. 14, No. 3, 1982, pp. 124–127.

[14] H. ElGindy and M.T. Goodrich, "Parallel Algorithms for Shortest Path Problems in Polygons," Technical Report MS-CIS-87-20, Department of Computer and Information Science, University of Pennsylvania, 1987.

[15] M.T. Goodrich, "Efficient Parallel Techniques for Computational Geometry," Ph.D. thesis, Department of Computer Science, Purdue University, 1987.

[16] M.T. Goodrich, "Finding the Convex Hull of a Sorted Point Set in Parallel," to appear in *Information Processing Letters*; also available as Purdue Computer Science Dept. Tech. Report CSD-TR-655, December 1986.

[17] M.T. Goodrich, "Triangulating a Polygon in Parallel," Purdue University Computer Science Tech. Report CSD-TR-679, May 1987.

[18] H.T. Kung, F. Luccio, F.P. Preparata, "On Finding the Maxima of a Set of Vectors," *J. ACM*, Vol. 22, No. 4, 1975, pp. 469–476.

[19] C.P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *Proc. 1985 IEEE Int. Conf. on Parallel Processing*, St. Charles, IL., pp. 180–185.

[20] R.E. Ladner and M.J. Fischer, "Parallel Prefix Computation," *J. ACM*, October 1980, pp. 831–838.

[21] J.H. Reif, "An Optimal Parallel Algorithm for Integer Sorting," *Proc. 26th IEEE Symp. on Foundations of Comp. Sci.*, 1985, pp. 496–504.

[22] J.H Reif and S. Sen, "Optimal Parallel Algorithms for Computational Geometry," to appear *Proc. 1987 IEEE Int. Conf. on Parallel Processing.*

[23] Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model," *Journal of Algorithms*, Vol. 2, 1981, pp. 88–102.

[24] L. Valiant, "Parallelism in Comparison Problems," *SIAM Journal on Computing*, Vol. 4, No. 3, September 1975, pp. 348–355.

[25] H. Wagener, "Optimally Parallel Algorithms for Convex Hull Determination," manuscript, 1985.

[26] C.K. Yap, "Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map," manuscript, 1987.

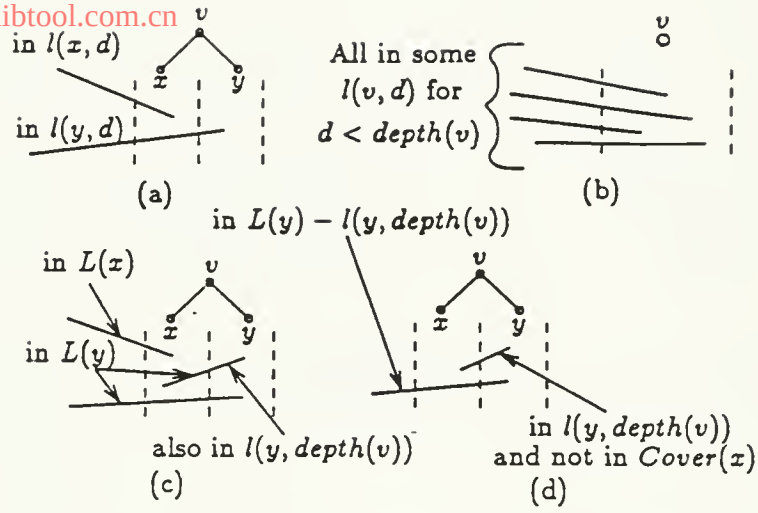

Figure 5: The plane-sweep tree equations. (a) $l(v,d) = l(x,d) \cup l(y,d)$, (b) $L(v) = l(v,0) \cup l(v, depth(v) - 1)$, (c) $L(v) = L(x) \cup (L(y) - (L(y) - l(y, depth(v))))$, (d) $Cover(x) = L(y) - l(y, depth(v))$.

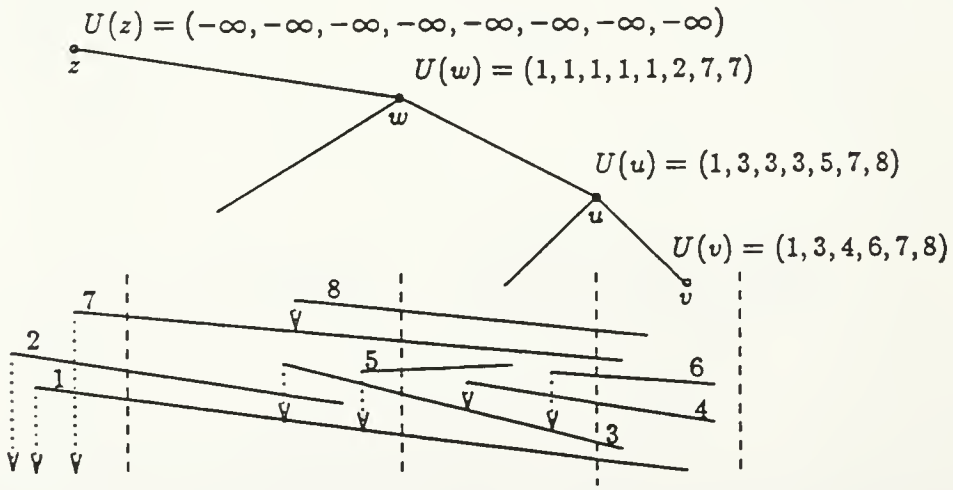

Figure 6: Segment identity changing during the cascading merge. We illustrate the way segment names change identity to that of their predecessor as we are performing the cascading merge. In this case we are constructing the $L(v)$'s. We denote the predecessor of each segment by a dotted arrow.

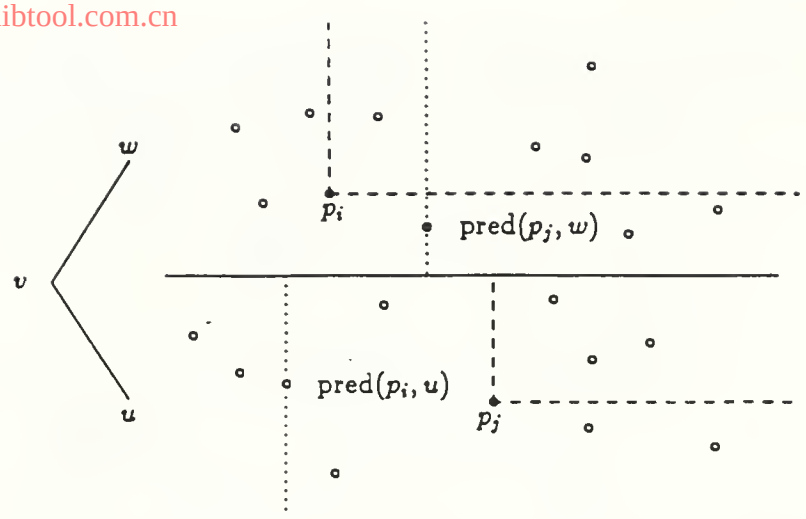

Figure 7: **The combining step for 3-dimensional maxima.** Points to the right of the dotted line 1-dominate p_i (resp. p_j), and points enclosed in the dashed lines 2-dominate p_i (p_j).

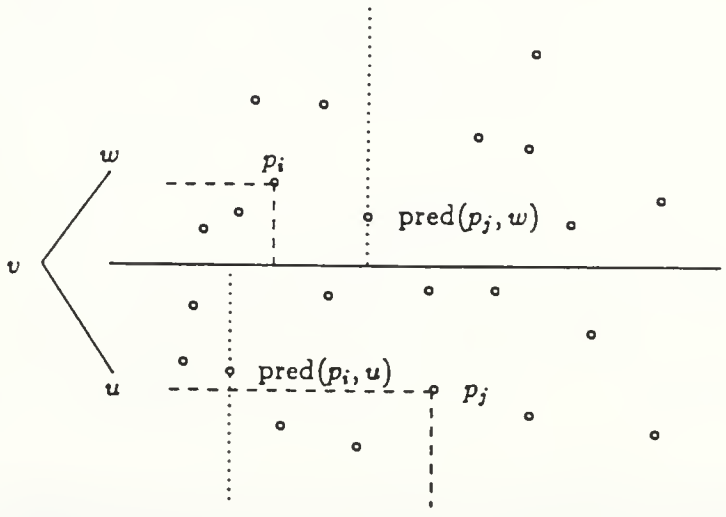

Figure 8: **The combining step for dominance counting.** Points to the left of the dotted line are 1-dominated by p_i (resp. p_j), and points enclosed in dashed lines are 2-dominated by p_i (p_j).

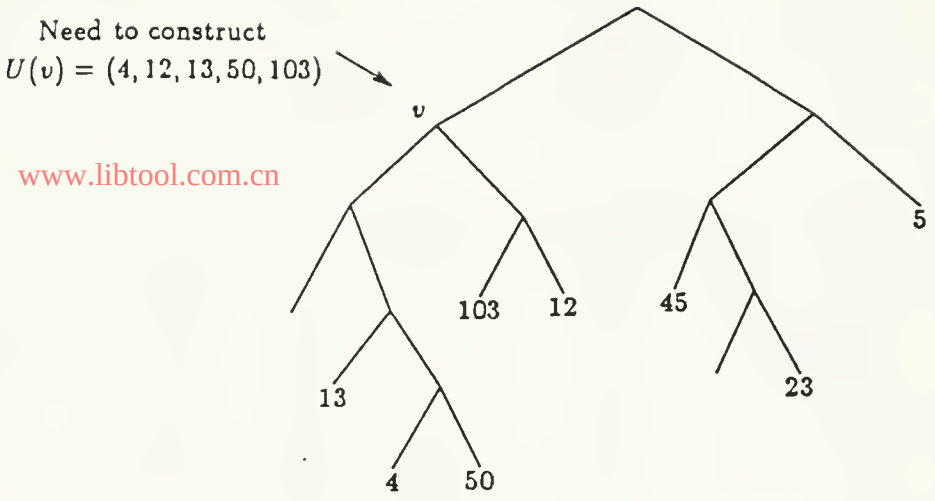

Figure 1: An example of the generalized merge problem.

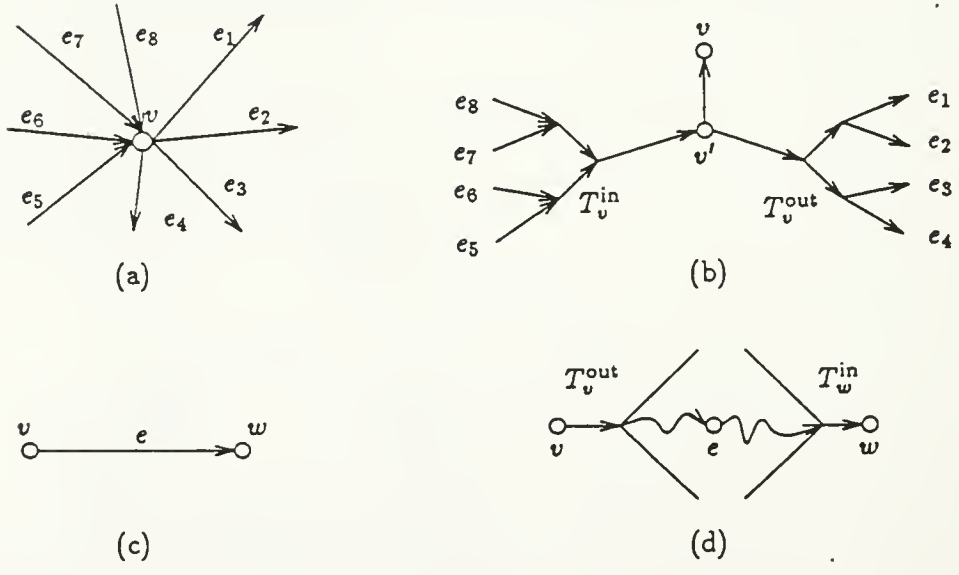

Figure 2: Converting G into a bounded degree graph $\hat{G}$. A node v in G (a) corresponds into a node v adjacent to its gateway v', which is connected to the fan-in tree and the fan-out tree for v (b). An edge e in G (c) is converted into a node in $\hat{G}$ which corresponds to a leaf node of two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

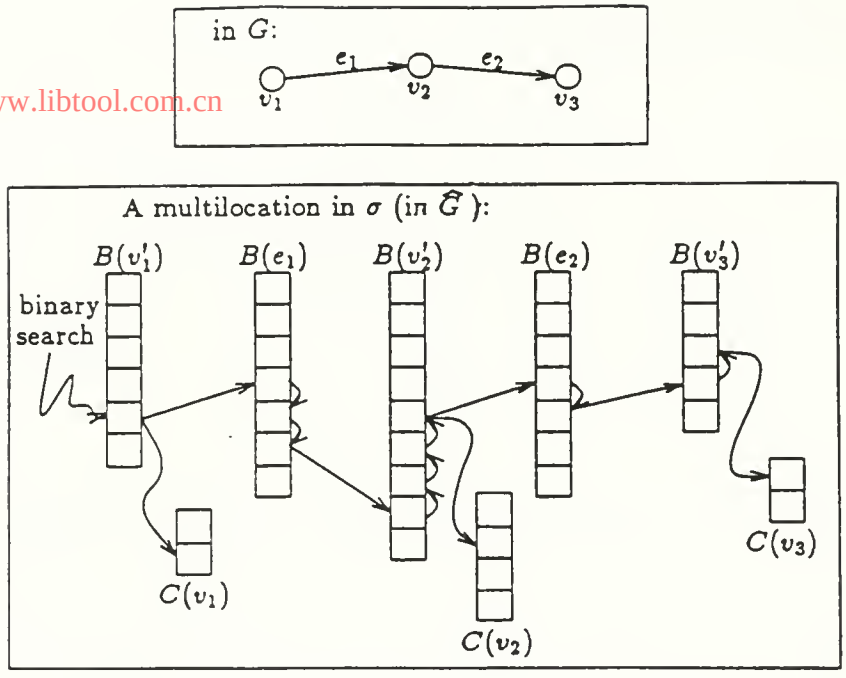

Figure 3: **A multilocation in a walk $\sigma = (v_1, v_2, v_3)$.**

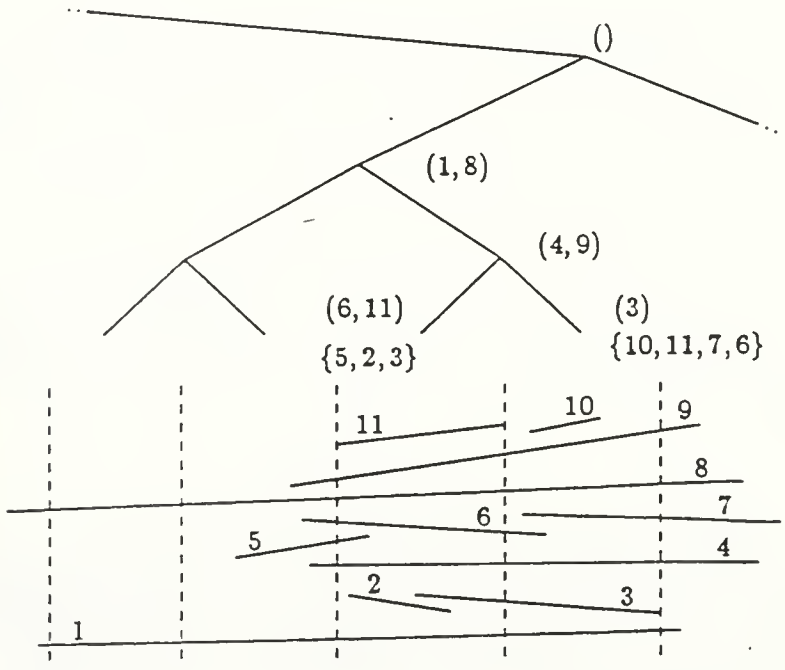

Figure 4: **A portion of a plane sweep tree.** The segments are numbered in this example by embedding the "above" relation of Section 2 in the total order $1, 2, \ldots, 11$. For simplicity we denote the list $Cover(v)$ by parentheses and the set $End(v)$ by set braces.

Errata:

Figures cited within the text were inadvertantly omitted. Please see the following list of page and figure numbers for the correct references.

(1) Page 4 should read: (See figure 1).

(2) Page 9 should read: (See figure 2).

(3) Page 10 should read: (See figure 3).

(4) Page 16 should read: (See figure 3).

(5) Page 17 should read: (See figure 4).

(6) Page 21 should read: (See figure 5).

(7) Page 24 should read: (See figure 6).

(8) Page 29 should read: (See figure 7).

(9) Page 31 should read: (See figure 8).

(10) Page 32 should read: (See figure 9).

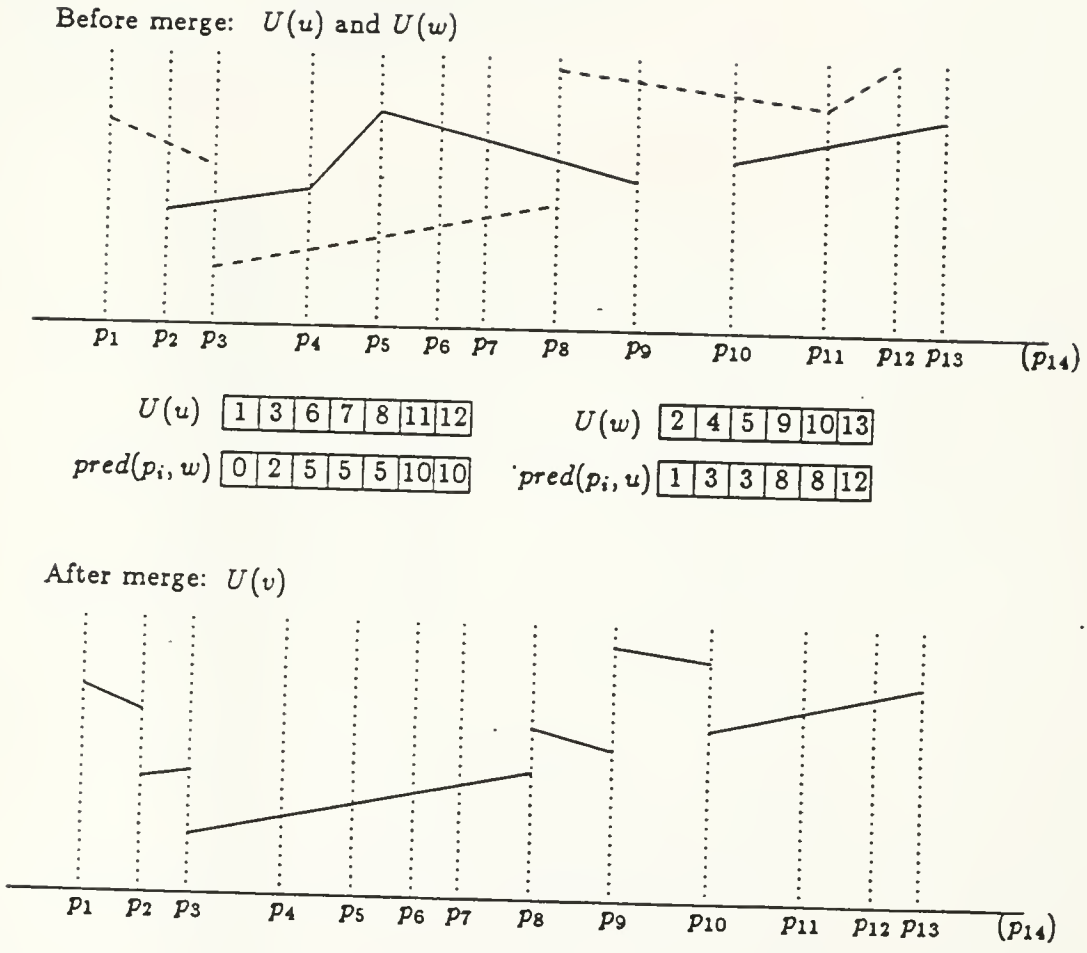

Figure 9: **An example of visibility merging.** The dashed segments correspond to the visible region for $X(u)$ and the solid segments correspond to the visible region for $X(w)$. For simplicity, we store the pointers $pred(p_i, u)$, and $pred(p_i, w)$ in arrays and denote each point p_i by its index 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).