Optimal Randomized Parallel Algorithms for Computational Geometry[1]

John H. Reif[2] and Sandeep Sen[2]

Abstract. We present parallel algorithms for some fundamental problems in computational geometry which have a running time of $O(\log n)$ using n processors, with very high probability (approaching 1 as $n \to \infty$). These include planar-point location, triangulation, and trapezoidal decomposition. We also present optimal algorithms for three-dimensional maxima and two-set dominance counting by an application of integer sorting. Most of these algorithms run on a CREW PRAM model and have optimal processor-time product which improve on the previously best-known algorithms of Atallah and Goodrich [5] for these problems. The crux of these algorithms is a useful data structure which emulates the plane-sweeping paradigm used for sequential algorithms. We extend some of the techniques used by Reischuk [26] and Reif and Valiant [25] for flashsort algorithm to perform divide and conquer in a plane very efficiently leading to the improved performance by our approach.

Key Words. Randomized, Parallel algorithm, Computational geometry, Point location, Triangulation, Trapezoidal decomposition.

1. Introduction. Recently there has been a growing effort toward developing efficient parallel algorithms for various fundamental problems in computational geometry. Aggarwal *et al.* [1] gave polylog algorithms for a various fundamental problems, namely Voronoi diagram, planar-point location, and triangulation among others. Atallah and Goodrich [6] improved these algorithms by reducing the running time of most of these algorithms to $O(\log n \cdot \log \log n)$ time using $O(n)$ processors. Hence, the previous efforts are suboptimal by at least a factor of $\log \log n$, since most of these problems need $\Theta(n \log n)$ time in the sequential case. For an optimal processor-time product the running time for most of these algorithms should be $O(\log n)$ using $O(n)$ processors.

Randomization has been successfully used in a wide number of applications (for example, see [25], [20], and [28]) and seems to be an ideal choice for optimizing algorithms for various problems in computational geometry. Clarkson [9], [10] used random sampling techniques to derive better upper bounds for algorithms for the post-office problem and higher-order Voronoi diagrams as well as some combinatorial quantities like k-sets and polytope separation. Haussler and Welzl

[1] This is a substantially revised version of the paper that appeared as "Optimal Randomized Parallel Algorithms for Computational Geometry" in the *Proceedings of the 16th International Conference on Parallel Processing*, St. Charles, Illinois, August 1987.

[2] Computer Science Department, Duke University, Durham, NC 27706, USA. This research was supported by DARPA/ARO Contract DAAL03-88-K-0195, Air Force Contract AFOSR-87-0386, DARPA/ISTO Contracts N00014-88-K-0458 and N00014-91-J-1985, and by NASA Subcontract 550-63 of Primecontract NAS5-30428.

Preprint of paper appearing in Algorithmica, Vol. 7, No. 1, January 1992, pp. 91-117.

[18] also used random sampling to solve half-space query problems. This is the first paper where random sampling methods have been applied to yield efficient parallel algorithms in computational geometry. Clarkson's methods yielded bounds for the *expected* running time of the algorithms. In contrast, we are able to bound the tail estimates of the running time with high probability. It is not clear how Clarkson's algorithms can be modified to obtain high-probability bounds without increasing the asymptotic running time of the algorithms (i.e., since the algorithms terminate in the claimed time bounds with constant probability, it is necessary to repeat the algorithm $O(\log n)$ times to decrease the failure probability to $1/n$). Although this may not be very important in a sequential environment, it turns out to be very crucial for bounding the resources used by a parallel algorithm. This becomes clearer in the proof of Theorem 2, where expected bounds do not appear to be sufficient for deriving the claimed bounds.

The term *very high likelihood* (*probability*) has been used in this paper to denote probability $> 1 - n^{-\beta}$ for some $\beta > 1$ where n is the input size. Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, we use $\tilde{O}$ to represent complexity bounds of the randomized algorithms. We say that a randomized algorithm has resource bound $\tilde{O}(f(n))$ if there is a constant c such that the resource used by the algorithm is not more than $c\alpha f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$. (An equivalent definition is bounding the resource by $\alpha \cdot f(n)$ with probability greater than $1 - n^{-c\alpha}$ and in the rest of the paper these definitions are used in an interchangeable manner.)

We have used some known techniques methods like *random-mate* and extended techniques of *Flashsort* [25] to two dimensions successfully to reduce the running time of a wide variety of fundamental problems. Random-mate was introduced in [21] to discard a constant fraction of zeros in a zero–one string in constant time, and was also later used by Gazit [17]. Random sampling methods used by Reischuk [26] and Flashsort provided the initial inspiration for divide and conquer on the plane; although the problem at hand being much harder, required several additional techniques. We note that Clarkson [11] independently derived bounds for doing divide and conquer efficiently for geometric problems in a more general setting.

The algorithms in this paper have optimal speedup when compared with their corresponding sequential cases. Table 1 summarizes the results in this paper and compares them with the previously[3] best-known results. The organization of the paper is as follows: In Section 1 we present a very simple algorithm for planar-point location which demonstrates the usefulness of randomization. In Section 2 we present a new data structure which can be constructed efficiently using random sampling, and we call it the *nested-plane-sweep* tree. In the following section we apply this data structure to develop optimal algorithms for a number of problems for which only efficient deterministic algorithms were known. In the final section we reduce problems like three-dimensional dominance and range counting to integer sorting which can be then be optimally solved using an algorithm similar to that of Reif [24]. Since the currently best-known algorithm for integer sorting

[3] Optimal deterministic algorithms for these problems were obtained independently by Atallah *et al.* [3]. Also, see Section 6 for some additional comments.

Table 1. Summary of our results.

Problem	Previous bounds	Our bounds
Planar-point location	$O(\log n \log\log n)$	$\tilde{O}(\log n)$
Trapezoidal decomposition	$O(\log n \log\log n)$	$\tilde{O}(\log n)$
Triangulation	$O(\log n \log\log n)$	$\tilde{O}(\log n)$
Three-dimensional maxima	$O(\log n \log\log n)$	$O(\log n)$
Two-set dominance counting	$O(\log n \log\log n)$	$O(\log n)$
Multiple range counting	$O(\log n \log\log n)$	$O(\log n)$
Visibility	$O(\log n \log\log n)$	$\tilde{O}(\log n)$

is optimal for word-size n^{ε}, for any $\varepsilon > 0$, our algorithms are not optimal in an information-theoretic sense (i.e., the sequential algorithms use only $O(\log n)$ bits per word). However, we feel that these are extremely simple and should be of interest from a practical viewpoint.

2. Point Location in Planar Subdivisions in Logarithmic Time with High Probability. A PSLG (planar straight line graph) is a connected graph which can be embedded on a plane using only straight line edges. Given a PSLG and a query point, the point-location problem is to identify the subdivision in the plane (induced by the PSLG) which contains the query point. If a PSLG has n vertices, in the sequential (uniprocessor) case, the asymptotic optimal query time performance of $O(\log n)$ can be achieved by a variety of approaches ([19] and [16] among others). The performance is achieved by binary search on a well-organized data structure constructed on the PSLG during the preprocessing. The preprocessing cost for the best-known sequential algorithms for point location is $\Omega(n \log n)$ operations, and thus the best parallel time for this problem cannot be better than $O(\log n)$ time using n processors. We propose a randomized method by which the preprocessing can be done in $O(\log n)$ time with very high probability using $O(n)$ processors given a PSLG which has only convex subdivisions. A similar algorithm has also been independently proposed by Dadoun and Kirkpatrick [14].

2.1. The Sequential Algorithm. Kirkpatrick [19] proposed a triangulation refinement technique which builds a hierarchy of triangulated PSLG from the initial graph. Unfortunately, his method does not appear to be directly parallelizable. A review of his method will help the reader to understand the parallel version presented here. Starting with a triangulated PSLG (i.e., all faces including the exterior face are triangular), his algorithm removes an independent set of vertices, and retriangulates the remaining graph. In addition, it keeps track of the set of the eliminated triangles that have nonempty intersection with each of the new triangles of the retriangulated graph. This procedure is repeated successively until we are left with a constant number of triangles. Actually we can stop when the problem size is reduced to $O(\log n)$. The search proceeds in a hierarchical manner from the highest level, where the problem can now be solved in constant time $O(\log n)$ time if we stop at an $O(\log n)$-size graph). Subsequently, at each level we relocate the point with respect to the triangles which intersect the triangle in the current level, thus refining the regions of location at each level. At the lowest level

the point is located with respect to the regions of the original PSLG and the search is complete. The search data structure is a directed acyclic graph (not a tree). Each of the nodes represents a triangular region and is connected by directed arcs to all the triangular regions that it intersects in the previous level. At any node, the algorithm determines in which of its chidren the point lies. Notice that a node can have more than one parent and hence the graph (the underlying search structure) is not a tree. The efficiency of this approach depends on the number of levels of the triangulated PSLG, and the number of intersections of each triangle with the triangles in the next lower level. By guaranteeing that a constant fraction of the vertices, and hence the triangles (since it is a planar graph), are eliminated at each successive level a logarithmic level search is achieved. However, in doing so we must also be careful that a triangle intersects a maximum of a constant number of triangles in the next level (i.e., each node has a bounded constant degree so that a constant time is spent at each search level). Both of the above criteria can be satisfied by identifying an independent set of vertices each having a degree $\leq d$, where d is a fixed constant. The number of edges in a planar triangulated graph (V, E) is $3|V| - 6$ from Euler's formula (assuming that the exterior face is also a triangle). This adds up to a total vertex degree of $6|V| - 12$, giving us a lower bound on the number of vertices with degree less than d. Thus the number of vertices with degree less than d is at least $6|V|/d - 2$ in a triangulated graph for $d > 6$ (a typical value of d is 12). Since a constant fraction of these vertices can be eliminated at each stage by identifying a maximal independent set of vertices, the height of the search tree is at most $O(\log|V|)$. The preprocessing time is dominated by the identification of this independent set of degree $\leq d$, which may cost $O(n)$ time ($n = |V|$) at each level resulting in a total time of $O(n \log n)$.

In the discussion below we present a randomized method to identify a large independent set in constant time for each level using n processors with a high probability. As a result, the search structure can be constructed in $O(\log n)$ time.

2.2. Choosing a Large Independent Set with High Probability. Associate a processor with each vertex and each edge of the PSLG (V, E). From Euler's formula we need only $O(|V|)$ processors for the PSLG. An independent set consisting of vertices of degrees less than or equal to d can be identified in the following manner.[4]

Algorithm Random-mate
Input: A PSLG in the form of a doubly connected edge list (DCEL).
Output: A large independent set of vertices with degree $\leq d$.

(1) Identify the set of vertices with degree $\leq d$. This can be done in constant time using one processor for each vertex. There will be at least $6|V|/d$ (ignoring the small constant) such vertices (from the previous discussion).
(2) This has two phases:
   (a) Randomly assign a tag "male" or "female" with equal probability to each selected vertex from step (1). The "males" constitute the candidates for an

[4] This is the random-mate technique mentioned earlier.

independent set. For each edge between two "males" pronounce both vertices "dead" by marking their corresponding cells. This is easily done by using one processor for every edge with concurrent-write capability. Note that it is very easy to get rid of the concurrent-read feature since a vertex has at most degree d and hence this step can be carried out in constant number of steps (instead of exactly one step).

(b) The "males" which are not "dead" form an independent set.

(3) Output the set X of "males" which are not "dead." The set X is an independent set of vertices having degree less than d.

It is obvious that the algorithm gives us an independent set which is possibly smaller than the maximal independent set (a null set in the worst case for a chain of males). However, we shall show that on the average this technique will produce an independent set proportional to a constant fraction of the total number of nodes (vertices of degree $\leq d$) with a very high probability.

LEMMA 1. *Let $n = |V|$ and let n_d denote the number of vertices with degree less than or equal to d ($n_d \geq 6n/d$ from* [19]). *There exist constants c, v ($0 < v < 1$) such that $P(|X| < vn) \leq e^{-cn}$.*

PROOF. Assuming equal probabilities for a vertex being assigned a "male" and a "female" tag, the probability of a vertex being in the independent set is greater than or equal to $(\frac{1}{2})^{d+1}$ (since it has $\leq d+1$ neighbors). We consider an independent set $I_3(n_d)$ of vertices which are at a distance 3 (i.e., the shortest distance between any two vertices in this set is three edges). Since the graph had a bounded degree d, the selection of each vertex can prevent the selection of at most $d_{\max} = (d-1)^3 + (d-1)^2 + (d-1)$ other vertices. Thus the cardinality of this set is at least $(1/d_{\max})6n/d$ or $6n/D$ (D is a constant). The event of a vertex $v \in I_3(n_d)$ being in X (independent set) is *independent* of any other vertex in $I_3(n_d)$. Thus the distribution of the cardinality of the independent set produced by Algorithm Random-mate can be bounded from below by a binomial distribution with $p = (\frac{1}{2})^{d+1}$, and mean $\mu = |I_3(n_d|p \geq 6np/D$.

Using Angluin–Valiant's bounds (see equation (3) in the Appendix), for $0 < \varepsilon < 1$,

$$\text{Prob}[|X| \leq (1-\varepsilon)\mu] < \exp(-\varepsilon^2\mu/2).$$

Using $v = (1-\varepsilon)6/D$ and $c = \varepsilon^2/(12D)$ we arrive at the required form. □

2.3. A Parallel Algorithm. We have shown that with very high probability, the size of the independent set will be greater than a constant fraction of n, which is the key to finding a logarithmic depth search structure. We present the complete Point-Location-Tree algorithm below

Procedure Point-Location-Tree

(0) Let N be the number of vertices in the current stage. If $N \leq k \log n$ (for some fixed constant k), then halt. (Then we can locate the query point in the $O(\log n)$ triangular faces in $O(\log n)$ time using a single processor.)

(1) Assume that the given PSLG is triangulated. (If not, then we can use the procedure for triangulation described in the latter sections to triangulate a PSLG in $O(\log n)$ time using $O(n)$ processors.)
(2) Choose an independent set of vertices each of which has degree ≤ 12 using the method described above with $d = 12$. Time $= O(1)$.
(3) Triangulate the remaining PSLG (after the removal of the independent set). Note that removal of a vertex of degree d ($d \leq 12$) necessitates triangulating a simple polygon of d vertices. This can be done in constant time using one processor for each polygon.
(4) Determine for each of the new triangles (created in step (3)), which of the old triangles it intersects. This can also be done in constant time using one processor for each of the new triangles using concurrent read.
(5) Go to (0).

THEOREM 1. *Algorithm Point-Location-Tree runs in* $\tilde{O}(\log n)$ *time using* $O(n)$ *processors and* $O(n \log \log n)$ *space in a CREW PRAM model. Furthermore, if the input* PSLG *is triangulated, we can reduce the processor bound to* $O(n/\log n)$ *without increasing the asymptotic running time of the algorithm.*

PROOF. Each of the steps at any level of recursion of the algorithm can be done in constant time using $O(n)$ processors. From Lemma 1, there exist constants v and c such that the probability of reducing the problem by a constant fraction $(1 - v)$ is very high. Let n_i denote the problem size (the number of vertices remaining in the graph) at stage i of recursion, where $n_0 = n$. We show that after $O(\log n)$ stages, the problem size is less than $O(\log n)$ with very high likelihood. From Lemma 1, $P[n_i \leq (1 - v)n_{i-1}] \geq 1 - n^{-cK}$ for $n_{i-1} > K \log n$ for some constant K. The probability that this fails to hold in any level i ($1 \leq i \leq \log_{1/(1-v)} n$) is less than $n^{-cK+\delta}$ for any $\delta > 0$. We abort the algorithm if the number of vertices is greater than $K \log n$ after the last stage and rerun the entire procedure. The argument for space is similar to the sequential case. We need an extra $O(\log \log n)$ factor space for the triangulation procedure.

If the input PSLG is triangulated, the number of operations performed is linear in n (follows from the sequential algorithm) and hence by Brent's slow-down procedure and using the load-balancing scheme of [13] or the randomized scheme of [21], the processor bounds can be reduced to $O(n/\log n)$ without affecting the asymptotic run-time. □

REMARK. Dadoun and Kirkpatrick [14] show that their algorithm runs in $O(\log n)$ expected parallel time. However, they do not extend their analysis for the high-probability bounds which is also $O(\log n)$ as shown in the previous theorem.

COROLLARY 1. *Given a planar subdivision S, consisting of* $O(n)$ *vertices, we can locate* $O(n)$ *query points in* $\tilde{O}(\log n)$ *time using* $O(n)$ *processors.*

PROOF. Follows immediately from the previous discussion. □

A direct application of this result leads to the improvement in performance of Aggarwal *et al.*'s [1] two-dimensional Voronoi diagram algorithm.

COROLLARY 2. *The Voronoi diagram of a set of $O(n)$ points on a plane can be constructed in $O(\log^2 n)$ expected running time using $O(n)$ processors in a CREW PRAM model.*

PROOF. Aggarwal *et al.* [1] use a divide-and-conquer algorithm, where in each of the log n stages of recursion we need to perform point location simultaneously for $O(n)$ vertices. Since their planar-point location needs $O(\log^2 n)$ time, the overall algorithm runs in $O(\log^3 n)$ time. Using our randomized planar-point-location algorithm, the expected running time of the algorithm is $O(\log^2 n)$. □

3. Constructing the Plane-Sweep Tree in $O(\log n)$ Parallel Time Using Randomization. There are many problems in computational geometry whose optimal sequential algorithms use the plane-sweeping paradigm. Aggarwal *et al.* [1] had proposed a data structure, which they call *plane-sweep-tree*. Using this, they were able to solve a number of problems efficiently (such as planar-point location and triangulations among others) in parallel using a linear number of processors. This data structure is similar to the *segment-tree* (proposed by J. Bentley and is also discussed in [22]), the intervals of which are induced by the projection of the endpoints of the input set of segments on one of the axes. This was used to emulate the plane-sweeping paradigm. The idea of using the plane-sweep tree was later further developed by Atallah and Goodrich [6]. We review some definitions and results from their papers as they relate to our work.

3.1. Review of Plane-Sweep Tree. Let T be a complete binary tree with its leaves corresponding to the $2e + 1$ intervals formed by projecting the $2e$ endpoints of a given set of e line segments onto the x-axis. Associated with each node v in the tree is an interval $[a_v, b_v]$ on the x-axis corresponding to the union of intervals of its descendants. Let Π_v represent the vertical strip $[a_v, b_v] \times (-\infty, \infty)$. A segment s_i *covers* a node v in T if its projection $X(s_i)$ on the x-axis spans the interval $[a_v, b_v]$ (corresponding to v) but does not span the interval of v's parent node. It can be easily shown that no segment s_i covers more than two nodes at any level and hence no more than $O(\log n)$ nodes of the tree.

For each node v of T, we keep track of the the following properties:

$H(v) = \{s_i | s_i \text{ covers } v\}$.

$W(v) = \{s_i | s_i \text{ has at least one endpoint in } \Pi_v\}$.

$L(v) = \{s_i | s_i \in W(v) \text{ and } s_i \text{ intersects the left boundary of } \Pi_v\}$.

$R(v) = \{s_i | s_i \in W(v) \text{ and it intersects the right boundary}\}$.

$I(v) = \{s_i | \text{the left endpoint of } s_i \text{ is in } \Pi_{\text{leftchild}(v)}$ and the right endpoint is in $\Pi_{\text{righchild}(v)}\}$.

If the given set of segments is nonintersecting (as in the case of a polygon), then these sets are completely ordered on the y-coordinate. The above quantities are computed while constructing the plane-sweep tree, which are used to search for segments directly above or below a given query point.

MULTILOCATION. Given a plane-sweep tree, this procedure locates, for each input point p, the segment in $H(v)$ which is directly above (or below) p for all vertices v in the tree such that p belongs to the interval Π_v.

AUGMENTED PLANE-SWEEP TREE. This is a data structure obtained from a plane-sweep tree by adding pointers to every node such that, given the position of an element in any node, its position in the parent node can be located in an additional constant time. This data structure enables us to perform fractional cascading.

3.2. Development of the New Data Structure

FACT 1 [6]. Multilocation of any query point can be done in $O(\log n)$ sequential time in an augmented plane-sweep tree.

The preprocessing cost of building this data structure was $O(\log^2 n)$ time using $O(n)$ processors and $O(n \log n)$ space. Consequently, the above-mentioned problems needed $\Omega(\log^2 n)$ time. Atallah and Goodrich [6] improved the preprocessing time for constructing the plane-sweep tree to $O(\log n \log\log n)$ using parallel merging techniques of Borodin and Hopcroft [7] while keeping the number of processors and the space requirement unchanged. The dependence of their approach on parallel merging can be summed up in the following manner:

FACT 2. The plane-sweep tree of k segments can be constructed in $O(f(p, k) \log k)$ time using p processors ($p \geq k$), where $f(p, k)$ denotes the time complexity of merging two sorted lists of $O(k)$ elements using p processors.

Since $f(k, k) = \Theta(\log\log k)$ (see [27] and [7]), the bound $O(\log n \log\log n)$ of Atallah and Goodrich's *Build-Up* algorithm for plane-sweep tree follows. Moreover, it enables us to state a useful lemma:

LEMMA 2. *The plane-sweep tree of n segments can be constructed in* $O(\log n)$ *time with* $O(n^{1+\varepsilon})$ *processors using Atallah and Goodrich's* [6] *approach.*

The proof follows from the well-known result [27], [7] that two sorted lists of $O(n)$ elements can be merged in constant time using $n^{1+\varepsilon}$ processors. Using $f(p, k) = O(1)$ in Fact 1 yields the required result.

An important common characteristic of Atallah and Goodrich's [6] algorithms is that the time complexity is dominated by the preprocessing cost. More specifically, given the precomputed data structure, triangulation and planar-point

location can be done in $O(\log n)$ time using n processors in a CREW model. Thus, it is worthwhile to try to improve the preprocessing time to $O(\log n)$ so that the overall running time of the algorithms can be reduced to $O(\log n)$. Our approach avoids merging two $O(n)$-size lists by using random sampling to build up a new data structure called the *Nested-Plane-Sweep Tree* in $O(\log n)$ time with very high probability. The overall running time of our algorithms is $\tilde{O}(\log n)$ with $O(n)$ processors.

We now give an outline of our method, which will be formalized later. The techniques used are similar to the random sampling used in the Flashsort algorithm of Reif and Valiant [25]. The use of random splitting is extended to two dimensions. For this, we choose randomly a sample of $\sqrt{n}$ segments from the given set of n segments. (We see later that we choose only an n^{ε_0} size sample, $0 < \varepsilon_0 < 1$, and the actual value of ε_0 will be less than $\frac{1}{13}$. All the arguments in this section apply for any ε_0 which has been set to $\frac{1}{2}$ for ease of presentation.) Every segment has an equal probability, i.e., $1/\sqrt{n}$ of being selected. We build up the augmented plane-sweep tree of Atallah and Goodrich [6] on this sample of $\sqrt{n}$ segments using $O(n)$ processors. The randomly chosen set of $O(\sqrt{n})$ segments partitions the plane into $O(\sqrt{n})$ disjoint regions (proof follows later). Each of the remaining $O(n)$ segments is located (using the multilocation algorithm of Atallah and Goodrich) in the $O(\sqrt{n})$ regions, in $O(\log n)$ time, using n processors. Though each segment may be divided into a number of smaller segments by the intervals induced by the plane-sweep tree of the sample set, the expected number of segments in each region will be shown to be $O(\sqrt{n})$ and less than $O(\sqrt{n}\log n)$ with very high probability. The algorithm can now be applied recursively to the subproblems in each of the $O(\sqrt{n})$ subregions, resulting in a recurrence equation of the form $\bar{T}(n) = \bar{T}(\sqrt{n}\log n) + O(\log n)$ which has a solution $\bar{T}(n) = O(\log n)$. A stepwise description of the algorithm is given below.

Procedure Nested-Sweep Tree

Input: Set S of nonintersecting line-segments $l_1, l_2, \ldots, l_n$ in a plane.

(1) Choose a set of $\sqrt{n}$ segments randomly from the original set. Each segment is selected with equal probability of $1/\sqrt{n}$.

(2) Use the *Build-up* and *Augment* algorithms of Atallah and Goodrich [6] to construct the augmented plane-sweep tree. Time $= O(\log n)$, space $= O(\sqrt{n}\log n)$.

(3) Locate the $n - \sqrt{n}$ segments in the plane-sweep tree, i.e., identify the region (the $O(\sqrt{n})$ regions) into which the sample set divides the plane) where each of the segments lie. Note that a segment may lie in more than one region in which case it is broken into a number of smaller segments each of which lies in only one region. This step uses Atallah and Goodrich's [6] *multilocation* algorithm. The segments which lie completely within a region can be located using the algorithm directly. However, segments that cross the boundaries can

be split into $O(\sqrt{n})$ broken segments (in the worst case) are located using a scheme described later.

(4) If the number of segments in any region is more than a threshold, then apply procedure Nested-Sweep Tree to each of these regions. (The threshold can be chosen to be $O(\log^r n)$ for some nonnegative integer r.)

Step (3) needs further explanation. For this we need a few probabilistic lemmas which bound the size of each subproblem and the total size of all the subproblems at any level of recursive call.

3.3. *Random-Sampling Methods and Probabilistic Bounds*

LEMMA 3. *The plane-sweep tree of the random sample of $\sqrt{n}$ segments divides the plane into less than or equal to $3\sqrt{n}$ trapezoidal regions.*

PROOF. Assuming distinct endpoints there are $2\sqrt{n}$ vertical segments through the endpoints, each of which can be shared by three such trapezoidal regions (see Figure 1). Moreover, each trapezoidal region is bounded by at most two vertical segments. Thus, there are at most $3\sqrt{n}$ such regions. □

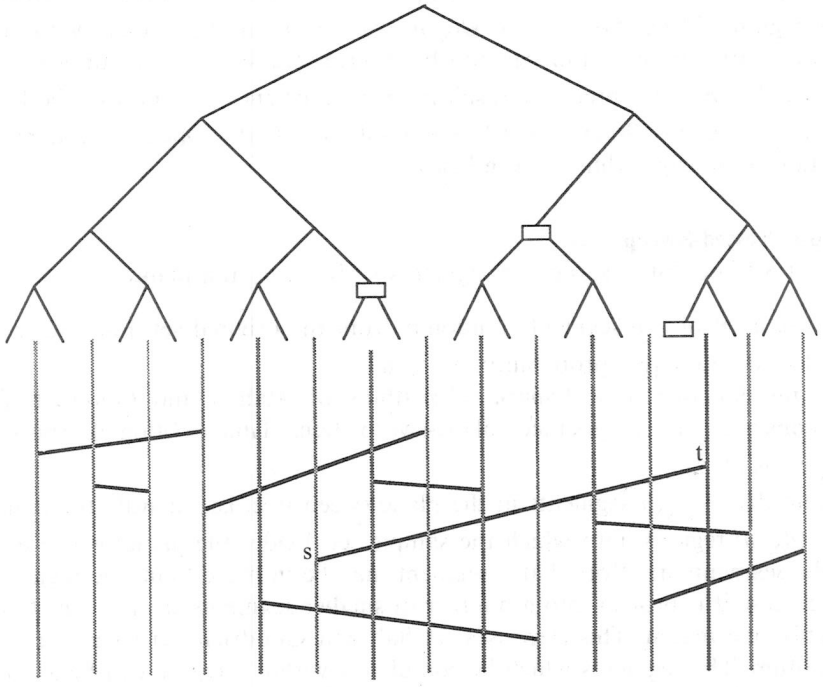

Fig. 1. The skeleton of a plane-sweep tree. The circled nodes have portions of the segment s–t (corresponding to the intervals covered by the nodes).

To bound the number of segments lying in a given trapezoidal region, we classify the segments into two groups:

(a) Segments that have at most one of their endpoints inside the region.
(b) Segments that lie completely within the region.

(a) The segments that lie partially in the region intersect either the left boundary, or the right boundary or both boundaries of the region. Consider an ordered list of the segments $L_U(i)$ intersecting the vertical segment passing through an endpoint in the upward direction. These segments can be completely ordered with respect to the ordinate (on y). The probability that this ordered list exceeds m is less than $(1 - 1/\sqrt{n})^m$ because each segment has probability $1/\sqrt{n}$ of being chosen in the sample. Let us denote by E_i the event that $m < k\sqrt{n} \log n$ (for some $k > 1$) for a fixed endpoint i. The probability that the event occurs for any endpoint (and there are $2n$ such endpoints) can be upper-bounded by $\sum_{i=1}^{2n} e^{-k \log n}$. For $k > 2$, the event that the ordered list of line segments $L_U(i)$ is less than $k\sqrt{n} \log n$ for all i (which is the complement of the previous event) holds with very high likelihood. Thus the number of segments intersecting any boundary is less than $O(\sqrt{n} \log n)$ with high likelihood (this is only a subset of the previous event).

Moreover, the expected number of line segments that intersect a boundary can be upper-bounded by $\sqrt{n}$ (mean of geometric distribution). Note that the random variables representing the number of segments intersecting each of the regions are not independent. Since the expectation of the sum is the sum of expectation, the total number of such segments over all the regions is expected to be $< 12n$ (since a trapezoid has at most two vertical segments going up and two going down and there are $< 3\sqrt{n}$ trapezoids). It follows that the probability that the total number of lines intersecting all the trapezoids exceeds $k_{\max} n$ ($k_{\max} > 24$) is less than $\frac{1}{2}$. This implies that by repeating the experiment of choosing $\sqrt{n}$ segments $c \log n$ times, the probability of failure (where a success is the event that the total number of intersections is less than $k_{\max} n$) is less than $1/n^c$.

If we choose independently $p(n) = O(\log n)$ sets of samples, one of them is good with very high likelihood. However, to determine if a sample is "good," we would have to carry out step (3) $O(\log n)$ times, each of which requires $O(\log n)$ time (such a method is described in Section 3.4). Instead, we try to estimate the the number of segments intersecting a trapezoid T_i using only a fraction of the input segments. For example, we can choose $c_0 \cdot n/\log^d n$ (for some fixed integer $d > 2$ and a constant c_0 (which will be determined from the required success probability of the algorithm) of the input segments randomly for the jth sample, R_j. Let X_i^j be the number of segments intersecting trapezoid T_i corresponding to sample R_j, $1 \leq j \leq b \log n$ (b is a fixed integer greater than 0) and let A_i^j be the number of segments intersecting T_i out of the $n/\log^d n$ randomly chosen input segments for the same sample. Clearly, A_i^j is a binomial random variable with parameters $c_0 \cdot n/\log^d n$, X_i^j/n. Assuming that X_i^j is greater than $\bar{c} \cdot \log^{d+1} n$, for some constant $\bar{c}$, we can apply Chernoff bounds (see the Appendix) to tightly bound the estimates within a constant multiplicative factor. Since we do it only for $1/\log^d n$ of the input segments, the total work done for the $O(\log n)$ random subsets does not change

the total work done (which we show to be $O(n \log n)$ in the next section). (Note that $X_i^j < \bar{c} \log^d n$, it is an easy case since $\sqrt{n} \cdot p \log^d n = o(n)$.)

More formally, by invoking Chernoff bounds (see equations (1) and (2) of the Appendix), for any $\alpha > 0$ (α is a function of c_0), there exists a c_1, independent of n,

$$\text{Prob}(A_i^j \leq \alpha c_1 X_i^j / \log^d n) \leq 1/n^\alpha$$

and

$$\text{Prob}(A_i^j \geq c_2 \alpha c_0 \cdot X_i^j / \log^d n) < 1/n^{c_0 \alpha} < 1/n^\alpha \qquad (\text{for } c_0 > 1).$$

From the last two inequalities, X_i^j is bounded by $L^j = A_i^j \log^d n / c_0 c_2 \alpha$ from below, and by $U^j = A_i^j \log^d n / c_1 \alpha$ from above. With appropriate changes in the constants, this condition holds with high likelihood (as defined in Section 2.1) for all X_i^j simultaneously. We do the procedure (described in the next section) simultaneously for all the samples R_j and choose the sample R_{j_0} using the following simple test:

Algorithm *Sample-select*
(Let $N^j = \sum A_i^j$ and let the actual number of intersections be denoted by T^j and the upper and lower bounds obtained from N^j by U^j and L^j respectively.)

If $k_{\text{total}} n > U^j$, then accept sample R^j (since $k_{\text{total}} n \geq U^j \geq T^j$), else
if $k_{\text{total}} n \leq L^j$, then the sample is "bad" (since $k_{\text{total}} n \leq L^j \leq T^j$), else
if $L^j \leq k_{\text{total}} n \leq U^j$, then accept the sample R^{j_0} for which E^{j_0} is minimum. Since both $k_{\text{total}} n$ and T^{j_0} lie in this interval this guarantees that $T^{j_0} \leq c_3 \cdot k_{\text{total}} n$ where $c_3 = U^j / L^j$ which is a constant.

Recall that from our earlier discussion at least one of the samples would satisfy the first or third condition with very high likelihood.

(b) The $O(\sqrt{n})$ subdivisions are trapezoidal (see Figure 2) regions which can be specified by at most four segments. Thus, there are $O(n^4)$ potential trapezoids of which only $O(\sqrt{n})$ were chosen in the random sample (Lemma 3). We call a trapezoid "good" if it has less than $O(\sqrt{n} \log n)$ segments within it and "bad" otherwise. Since the probability that any chosen trapezoid containing more than m segments is less than $(1 - 1/\sqrt{n})^m$, the bounds of (a) hold. Thus all the chosen trapezoids are "good" with very high probability (the number of potential trapezoids from $\sqrt{n}$ segments is less than n^2). The total number of segments lying within all trapezoids is obviously less than n (from the input size).

We can summarize the above observations in the following lemma:

LEMMA 4. *There exist constants α, $\bar{c}$ such that the probability of any one of the $O(\sqrt{n})$ trapezoidal regions containing more than $O(\bar{c}\alpha\sqrt{n} \log n)$ segments (or more appropriately broken segments) is less than $n^{-\alpha}$. Moreover, the total number of such segments over all trapezoidal regions is less than $l_{\max} n$ with very high likelihood.*

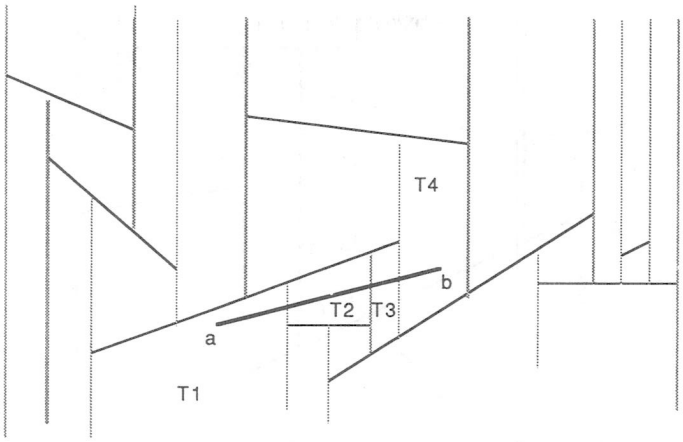

Fig. 2. Segment a–b is multilocated in the trapezoidal regions labeled $T1$, $T2$, $T3$, and $T4$.

The success of the recursive calls at any level can be argued in an identical manner except that we allow the probability of success at level i to diminish to $1 - n^{-1/2^{i-1}\alpha}$. For example, in level i, we do the resampling only $\log n/2^i$ times.

3.4. Partitioning Segments into Trapezoidal Regions. We use a *locus-based* approach to solve this problem. This approach involves considering each query as a higher-dimensional point and partitioning the underlying space into regions providing the same answer. Thus the query problem is reduced to a point-location problem, given sufficient preprocessing time and space. The problem at hand involves preprocessing the trapezoidal subdivisions (induced by the sample) in such a manner that given the endpoints of any segment we should be able to list the regions it intersects in $O(\log n)$ time using $[k/\log n]$ processors where k is the number of regions that it intersects. We show that the preprocessing for n segments can be done in $O(\log n)$ time using $O(n^c)$ processors, where c is a fixed constant. Thus we can choose any sample of size less than $n^{1/c}$.

Since the input segments are nonintersecting, these can only extend between regions where there exists a "clear path" which does not intersect any other sample segment. It must also be clear that there can exist more than one such "clear path" between two regions. Our objective is to partition the plane into regions (equivalent classes), such that given any fixed pair of such regions (where the endpoints of a segment lies), the regions that the segment intersects can be predetermined. The boundaries of these "clear paths" are straight lines joining vertices of the trapezoidal subdivisions and the equivalent regions are intersections of the half-spaces defined by these boundaries. Since we need a fast preprocessing procedure, we settle for a finer partition of space (i.e., more than one pair of partitioned regions may intersect exactly the same set of trapezoids). If n is the size of the trapezoidal regions, there are $O(n)$ vertices which gives rise to at most (n^2) boundary conditions which are straight lines joining every pair of vertices. Some of them may intersect sampled segments and hence are not boundaries of

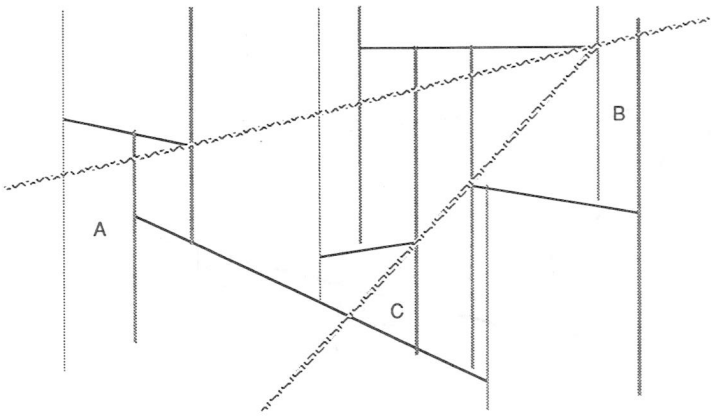

Fig. 3. There is a clear path between regions A and B whereas there cannot be a segment with endpoints in regions C and B. This does not affect the asymptotic complexity of the algorithm, i.e., there is some redundant work being done since some of the pairs of regions do not have any feasible segments.

"clear paths" but it does not affect our procedure since they would only make the partition more fine. (See Figure 3 for an illustration.) These $O(n^2)$ lines can intersect in $O(n^4)$ vertices, giving rise to $O(n^4)$ regions in the partition. This implies that there are $O(n^8)$ possible pairs of regions where endpoints of a segment could lie and we can precompute for each such pair which trapezoids the corresponding segment intersects using $O(n^9)$ processors in $O(\log n)$ time.

For the point-location problem, we use a preprocessing scheme similar to the one by Dobkin and Lipton [15]. The following result is a straightforward consequence of their paper

LEMMA 5. *Given a set of m line segments in a plane, it is possible to preprocess them in $O(\log m)$ time using $O(m^3)$ processors such that point location for any query point can be done in $O(\log m)$ time. The space needed is $O(m^3)$.*

PROOF. We merely mimic the sequential algorithm. After computing all possible pairs of intersections in $O(1)$ time using $O(m^2)$ processors, we project the intesection on the x-axis and for each interval we compute the total ordering of the lines in $O(\log m)$ time using m processors for each of the intervals. Thus the space required is $O(m^3)$. □

Because of this preprocessing there is an increase in the number of regions from what we mentioned in the previous paragraph. Each region is now a trapezoid (it could also be a triangle) and there are $O(n^6)$ regions. Accordingly we require $O(n^{13})$ processors to precompute the possible intersections. Since each region is a trapezoid, we choose a sample point in each trapezoid to do the precomputing part. We make a table corresponding to each pair of regions containing the trapezoids the corresponding segment intersects. For any input segment, we first

do the point location for its endpoints and then perform another search on this ordered pair of regions. Clearly, the whole procedure is a constant number of binary searches and takes $O(\log n)$ time. We can also store the number of intersecting trapezoids for each entry, so that we can allocate the required number of processors corresponding to each segment to list the intersecting trapezoids (using a prefix sum).

Thus we can state the main result of this section as follows:

THEOREM 2. *Algorithm Nested-Plane-Sweep Tree runs in* $O(\log n)$ *time and* $O(n \log \log n)$ *space using* $O(n)$ *processors in a CREW PRAM model.*

PROOF. Since steps (1), (2), and (3) run in $O(\log n)$ time, we have the following recurrence equation for $\bar{T}(n)$, the expected time complexity of the algorithm:

$$\bar{T}(n) = O(\log n) + (1 - n^{-\alpha})\bar{T}(n^{\varepsilon} \log n) + n^{-\alpha}\bar{T}(n - n^{\varepsilon}).$$

The solution of this recurrence equation gives $\bar{T}(n) = O(\log n)$ + low-order terms. From our previous discussion $\varepsilon < \frac{1}{13}$.

At this point we have shown that with a sufficient number of processors, the algorithm executes in expected $O(\log n)$ time. Since the number of (broken) segments can increase by a constant factor ($k_{\max}$), this could increase the input size by a polylogarithmic factor over $O(\log \log n)$ levels. Consequently, the processor bounds would have to increase by the same factor (from being linear) to achieve the time bound of $O(\log n)$. To avoid this, we make the following modification in the algorithm. After partitioning the segments into the trapezoidal regions (using the procedure in Section 3.4), we group the segments into two categories:

(a) Segments (part-segments) that have at least one endpoint in the region.
(b) Segments that span the region (horizontally).

Notice that the number of segments of type (a) is less than $2n$ and the segments of type (b) in a region i, S^i can be completely ordered (with respect to the y-coordinate). While performing *multilocation*, a straightforward binary search suffices for ordered segments in S_i. Consequently, we do not further preprocess the segments in S_i, i.e., we can leave them from further recursive calls. Thus, the total size of the subproblems at any level of the recursive call is no more than $2n$. Processor allocation is achieved by simply setting processors in a region equal to the number of endpoints lying in it. This shows that the algorithm does not need more than $O(n)$ processors.

To analyze the cumulative effects of the nested calls on the worst-case total runtime of the algorithm, we need to consider only one such sequence of nested calls. This gives the probability that a leaf-node process fails to complete within $O(\log n)$ time. Since the success of the algorithm depends on completion of all such processes (at the leaf level), the probability that the algorithm fails to halt in $O(\log n)$ time is less than $n \cdot \text{Prob}[\text{one such process fails}]$. Note that this part of the analysis is very similar to *Flashsort*.

From Lemma 4 we know that, for any given level i, the time T_i can be bounded by

$$P[T_i \geq c_0 \log n(\varepsilon_0)^i] \leq 2^{-(\varepsilon_0)^i \log n \alpha c_0},$$

where $\varepsilon_0 < \frac{12}{13}$, since we choose only $n^{1/13}$ sample segments. Setting $t_i = (\varepsilon_0)^i \log n(c - c_0)$, we obtain

$$P[T_i \geq c \log n(\varepsilon_0)^i + t_i] \leq 2^{-(\varepsilon_0)^i c \log n} < 2^{-t_i}.$$

If T is the total time for this worst-case chain of nested calls and $k = 1/1 - \varepsilon_0$, the probability that it takes more than $1/1 - \varepsilon_0 \log nc_0 + t$, where $t = 1/1 - \varepsilon_0(c - c_0) \log n$, is less than

$$\prod_{\Sigma t_i = t} 2^{-t_i} \leq \sum 2^{-t}$$

over $(\log n)^{O(\log \log n)} \cdot \log \log n$ tuples. Thus

$$P[T > k \log nc_0 + t] < 2^{-t + O(\log \log^2 n) + \log \log \log n}.$$

Using $t = k(c - c_0) \log n$, for large values of n, we can rewrite the above expression as

$$P[T > k \log n] < 2^{(c - c_0) \log n}.$$

For $c > 4c_0$, i.e., $c - c_0 > \frac{3}{4}c$, we have the following required bound:

$$P[T > k \log n] \leq 2^{-(3/4)c \log n} = n^{-\beta c}. \qquad \square$$

The worst-case processor bounds follow from the previous argument, which remains identical.

4. Applications of Nested-Plane-Sweep Tree. In this section we exploit the data structure that we constructed in the previous section to improve the running time of various algorithms like triangulation, intersection detection, three-dimensional maxima, two-dimensional dominance counting, and visibility from a point.

LEMMA 6. *The nested-plane-sweep tree constructed in the previous section can be used to multilocate any query point p in* $\tilde{O}(\log n)$ *sequential time.*

PROOF. From Fact 1 multilocation in an $O(n_i)$ node plane-sweep tree can be performed in $O(\log n_i)$ serial time. Thus, the expected time for multilocation in the nested-plane-sweep tree obeys the following recurrence relation:

$$\bar{T}(n) = O(\log n) + (1 - n^{-\beta})\bar{T}(\sqrt{n} \log n) + (n^{-\beta})\bar{T}(n)$$

giving $\bar{T}(n) = O(\log n)$ which is the multilocation time for each query point p using one processor. Equivalently, multilocation for $O(n)$ query points can be performed in $O(\log n)$ expected time using $O(n)$ processors. The worst-case time bounds can be derived using techniques similar to the proof of Theorem 2. □

4.1. *Trapezoidal Decomposition and Triangulation*

DEFINITION (Trapezoidal Decomposition). Let $P = \{v_1, v_2, \ldots, v_n\}$ be a simple polygon, where v_i's denote the vertices of P, and are listed so that the interior of P is to the left of the walk $v_1 v_2 \cdots v_n$. For any vertex v_i of P a trapezoidal edge for v_i is an edge of P, which is directly above or below v_i, such that the vertical line segment from v_i to this edge is interior to P. A vertex can have none, one, or two such edges. Trapezoidal decomposition of a polygon P is to find the trapezoidal edges for each vertex of P.

LEMMA 7. *Given a simple polygon P, a trapezoidal decomposition of P can be constructed in $\tilde{O}(\log n)$ time using $O(n)$ processors and $O(n \log n)$ space in a CREW PRAM model.*

PROOF. The trapezoidal decomposition can be done in two distinct stages (see [6] for details). In the preprocessing part, a nested-plane-sweep tree is constructed on the edges of the simple polygon P. From Theorem 2 this runs in $O(\log n)$ time with very high probability. Subsequently, all the vertices of the polygon are multilocated simultaneously using $O(n)$ processors which takes $O(\log n)$ time with very high likelihood (from Lemma 6). For each point, it takes a constant time to determine if the vertical line intersecting the trapezoidal edge is within the polygon P using one processor per point. □

FACT 3 [6]. Given a one-sided monotone polygon with n vertices, P can be triangulated in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in a CREW PRAM model.

Our next theorem is a direct consequence of this fact and Lemma 7.

THEOREM 3. *Given a simple polygon with n vertices, P can be triangulated in $\tilde{O}(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in a CREW PRAM model.*

PROOF. The algorithm consists of three phases:

(1) We first build the nested-plane-sweep tree on the edges of the polygon. With very high probability this can be done in $O(\log n)$ time and $O(n \log n)$ space (from Theorem 2).
(2) The simple polygon is decomposed into one-sided monotone polygons by using trapezoidal decomposition (see [6] for details). Thus, from Lemma 7 the expected running time for this phase is $O(\log n)$.
(3) Each of the one-sided monotone polygons can be triangulated in $O(\log n)$ time and $O(n)$ space from Fact 3.

The proof of the theorem follows directly from the above steps. □

4.2. *Visibility from a Point*

DEFINITION (Visibility from a Point). Given a set of line segments $S = \{s_1, s_2, \ldots, s_n\}$, which do not intersect, except possibly at endpoints, and a point p, determine the part of the plane which is visible from p when every segment s_i is opaque.

We assume the point p to be at negative infinity below all segments. The algorithm that we present below can be appropriately modified for any general point. Notice that now (with the assumption) the problem is to determine the projections of segments on the x-axis, such that in case of overlap the segment closer to the x-axis has precedence (see Figure 4).

The solution to this problem consists of a sequence of points $\{p_1, p_2, \ldots, p_{2n}\}$ such that two consecutive points p_i, p_{i+1} define a segment visible in the interval $[x(p_i), x(p_{i+1})]$. An important property of this problem is that the visibility is a continuous function between the endpoints of the segments, i.e., for all points between two endpoints (projection of the endpoints on the x-axis), the same segment will be visible. We can refer to the endpoints as *critical* points where the visibility function defined as Vis(x) may change its value. This property enables us to solve the problem efficiently. Since Vis(x) is constant between any two consecutive endpoints, we choose a point p which lies in this interval below all the segments and draw a vertical line through p. The segment which intersects the vertical line first (from below) is the visible segment in the corresponding interval. Formally, the algorithm can be described as follows:

Algorithm Visibility

(1) Given a set S of nonintersecting segments, sort their endpoints on the x-coordinate. This can be done in $O(\log n)$ time using Cole's [12] parallel

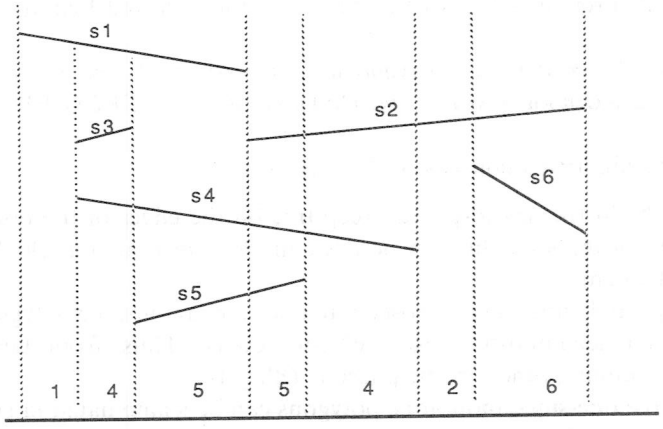

Fig. 4. The intervals on the horizontal segment are labeled by the segment visible in that interval.

mergesort algorithm which does not have a large constant as does the AKS algorithm.
(2) For each of the $2n-1$ bounded intervals, choose a point within the interval. A possible choice is the midpoint of an interval. Let us denote them by $\{p_1, p_2, \cdots, p_{2n-1}\}$.
(3) Construct a nested-plane-sweep tree on the segments of set S.
(4) Multilocate the $2n-1$ midpoints on this tree giving us ordered pairs of the form (p_i, s_j) which carries the required visibility information for the interval $[x_i, x_{j+1}]$.

The following theorem is a direct consequence of our previous discussion:

THEOREM 4. *Given a set S of segments and a point p, the visibility of the plane induced by this set of segments from this point can be computed in $\tilde{O}(\log n)$ time using $O(n)$ processors in a CREW PRAM.*

5. Dominance Problems. The problems discussed in the previous section used trapezoidal decomposition which can be efficiently carried out with the nested-plane-sweep tree. A plane-sweep tree was constructed on a random sample, and was used to divide and conquer on the plane which was divided into trapezoidal regions. An important property which was not mentioned explicitly was that by dividing the plane in such a manner we did not require a merging phase. For trapezoidal decomposition, locating a point within a particular trapezium at any phase eliminated interaction with the other regions since the trapezoidal edge could not be exterior to this region. The problems dealt with in this section are of a somewhat different nature, and there may arise a need to determine the relative location of a query point with respect to more than one trapezoidal regions. This distinction can be brought out more clearly by the following example: Given a set of nonintersecting segments and a query point, p, we wish to determine the following:

(1) Which segment is immediately above p?
(2) Among all the segments above p which one is the longest?

The first problem can be solved by trapezoidal decomposition which can be done efficiently using the nested-plane-sweep tree. However, the second problem is more difficult because it cannot be solved by simply finding the segment immediately above the point p within one region. It will be shown that the latter problem can be solved easily using the normal plane-sweep tree but the construction of this tree is the bottleneck, i.e., it takes $O(\log n \log\log n)$ time using $O(n)$ processors. To avoid building the full data structure we argue that we do not need all the attributes (mentioned in Section 3.1) in the plane-sweep tree and also avoid the need for fractional cascading in the algorithms presented in this section.

5.1. Maximal Set in Three-Dimensions. The *Maxima* problem is defined as follows: given a set S of points in d-dimensions, the maxima of S is the set of all

points p_i, such that $p_i \in S$, and is not dominated on all dimensions by any other point in set S.

Efficient sequential algorithms are known for only two and three dimensions (the $O(n \log n)$-time algorithm is optimal). For two dimensions, an $O(\log n)$ algorithm using $O(n)$ processors is easily obtainable by using the $O(\log n)$ AKS sorting network or Cole's [12] parallel mergesort (which has a much lower constant). The best known result for three-dimensional maxima is $O(\log n \log \log n)$ from Atallah and Goodrich's algorithm [6] in which they use parallel merging techniques. We present a method to reduce the three-dimensional maxima problem to integer sorting so that we can solve it in $O(\log n)$ time using $O(n)$ processors.

Given a set S of points in three dimensions, consider their projection on the x–y plane. For any point (x_i, y_i) draw the segment s_i parallel to x-axis joining $(0, y_i)$ and (x_i, y_i). Note that any point (x_j, y_j) is dominated by another point (x_i, y_i) iff the point lies below s_i and also $z_j \leq z_i$. This means that the point is dominated by p_i on all the axes. Instead of comparing the z-coordinates of all s_i which are above p_j, it is enough to compare z_j with the maximum of the z-coordinate for all the segments s_i which dominate p_j (Figure 5). Note that identifying the segments which are above p_j is similar to the multilocation of p_j. However, this time we are not only interested in the segment which lie directly above p_j but also which of them has the maximum z-coordinate and this is unique for any interval. It is possible to precompute the maximum z-coordinate for each interval so that the necessary comparisons can be performed in constant additional time during each step of multilocation. This preprocessing requires a parallel-prefix type of computation. The following well-known result can be used to do this efficiently:

FACT 4. Parallel-prefix computation for an n element sequence can be done in $O(\log n)$ time using $O(n/\log n)$ processors (for example, see [24]).

The sequence of elements in each node are segments "covering" the correspond-

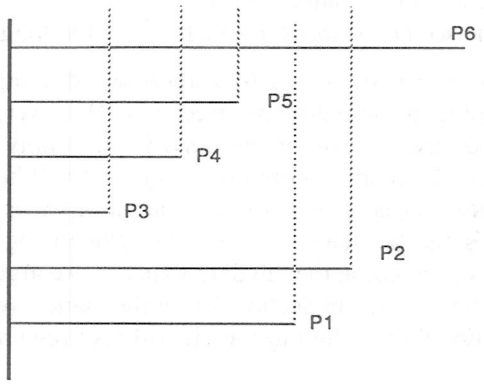

Fig. 5. Point $p1$ is not a maxima iff it is dominated by $p2$ and $p6$ in the z-coordinate. In other words $z(p1) < \max\{z(p2), z(p3)\}$.

ing interval of the plane-sweep tree sorted on the y-axes and the operation needed is MAX. (Recall the definition of "covering" from Section 3.1.)

However, as mentioned previously, we cannot afford to construct the full data structure of [6] and therefore the following observations can be made:

OBSERVATION 1. We do not require fractional cascading to perform multilocation of the input points. We can instead use their ranks in the list $H(v)$. Note that fractional cascading gives us more information for multilocating those points that are not in the input set (used for building the tree).

OBSERVATION 2. We can build the sets $H(v)$ in $O(\log n)$ time using the following result:

FACT 5 [23]. Integer sort of n keys in the range $[1, n^{O(1)}]$ can be performed in time $O(\log n)$ using $n/\log n$ processors in a CREW PRAM model given word size of n^{ε} bits for any $\varepsilon > 0$.

We first build the skeleton of the plane-sweep tree on the intervals induced by the endpoints of the segments (corresponding to the input points). Note that the segments can be totally ordered on the y-coordinates and, after an initial sorting on the y-coordinate, we can made use of their ranks (which is an integer in the interval $[1, n]$) instead of the actual y-coordinate. Using one processor per segment and concurrent read, we can find the allocation nodes for each segment—there can be at most $2 \cdot \log n$ for each. Due to the special nature of the line segments (all have the same left x-coordinate), it can be seen that the allocation node cannot be a right child. We also make a slight modification—we not only allocate segments to the appropriate intervals but also to all the left children in the *allocation path* (see Figure 6 for an illustration of such nodes). These are special allocation nodes of a segment so we distinguish them by some special markings. Note that there can be at most $\log n$ such special allocation nodes for each segment. We can next build the unordered lists for $H(v)$ ($H(v)$ is ordered on the y-axis) for each node v of the plane-sweep tree using lexicographic sorting and prefix computation on the ordered pairs (v, s_i). Though there can be $O(n \log n)$ such pairs, this can still be done in $O(\log n)$ time using n processors using Facts 4 and 5. Another application of sorting (on the ranks of the y-coordinate) and prefix computation gives us the $H(v)$ in the same time bounds.

The full algorithm is presented below:

Procedure 3-D Maxima

Input: n points in three dimensions, where each point p_i is defined by its three coordinates (x_i, y_i, z_i).

(1) Build a plane-sweep tree on the segments joining (x_i, y_i) and $(0, y_i)$, and compute the $H(v)$'s using the scheme described in Observation 2. At the end of this step, we have, for each segment, its rank in all the $H(v)$'s (at most $2 \log n$) which will be used for multilocation.

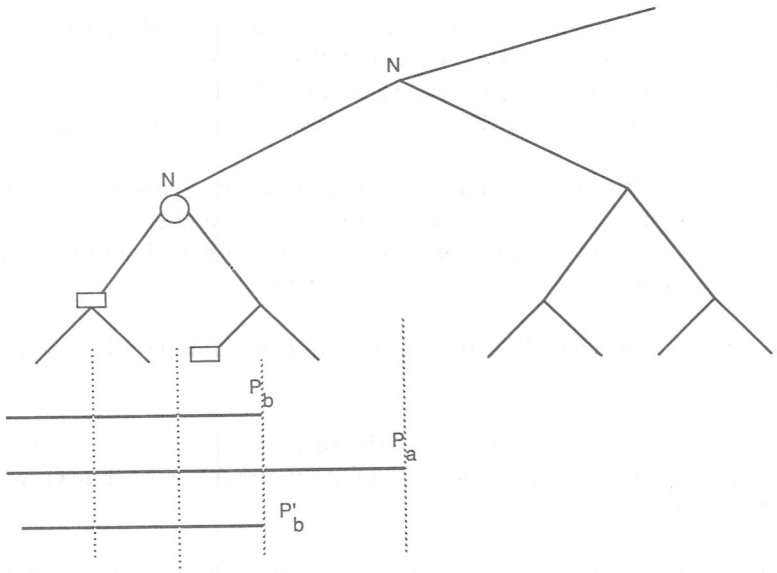

Fig. 6. p_a is allocated to the circled node and the square nodes store p_b and p'_b. Nodes marked N are the special nodes of allocation for p_b which are the left children on the allocation path for p_b.

(2) For each node n_i, calculate $\text{Max}(s_{k,n_i}) = \max_{z\text{-coord}}\{s_{1,n_i}s_{2,n_i}\cdots s_{kn_i}\}$ for $k \leq |H(n_i)|$, where s_{1,n_i}, $s_{2,n_i}, \ldots, s_{|H(n_i)|,n_i}$ are segments in node n_i ordered on the y-coordinate (which is the set $H(n_i)$ from its definition). During this step, the marked entries (which are to be used for multilocation) are assigned y-coordinates of $-\infty$ so that they do not affect the computation.

(3) At each node, in the path followed by a point p_i from the root to the rightmost allocation node of s_i, compare $z(p_i)$ with the $\text{Max}(s_{k,n_i})$, where p_i lies immediately below s_{k,n_i} using the rank of s_i in the corresponding $H(v)$. If at any time during the search $z(p_i)$ is less than this value, p_i is not a maxima of S. (Note that this step can be done in $O(\log n)$ time by using one processor per point.)

THEOREM 5. *Algorithm 3-D Maxima finds the maximal points correctly in* $O(\log n)$ *time and* $O(n \log n)$ *space using* $O(n)$ *processors in a CREW PRAM model.*

PROOF. The proof of the time complexity is based on the previous discussion. To see that it identifies the maximal points correctly, we have to explain the significance of the "marked" nodes in more detail. Given points p_a and p_b, where p_a dominates p_b, we have to ensure that they are allocated to at least one common node, so that in step (3), we are able to eliminate p_b from the set of maximal elements. This may not happen in the normal allocation scheme (as shown in Figure 6), hence there is a need for special nodes. The assignment of $-\infty$ to the special nodes in step (2), prevents p_b from dominating p_a if $y(p_b) > y(p_a)$ and $z(p_b) > z(p_a)$, but $x(p_b) < x(p_a)$. □

5.2. Two-Set Dominance on a Plane. The two-set dominance counting problem is defined as follows: 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, we wish to determine the number of points in V which are dominated (on two coordinates) by each point q_i in U. This seems to be closely related to the two-dimensional dominance problem but is harder since not only do we have to identify points that are not maximum but also find *all* points which dominate a given point. However, this problem can also be solved using a scheme similar to the previous problem.

Given the set U and V, we transform each point $q_i \in U$, with coordinates (x_i, y_i), into a segment s_i joining (x_i, y_i) and $(0, y_i)$ (similar to the previous problem) and build a plane-sweep tree on these segments. Then, in parallel, allocate each point p_i of set V on this tree to the appropriate intervals (all the nodes in the path which are left children in the tree). These are marked as special elements to be used for dominance counting. Next, we construct the sets $H(v)$'s using the original segments as well as the points of V. We mark the elements corresponding to p_i and q_i with 1 and 0, respectively. In each node, the number of special points in $H(v)$ (the points of V) lying under a segment s_i is the number of points dominated by s_i in that interval. Thus, we can carry out a prefix sum computation using the reverse order of the $H(v)$'s. To find the total number of points dominated by q_i, we simply add sequentially the numbers over all the intervals (at most $\log n$) in which the segment is allocated. A formal description of the algorithm is given below.

Two-Set Dominance Counting

(1) Build a plane-sweep tree on the segments joining (x_i, y_i) and $(0, y_i)$ where (x_i, y_i) correspond to coordinates of a point q_i in set U.
(2) Locate each point p_i of set V simultaneously on the plane-sweep tree constructed in step (1) and allocate it to all the nodes (intervals) which are the left children in the path taken by the point in the tree. (Note that p_i is not stored as a segment but just as a point.)
(3) Mark the points of V and construct $H(v)$ by lexicographic sorting on the pairs $(v, y(p_i))$ using the integer sorting algorithm in [23].
(4) For each node v of the tree perform a parallel prefix sum on the set $H(v)$ to count the number of points of V lying below a q_i. This gives us the total number of points dominated by a segment (and hence the corresponding point) in a certain interval.
(5) For any segment find the total number of points dominated by all the intervals of the segment. This can also be done using one processor per q_i, since there can be at most $\log n$ intervals for each segment s_i.

THEOREM 6. *Algorithm Two-Set Dominance Counting runs correctly in $O(\log n)$ time using $O(n)$ processors in a CREW model where $n = |U| + |V|$.*

PROOF. The proof of correctness of the algorithm is similar to three-dimensional dominance where we use some special allocation nodes. To ensure a

correct count of the number of points in set V dominated by q_k, we have to count each point p_i (dominated by q_k) exactly once. Thus q_k and p_i should share exactly one node of allocation. The special allocation nodes guarantee that there will be at least one shared node. The argument that there is at most one such node follows from the way that p_i and q_i are stored in the tree. All the nodes where p_i's are allocated are in a path (and hence are related as ancestors), whereas the q_i are stored (as transformed segments) in nodes which cannot be ancestors of each other. The justification for time complexity follows easily from the preceding discussion since each of the steps can be completed in $O(\log n)$ time. □

Two-set dominance counting can be used to solve a number of problems efficiently, among which the multiple range counting problem is very important. The multiple range counting problem can be described as following: Given a set V of l points and a set R of m rectangles (ranges), the multiple range counting problem is to compute the number of points interior to each rectangle. Note that this is equivalent to a data-base query where the ranges are defined by two different keys. The next result follows immediately.

COROLLARY 3. *Given a set V of l points in a plane, and a set R of m isothetic rectangles, we can solve the multiple range counting problem for V and R in $O(\log n)$ time with very high probability using $O(n)$ processors where $n = l + m$.*

PROOF. We can transform this problem into the two-dimensional dominance counting problem as follows: Given a rectangle r defined by its four corners (p_1, p_2, p_3, p_4) the number of points enclosed by a rectangle is equal to $d(p_1) - d(p_2) - d(p_3) + d(p_4)$. Here p_1 is the upper right corner of the rectangle, p_4 is the lower right corner, and $d(v)$ denotes the number of points in V two-dominated (dominated on both coordinates) by v. The result follows. □

6. Concluding Remarks. In this paper we have designed parallel algorithms for solving a variety of problems in computational geometry which have optimal running time with high probability. Some of these algorithms use a new data structure, which we call the nested-plane-sweep tree. This appears to be a useful modification of the plane-sweep tree described in [1] and [6]. On the other hand, the plane-sweep tree itself can be used to solve a number of problems optimally if it can be constructed optimally. Random splitting was used very effectively to divide and conquer on the plane, raising hopes about extending these techniques efficiently to problems with higher dimensions like the three-dimensional convex hulls. A challenging exercise will be mapping these algorithms to fixed interconnection networks like hypercube and maintain the same performance.

As mentioned in the introduction, Atallah *et al.* [3] have independently discovered optimal deterministic algorithms for all these problems. Some of their algorithms use the less powerful EREW model and their algorithms and they do not use n^ε size words that we need for integer sorting. Since they are based on an approach used by Cole for the optimal parallel mergesort algorithm (see [12]), there is little hope of extending them to the fixed interconnection models while maintaining the optimal performance achieved in the PRAM models.

Moreover, as demonstrated by Clarkson [11], randomized methods have very wide applications in computational geometry in terms of simplifying deterministic algorithms or speeding up suboptimal algorithms. Though his methods were restricted to a sequential setting, this paper provides ample evidence that they can be extended to the parallel setting with some additional techniques (as seen in Section 3.4). In addition, the algorithms can be made to run in a certain time (and number of processors) with very high likelihood instead of just a constant probability. There are still a number of important problems in computational geometry which do not have optimal parallel algorithms and we feel that this paper has merely scratched the surface.

Acknowledgments. We wish to thank Ken Clarkson for pointing out a serious error in the proof of Lemma 4 in an earlier version of the manuscript. More specifically, we had claimed high probability bounds using a straightforward sampling procedure which was refuted by a counterexample due to Peter Shor. We are also grateful to the referees for their comments for improving presentation and especially for noticing that the assumption regarding binomial distribution in the proof of Lemma 1 was incorrect in the way it was being used. We are also grateful to Sanguthevar Rajasekaran for several insightful discussions on the probabilistic arguments and to Salman Azhar for carefully reading an earlier version.

Appendix. We say a random variable X upper bounds another random variable Y (equivalently Y lower bounds X) if, for all x such that $0 \leq x \leq 1$, $\text{Prob}(X \leq x) \leq \text{Prob}(Y \leq x)$.

A Bernoulli trial is an experiment with two possible outcomes, namely success and failute. The probability of success is p.

A binomial variable X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *probability mass function* of X can be easily seen to be

$$\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^n (1-p)^{n-k}.$$

The tail end of the Binomial distribution can be bounded by *Chernoff* bounds. In particular the following approximations are frequently used:

(1) $$\text{Prob}(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np},$$

(2) $$\text{Prob}(X \leq m) \leq \left(\frac{np}{m}\right)^m e^{-np+m},$$

(3) $$\text{Prob}(X \leq (1-\varepsilon)pn) \leq \exp\left(\frac{-\varepsilon^2 np}{2}\right),$$

(4) $$\text{Prob}(X \geq (1+\varepsilon)np) \leq \exp\left(\frac{-\varepsilon^2 np}{3}\right)$$

for all $0 < \varepsilon < 1$. The last two bounds actually follow from the Chernoff bounds which (for a discrete distribution) is given as

$$\text{Prob}[A \geq x] \leq z^{-x} G_A(z),$$

where $G_A(z)$ is the probability generating function. To minimize the bound we substitute $z = z_0$ which minimizes the right-hand side expression.

The probability mass function of a modified geometric random variable X is specified by $p_X(i) = p(1-p)^i$ for $i = 0, 1, 2, \ldots$. The probability generating function of a modified geometric random variable X is given by

$$G_Y(z) = \frac{p}{1-(1-p)z},$$

where p is the probability of success in each individual trial. Recall that Y is the number of trials before we have a success. Thus the generating function of sum of $\log n$ independent identical modified geometric random variable is given by

$$G_Y(z) = \left[\frac{p}{1-(1-p)z}\right]^{\log n}.$$

Using Chernoff bounds and minimizing the bounds by differentiating we obtain

$$\text{Prob}[Y \geq c \log n] \leq n^{-c \log_2(2c/(c+1))} \text{ for } p = \tfrac{1}{2}.$$

For $c > 2$, we can write it as

$$\text{Prob}[Y \geq c \log n] \leq n^{-\alpha c} \quad \text{where} \quad \alpha > 0. \tag{5}$$

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, Parallel computational geometry, *Proc. of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 468–477.

[2] M. Ajitai, J. Komlos, and E. Szemeredi, Sorting in c log n parallel steps, *Combinatorica*, **3**, 1983, 1–19.

[3] M. J. Atallah, R. Cole, and M. T. Goodrich, Cascading divide-and-conquer: a technique for designing parallel algorithms, *Proc. of the 28th Annual IEEE Symposium on the Foundations of Computer Science*, 1987, pp. 151–160.

[4] M. J. Atallah and M. T. Goodrich, Efficient parallel solutions to some geometric problems, *Proc. of the 1985 IEEE International Conference on Parallel Processing*, pp. 411–417.

[5] M. J. Atallah and M. T. Goodrich, Parallel algorithms for some functions of two convex polygons, *Proc. of the 24th Allerton Conference on Communications, Control, and Computing*, 1986.

[6] M. J. Atallah and M. T. Goodrich, Efficient plane sweeping in parallel, *Proc. of the 2nd ACM Symposium on Computational Geometry*, 1986, pp. 216–225.

[7] A. Borodin and J. Hopcroft, Routing, merging and sorting on parallel models of computation, *J. Comput. System Sci.*, **30**(1), 1985, 130–145.

[8] H. Chernoff, A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations, *Ann. of Math. Statist.*, **23**, 1952, 493–507.
[9] K. L. Clarkson, A probabilistic algorithm for the post-office problem, *Proc. of the 17th Annual SIGACT Symposium*, 1985, pp. 174–184.
[10] K. L. Clarkson, New applications of random sampling in computational geometry, *Discrete Comput. Geom.*, **2**, 1987, 195–222.
[11] K. L. Clarkson, Applications of random sampling in computational geometry, II, *Proc. of the 4th Annual ACM Symposium on Computational Geometry*, 1988, pp. 1–11.
[12] R. Cole, Parallel merge sort, *Proc. of the 27th Annual IEEE Symposium on Foundations of Computer Science*, 1986, pp. 511–516.
[13] R. Cole and U. Vishkin, Approximate and exact parallel scheduling with applications to list, tree and graph problems, *Proc. of the 27th Annual IEEE Symposium on Foundations of Computer Science*, 1986, pp. 478–491.
[14] N. Dadoun and D. G. Kirkpatric, Parallel processing for efficient subdivision search, *Proc. of the 3rd Annual Symposium on Computational Geometry*, 1987, pp. 205–214.
[15] D. Dobkin and R. J. Lipton, Multidimensional searching problems, *SIAM J. Comput.*, **5**(2), 1976, 181–186.
[16] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, Optimal point location in a monotone subdivision, *SIAM J. Comput.*, **15**(2), 1986.
[17] H. Gazit, An optimal randomized parallel algorithm for finding connected components in a graph, *Proc. of the 27th Annual IEEE Symposium on Foundations of Computer Science*, 1986, pp. 492–501.
[18] D. Haussler and E. Welzl, ε-nets and simplex range queries, *Discrete Comput. Geom.*, **2**(2), 1987, 127–152.
[19] D. G. Kirkpatrick, Optimal search in planar subdivisions, *SIAM J. Comput.*, **12**, 1979, 18–27.
[20] N. Meggido, Parallel Algorithms for Finding the Maximum and the Median Almost Surely Constant Time, Preliminary Report, Computer Science Dept., C.M.U., Pittsburg, Oct. 1982.
[21] G. L. Miller and H. Reif, Parallel tree contraction and its application, *Proc. of the 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 478–489.
[22] F. P. Preparata and I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
[23] S. Rajasekaran and S. Sen, On parallel integer sorting, Technical Report CS-1987-38, Dept. of Computer Science, Duke University, 1987.
[24] J. Reif, An optimal parallel algorithm for integer sorting, *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 496–503.
[25] J. H. Reif and L. G. Valiant, A logarithmic time sort for linear size networks, *J. Assoc. Comput. Mach.*, **34**(1), 1987, 60–76.
[26] R. Reischuk, A fast probabilistic parallel sorting algorithm, *Proc. of the 22nd Annual IEEE Symposium on Foundations of Computer Science*, 1981, pp. 212–219.
[27] L. G. Valiant, Parallelism in comparison problems, *SIAM J. Comput.*, **4**, 1975, 348–355.
[28] L. G. Valiant, A scheme for fast parallel communication, *SIAM J. Comput.*, **11**(2), 1982, 350–361.