DETERMINISTIC PARALLEL COMPUTATIONAL GEOMETRY

by Mikhail Atallah, Danny Chen

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Deterministic Parallel Computational Geometry

Mikhail J. Atallah[*] Danny Z. Chen[†]

Abstract

We describe general methods for designing deterministic parallel algorithms in computational geometry. We focus on techniques for shared-memory parallel machines, which we describe and illustrate with examples. We also discuss some open problems in this area.

[*]Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA. E-mail: mja@cs.purdue.edu. Portions of the work of this author were supported by the National Science Foundation under Grant CCR-9202807, and by sponsors of the COAST Laboratory.

[†]Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA. E-mail: chen@cse.nd.edu. The work of this author was supported in part by the National Science Foundation under Grant CCR-9623585.

1 Introduction

Many problems in computational geometry come from application areas (such as pattern recognition, computer graphics, operations research, computer-aided design, robotics, etc) that require real-time speeds. This need for speed makes parallelism a natural candidate for helping achieve the desired performance. For many of these problems, we are already at the limits of what can be achieved through sequential computation. The traditional sequential methods can be inadequate for those applications in which speed is important and that involve a large number of geometric objects. Thus, it is important to study what kinds of speed-ups can be achieved through parallel computing. As an indication of the importance of this research direction, we note that four of the eleven problems used as benchmark problems to evaluate parallel architectures for the DARPA Architecture Workshop Benchmark Study of 1986 were computational geometry problems.

In parallel computation, it is the rule rather than the exception that the known sequential techniques do not translate well into a parallel setting; this is also the case in parallel computational geometry. The difficulty is usually that these techniques use methods which either seem to be inherently sequential, or would result in inefficient parallel implementations. Thus new paradigms are needed for parallel computational geometry. The goal of this chapter is to give a detailed look at the currently most successful techniques in parallel computational geometry, while simultaneously highlighting some open problems, and discussing possible extensions to these techniques. It differs from [33] in that it gives a much more detailed coverage of the shared-memory model at the expense of the coverage of networks of processors. Since it is impossible to describe all the parallel geometric algorithms known, our focus is on general algorithmic techniques rather than on specific problems; no attempt is made to list exhaustively all of the known deterministic parallel complexity bounds for geometric problems. For more discussion of parallel geometric algorithms, the reader is referred to [15, 139].

The rest of the chapter is organized as follows. Section 2 briefly reviews the PRAM parallel model and the notion of efficiency in that model, Section 3 reviews basic subproblems that tend to arise in the solutions of geometric problems on the PRAM, Section 4 is about inherently sequential (i.e. non-parallelizable) geometric problems, Section 5 discusses the parallel divide and conquer techniques, Section 6 discusses the cascading technique, Section 7 discusses parallel fractional cascading, Section 8 discusses cascading with labeling func-

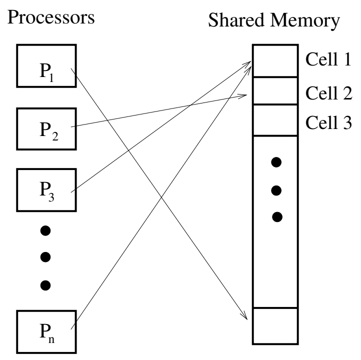

Figure 1: The PRAM model.

tions, Section 9 discusses cascading in the EREW model, Section 10 discusses parallel matrix searching techniques, Section 11 discusses a number of other useful PRAM techniques, and Section 12 concludes.

2 PRAM Models

The PRAM (Parallel Random Access Machine) has so far been the main vehicle used to study the parallel algorithmics of geometric problems, and hence it is the focus of this chapter. This section briefly reviews the PRAM model and its variants.

The PRAM model of parallel computation is the shared-memory model in which the processors operate synchronously [153], as illustrated in Figure 1. A *step* on a PRAM consists of each processor reading the content of a cell in the shared memory, writing data in a cell of the shared memory, or performing a computation within its own registers. Thus all communication is done via the shared memory. There are many variants of the PRAM, differing from one another in the way read and/or write conflicts are treated. The CREW (Concurrent Read Exclusive Write) version of this model allows many processors to simultaneously read the content of a memory location, but forbids any two processors from simultaneously attempting to write in the same memory location (even if they are trying to write the same thing). The CRCW (Concurrent Read Concurrent Write) version of the PRAM differs from the CREW one in that it also allows many processors to write simultaneously in the same memory location: In any such common-write contest, only one processor succeeds, but it is not known in advance which one. (There are other versions of the CRCW PRAM but we shall not concern ourselves with these here.) The EREW PRAM is the weakest version of the PRAM: It forbids both concurrent reading and concurrent writing.

The PRAM has been extensively used in theoretical studies as a vehicle for designing parallel algorithms. Although it captures important parameters of a parallel computation, the PRAM does not account for communication and synchronization (more recent variations of the model do account for these factors, but we do not discuss them since essentially no parallel geometric algorithms have yet been designed for them). The PRAM is generally considered to be a rather unrealistic model of parallel computation. However, although there are no PRAMs commercially available, algorithms designed for PRAMs can often be efficiently simulated on some of the more realistic parallel models. The PRAM enables the algorithm designer to focus on the structure of the problem itself, without being distracted by architecture-specific issues. Another advantage of the PRAM is that, if one can give strong evidence (in the sense explained in the next paragraph) that a problem has no fast parallel solution on the PRAM, then there is no point in looking for a fast solution to it on more realistic parallel models (since these are weaker than the PRAM).

We now review some basic notions concerning the speed and efficiency of PRAM computations. The *time* $\times$ *processor* product of a PRAM algorithm is called its *work* (i.e., the total number of operations performed by that algorithm). A parallel algorithm is said to run in *polylogarithmic* time if its time complexity is $O(\log^k n)$, where n is the problem size and k is a constant independent of n (i.e., $k = O(1)$). A problem solvable in polylogarithmic time using a polynomial number of processors is said to be in the class NC. It is strongly believed (but not proved) that not all problems solvable in polynomial time sequentially are solvable in polylogarithmic time using a polynomial number of processors (i.e., it is believed that P $\neq$ NC). As in the theory of NP-completeness, there is an analogous theory in parallel computation for showing that a particular problem is probably not in NC: By showing that the membership of that problem in NC would imply that P $=$ NC. Such a proof consists of showing that each problem in P admits an NC reduction to the problem at hand (an NC reduction is a reduction that takes polylogarithmic time and uses a polynomial number of processors). Such a problem is said to be *P-complete.* For a more detailed discussion of the class NC and parallel complexity theory, see (for example) [189] or [156]. A proof establishing P-completeness of a problem is viewed as strong evidence that the problem is "inherently sequential".

Once one has established that a geometric problem is in NC, the next step is to design a

PRAM algorithm for it that runs as fast as possible, while being efficient in the sense that it uses as few processors as possible. Ideally, the parallel time complexity of the PRAM algorithm should match the parallel lower bound of the problem (assuming such a lower bound is known), and its work complexity should match the best known sequential time bound of the problem. A parallel lower bound for a geometric problem is usually established by showing that such an algorithm can be used to solve some other (perhaps non-geometric) problem having that lower bound. For example, it is well known [94] that computing the logical OR of n bits has an $\Omega(\log n)$ time lower bound on a CREW PRAM. This can easily be used to show that detecting whether the boundaries of two convex polygons intersect also has an $\Omega(\log n)$ time lower bound in that same model, by encoding the n bits whose OR we wish to compute in two concentric regular n-gons such that the ith bit governs the relative positions of the ith vertices of the two n-gons. Interestingly, if the word "boundaries" is removed from the previous sentence then the lower bound argument falls apart and it becomes possible to solve the problem in constant time on a CREW PRAM, even using a sublinear number of processors [43, 224].

Before reviewing the techniques that have resulted in many PRAM geometric algorithms that are fast and efficient in the above sense, a word of caution is in order. From a theoretical point of view, the class NC and the requirement that a "fast" parallel algorithm should run in polylogarithmic time, are eminently reasonable. But from a more practical point of view, not having a polylogarithmic time algorithm does not entirely doom a problem to being "non-parallelizable". One can indeed argue [221] that, e.g., a problem of sequential complexity $\Theta(n)$ that is solvable in $O(\sqrt{n})$ time by using $\sqrt{n}$ processors is "parallelizable" in a very real sense, even if no polylogarithmic time algorithm is known for it.

3 Basic Subproblems

This section reviews some basic subproblems that tend to be used as subroutines in the design of PRAM geometric algorithms.

3.1 Sorting and Merging

Sorting is probably the most frequently used subroutine in parallel geometric algorithms. Fortunately, for PRAM models we know how to sort n numbers optimally: $O(\log n)$ time and n processors on the EREW PRAM [87, 12]. Merging on the PRAM is easier than sorting [211, 222, 54, 147]; on the CREW PRAM it is $O(\log\log n)$ time with $n/\log\log n$ processors, and $O(\log n)$ time with $n/\log n$ processors on the EREW PRAM.

3.2 Parallel Prefix

Given an array A of n elements and an associative operation denoted by $+$, the parallel prefix problem is that of computing an array B of n elements such that $B(i) = \sum_{k=1}^{i} A(k)$. Parallel prefix can be performed in $O(\log n)$ time and $n/\log n$ processors on an EREW PRAM [159, 164], $O(\log n/\log\log n)$ time and $n\log\log n/\log n$ processors on a CRCW PRAM [92]. Computing the smallest element in array A is a special case of parallel prefix: In the CRCW model, this can be done faster than general parallel prefix—in $O(1)$ time with $n^{1+\epsilon}$ processors for any positive constant ϵ or, alternatively, in $O(\log\log n)$ time with $n/\log\log n$ processors [211].

3.3 List Ranking

List ranking is a more general version of the parallel prefix problem: The elements are given as a linked list, i.e., we are given an array A each entry of which contains an element as well as a *pointer* to the entry of A containing the predecessor of that element in the linked list. The problem is to compute an array B such that $B(i)$ is the "sum" of the first i elements in the linked list. This problem is considerably harder than parallel prefix, and most tree computations as well as many graph computations reduce, via the *Euler tour technique* (described below), to solving this problem. EREW PRAM algorithms that run in $O(\log n)$ time and $n/\log n$ processors are known for list ranking [90, 24].

3.4 Tree Contraction

Given a (not necessarily balanced) rooted tree T, the problem is to reduce T to a 3-node tree by a sequence of *rake* operations. A rake operation can be applied at a leaf v by removing v and the parent of v, and making the sibling of v a child of v's grandparent (note that rake cannot be applied at a leaf whose parent is the root). This is done as follows: Number the leaves 1, 2, ..., etc., in left to right order, and apply the rake operation first to all the odd-numbered leaves that are left children, then to the other odd-numbered leaves. Renumber the remaining leaves and repeat this operation, until done. The number of iterations performed by such an algorithm is logarithmic because the number of leaves is reduced by half at each iteration. Note that applying the rake to all the odd-numbered leaves at the same time would not work, as can be seen by considering the situation where v is an odd-numbered left child, w is an odd-numbered right child, and the parent of v is the grandparent of w (what goes wrong in that case is that v wants to remove its parent p and simultaneously w wants its sibling to become child of p). Tree contraction is an abstraction of many other problems, including that of evaluating an arithmetic expression tree [175]. Many elegant optimal EREW PRAM algorithms for it are known [1, 90, 128, 158], running in $O(\log n)$ time with $n/\log n$ processors.

3.5 Brent's Theorem

This technique is frequently used to reduce the processor complexity of an algorithm without any increase in the time complexity.

Theorem 3.1 (Brent): *Any synchronous parallel algorithm taking time T that consists of a total of W operations can be simulated by P processors in time $O((W/P) + T)$.*

There are actually two qualifications to the above Brent's theorem [60] before one can apply it to a PRAM: (i) At the beginning of the i-th parallel step, we must be able to compute the amount of work W_i done by that step, in time $O(W_i/P)$ and with P processors, and (ii) we must know how to assign each processor to its task. Both (i) and (ii) are generally (but not always) easily satisfied in parallel geometric algorithms, so that the hard part is usually achieving W operations in time T.

3.6 Euler Tour Technique

"Wrapping a chain" around a tree defines an Euler tour of that tree. More formally, if T is an undirected tree, the Euler tour of T is obtained in parallel by doing the following for every node v of T: Letting $w_0, w_1, \ldots, w_{k-1}$ be the nodes of T adjacent to v, in the order in which they appear in the adjacency list of v, we set the successor of each (directed) edge (w_i, v) equal to the (directed) edge $(v, w_{(i+1) \bmod k})$. See Figure 2 for an illustration. Most tree computations as well as many graph computations reduce, via the Euler tour technique, to the list ranking problem [216], as the following examples demonstrate.

1. Rooting an undirected tree at a designated node v: Create the Euler tour of the tree, "open" the tour at v (thus making the tour an Euler path), then do a parallel prefix along the linked list of arcs described by the successor function (with a weight of 1 for

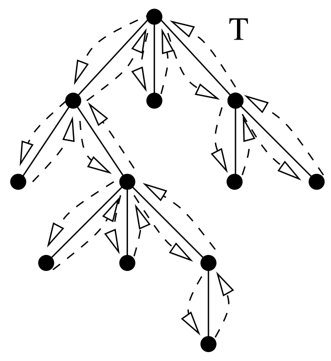

Figure 2: The Euler tour of a tree T.

each arc). For each undirected edge $\{x, y\}$ of the tree, if the prefix sum of the directed edge (x, y) in the Euler tour is less than that of the directed edge (y, x), then set x to be the parent of y. This parallel computation of parents makes the tree rooted at v.

2. Computing post-order numbers of a rooted tree, where the left-to-right ordering of the children of a vertex is the one implied by the Euler path: Do a list ranking on the Euler path, where each directed edge (x, y) of the Euler path has a weight of unity if y is the parent of x, and a weight of zero if x is the parent of y.

3. Computing the depth of each node of a rooted tree: Do a list ranking on the Euler path, where each directed edge (x, y) of the Euler path has a weight of -1 if y is the parent of x, and a weight of $+1$ if x is the parent of y.

4. Numbering of descendants of each node: Same computation as for post-order numbers, followed by making use of the observation that the number of descendants of v equals the list rank of $(v, parent(v))$ minus that of $(parent(v), v)$.

These examples of reductions of so many problems to list ranking demonstrate the importance of the list ranking problem. See [216, 153] for other examples, and for more details about the above reductions.

3.7 Lowest Common Ancestors (LCA)

The problem is to preprocess a rooted tree T so that a lowest common ancestor (LCA) query of any two nodes x, y of T can be answered in constant time by one processor. The preprocessing is to be done in logarithmic time and $n/\log n$ EREW PRAM processors. The problem is easy when T is a simple path (list ranking does the job), or when it is a complete binary tree (it is then solved by comparing the binary representations of the in-order numbers of x and y, specifically finding the leftmost bit where they disagree). For a general tree, the problem is reduced to that of the *range minima*: Create an Euler tour of the tree, where each node v knows its first and last appearance on that tour as well as its depth (i.e., the distance from the root) in the tree. This reduces the problem of answering an LCA query to determining, in constant sequential time, the smallest entry in between two given indices i, j in an array. This last problem is called the range-minima problem. The book [153] contains more details on the reduction of LCA to range minima, a solution to range minima, and references to the relevant literature for this problem.

The above list of basic subproblems is not exhaustive in that (i) many techniques that are basic for general combinatorial problems were omitted (we have focused only on those most relevant to geometric problems rather than to general combinatorial problems), and (ii) among the techniques applicable to geometric problems, we have postponed covering the more specialized ones.

4 Inherently Sequential Geometric Problems

Most of the problems shown to be P-complete to date are not geometric (most are graph or algebra problems). This is no accident: Geometric problems in the plane tend to have enough structures to enable membership in NC. Even the otherwise P-complete problem of linear programming [119, 120] is in NC when restricted to the plane. In the rest of this section, we mention the (very few) planar geometric problems that are known to be P-complete, and also a problem that is conjectured to be P-complete. Several of the problems known to be P-complete involve a collection of line segments in the plane.

The P-completeness proofs of the three problems in the following subsections were given in [28]; for the third problem see also [149]. The proofs consist of giving NC reductions from the monotone circuit value problem and planar circuit value problem, which are known to be P-complete [129, 160, 181]. These reductions typically involve the use of geometry to simulate a circuit, by utilizing the relative positions of objects in the plane.

4.1 Plane-Sweep Triangulation

One is given a simple n-vertex polygon P (which may contain holes) and asked to produce the triangulation that would be constructed by the following sequential algorithm: Sweep the plane from left to right with a vertical line L, such that each time L encounters a vertex v of P one draws all diagonals of P from v that do not cross previously drawn diagonals. This problem is a special case of the well-known polygon triangulation problem (see [124, 193]), and it clearly has a polynomial time sequential solution.

4.2 Weighted Planar Partitioning

Suppose one is given a collection of n non-intersecting line segments in the plane, such that each segment s is given a distinct weight $w(s)$, and asked to construct the partitioning of the plane produced by extending the segments in the sorted order of their weights. The extension of a segment "stops" at the first segment (or segment extension) that is "hit" by the extension. This problem has applications to "art gallery problems" [98, 186], and is P-complete even if there are only 3 possible orientations for the line segments. It is straightforward to solve it sequentially in $O(n \log^2 n)$ time (by using the dynamic point-location data structure of [194]), and in $O(n \log n)$ time by a more sophisticated method [98].

4.3 Visibility Layers

One is given a collection of n non-intersecting line segments in the plane, and asked to label each segment by its "depth" in terms of the following layering process (which starts with $i = 0$): Find the segments that are (partially) visible from point $(0, +\infty)$, label each such segment as being at depth i, remove each such segment, increment i, and repeat until no segments are left. This is an example of a class of problems in computational geometry known as *layering* problems or *onion peeling* problems [63, 165, 187], and is P-complete even if all the segments are horizontal.

4.4 Open Problems

Perhaps the most famous open problem in the area of geometric P-completeness is that of the convex layers problem [63]: Given n points in the plane, mark the points on the convex hull of the n points as being layer zero, then remove layer zero and repeat the process, generating layers $1, 2, \ldots,$ etc. See Figure 3 for an example. Some recent work has shown that a generalization of the convex layers problem, called multi-list ranking [114], is indeed P-complete. Although the results in [114] shed some light on the convex layers problem, the P-completeness of the convex layers problem remains open.

In view of the P-completeness of the above-mentioned visibility layers problem, it seems

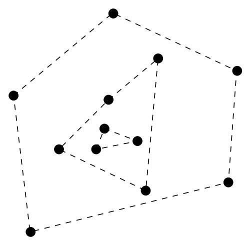

Figure 3: The convex layers of a point set in the plane.

reasonable to conjecture that the convex layers problem is also P-complete; however not all layering problems are P-complete. For example, the layers of maxima problem (defined analogously to convex layers but with the words "maximal elements" playing the role of "convex hull") is easily shown to be in NC by a straightforward reduction to the computation of longest paths in a directed acyclic graph [31] (each input point is a vertex, and there is a directed edge from point p to point q iff both the x and y coordinates of p are $\geq$ those of q, respectively).

5 Parallel Divide and Conquer

As in sequential algorithms, divide and conquer is a useful technique in parallel computation. Parallel divide and conquer comes in many flavors which we discuss next.

5.1 Two-Way Divide and Conquer

The sequential divide and conquer algorithms that have efficient PRAM implementations are those for which the "conquer" step can be done extremely fast (e.g., in constant time). Take, for example, an $O(n \log n)$ time sequential algorithm that works by recursively solving two problems of size $n/2$ each, and then combining the answers they return in linear time. In order for a PRAM implementation of such an algorithm to run in $O(\log n)$ time with n processors, the n processors must be capable of performing the "combine" stage in constant time. For some geometric problems this is indeed possible (e.g., the convex hull problem [43, 224]). The time complexity $T(n)$ and processor complexity $P(n)$ of such a PRAM implementation then obey the recurrences

$$T(n) \leq T(n/2) + c_1,$$

$$P(n) \leq \max\{n, 2P(n/2)\},$$

with boundary conditions $T(1) \leq c_2$ and $P(1) = 1$, where c_1 and c_2 are positive constants. These imply that $T(n) = O(\log n)$ and $P(n) = n$.

But for many problems, such an attempt at implementing a sequential algorithm fails because of the impossibility of performing the "conquer" stage in constant time. For these, the next approach often works.

5.2 "Rootish" Divide and Conquer

By "rootish", we mean partitioning a problem into $n^{1/k}$ subproblems to be solved recursively in parallel, for some positive constant integer k (usually, $k = 2$). For example, instead of dividing the problem into two subproblems of size $n/2$ each, we divide it into (say) $\sqrt{n}$ subproblems of size $\sqrt{n}$ each, which we recursively solve in parallel. That the conquer stage takes $O(\log n)$ time (assuming it does) causes no harm with this subdivision scheme, since the time and processor recurrences in that case would be

$$T(n) \leq T(\sqrt{n}) + c_1 \log n,$$

$$P(n) \leq \max\{n, \sqrt{n}P(\sqrt{n})\},$$

with boundary conditions $T(1) \leq c_2$ and $P(1) = 1$, where c_1 and c_2 are positive constants. These imply that $T(n) = O(\log n)$ and $P(n) = n$.

The problems that can be solved using rootish divide and conquer scheme include the convex hull of points [4, 42, 131], the visibility of non-intersecting planar segments from a point [53], the visibility of a polygonal chain from a point [36], the convex hull of a simple polygon [71], detecting various types of weak visibility of a simple polygon [69, 72, 73, 74], triangulating useful classes of simple polygons [71, 132], and the determination of monotonicity of a simple polygon [76]. This scheme also finds success in other computational models such as the hypercube [32, 70]. The scheme is useful in various ways and forms, and sometimes with recurrences very different from the above-mentioned ones, as the next example demonstrates.

5.3 Example: Visibility in a Polygon

Given a *source* point q and a simple n-vertex polygonal chain P in the plane, this visibility problem is that of finding all the points of P that are visible from q if P is "opaque". We seek an algorithm that takes $O(\log n)$ time and uses $O(n/\log n)$ EREW PRAM processors, which is optimal to within a constant factor because (i) there is an obvious $\Omega(n)$ sequential

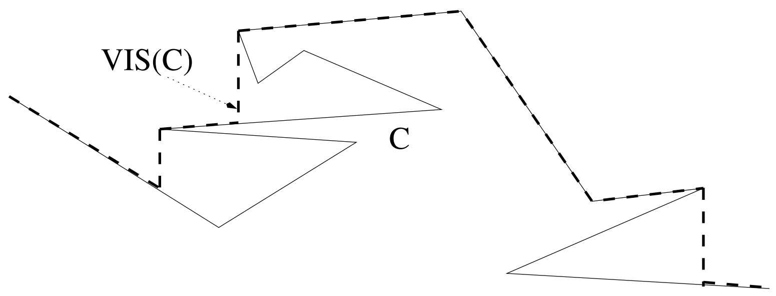

Figure 4: The visibility chain $VIS(C)$ of a polygonal chain C from the point $q = (0, \infty)$.

lower bound for the problem, and (ii) an $\Omega(\log n)$ lower bound on its EREW PRAM time complexity can be obtained by reducing to it the problem of computing the maximum of n entries, a problem with a known logarithmic time lower bound [94].

This is one instance of a problem in which one has to use a hybrid of two-way divide and conquer and rootish divide and conquer, in order to obtain the desired complexity bounds. The recursive procedure we sketch below follows [36] and takes two input parameters (one of which is the problem size) and uses either fourth-root divide and conquer or two-way divide and conquer, depending on the relative sizes of these two parameters. The role played by the geometry is central to the "combine" step (other algorithms of this kind can be found in [68, 71, 72, 73, 74, 130, 131, 132] for solving many problems on polygons and point sets).

We call **VisChain** the recursive procedure for computing the visibility chain of a simple polygonal chain from the given source point q (see Figure 4 for an example of the visibility chain $VIS(C)$ from the point $q = (0, \infty)$). The procedure is outlined below. Let $|C|$ denote the number of vertices of a polygonal chain C. The initial call to the procedure is **VisChain**$(P, n, \log n)$, where P is a simple polygonal chain and $n = |P|$.

VisChain(C, m, d)

Input: A simple polygonal chain C, $m = |C|$, and a positive integer d of our choice.

Output: The visibility chain of C, $VIS(C)$, from the point q.

Step 1. If $m \leq d$, then compute $VIS(C)$ with one processor in $O(m)$ time, using any of the known sequential linear-time algorithms.

Step 2. If $d < m \leq d^2$, then divide C into two subchains C_1 and C_2 of equal length and recursively call **VisChain**$(C_1, |C_1|, d)$ and **VisChain**$(C_2, |C_2|, d)$ in parallel.

Then compute $VIS(C)$ from $VIS(C_1)$ and $VIS(C_2)$, in $O((\log m)^2)$ time and using one processor.

Step 3. If $m > d^2$, then partition C into $g = (m/d)^{1/4}$ subchains $C_1, C_2, \ldots, C_g$ of length $m^{3/4}d^{1/4}$ each. In parallel, call $\mathbf{VisChain}(C_1, |C_1|, d)$, $\mathbf{VisChain}(C_2, |C_2|, d)$, $\ldots$, $\mathbf{VisChain}(C_g, |C_g|, d)$. Then compute $VIS(C)$ from $VIS(C_1)$, $VIS(C_2)$, $\ldots$, $VIS(C_g)$, in $O(\log m)$ time and using m/d processors (we henceforth use g to denote $(m/d)^{1/4}$).

end.

The main difficulty lies in the "conquer" steps: Even though C_i and C_j do not intersect, there can be more than one intersection between their visibility chains $VIS(C_i)$ and $VIS(C_j)$, and we have (cf. Step 3 above) only $g^2 = (m/d)^{1/2}$ processors to compute these intersections between each pair $(VIS(C_i), VIS(C_j))$. Doing this in $O(\log m)$ time may appear impossible at first sight: the length of each of $VIS(C_i)$ and $VIS(C_j)$ can be $m^{3/4}d^{1/4}$, and there is a well-known *linear* lower bound [65] on the work needed for computing the two intersections of two arbitrary simple polygonal chains that intersect twice. This seems to imply that, since we have only $(m/d)^{1/2}$ processors assigned to the task, it will take $m^{3/4}d^{1/4}(m/d)^{-1/2}$ $= m^{1/4}d^{3/4}$ time rather than the claimed $O(\log m)$. What enables us to achieve $O(\log m)$ time is the fact that both C_i and C_j are subchains of a simple polygonal chain.

Observe that, if we could perform the various steps of the above algorithm within the claimed bounds, then it would indeed run in $O(d + \log m)$ time with $O(m/d)$ processors since its time and processor complexities would satisfy the following recurrences:

$$t(m,d) = \begin{cases} c_1 m & \text{if } m \le d \\ t(m/2, d) + c_2(\log m)^2 & \text{if } d < m \le d^2 \\ t(m^{3/4}d^{1/4}, d) + c_3 \log m & \text{if } d^2 < m \end{cases}$$

$$p(m,d) = \begin{cases} \max\{1,\ m/d\} & \text{if } m \le d \\ \max\{2p(m/2, d),\ m/d\} & \text{if } d < m \le d^2 \\ \max\{(m/d)^{1/4}p(m^{3/4}d^{1/4}, d),\ m/d\} & \text{if } d^2 < m \end{cases}$$

where c_1, c_2, c_3 are constants. From the above recurrences, the following bounds for $t(m,d)$ and $p(m,d)$ are easy to prove by induction:

$$t(m,d) \le \begin{cases} \alpha_1 d & \text{if } m \le d \\ \alpha_2 d + \beta_2(\log m)^2 \log d & \text{if } d < m \le d^2 \\ \alpha_3 d + \beta_3 \log m & \text{if } d^2 < m \end{cases}$$

$$p(m,d) = m/d$$

where α_1, α_2, β_2, α_3, β_3 are constants.

Choosing d to be $\log m$, the above implies that $t(m, \log m) = O(\log m)$ and $p(m, \log m) = O(m/\log m)$. Hence the call **VisChain**$(P, n, \log n)$ would compute $\mathit{VIS}(P)$ in $O(\log n)$ time using $O(n/\log n)$ processors.

Thus it suffices to show how, with m/d processors, to do the "combine" part of Step 2 in $O((\log m)^2)$ time, and more importantly, how to implement the "combine" part of Step 3 in $O(\log m)$ time. The geometric facts that make this possible are:

- If C' is a subchain of a chain C then $\mathit{VIS}(C) \cap \mathit{VIS}(C')$ has at most three connected components (i.e., at most three separate portions of $\mathit{VIS}(C')$ appear in $\mathit{VIS}(C)$).
- If C' and C'' are two subchains of C that are disjoint except that they may share one endpoint, then there are at most *two* intersections between $\mathit{VIS}(C')$ and $\mathit{VIS}(C'')$.

Although the above limits to two the number of possible intersections between the visibility chains of two subchains of C that are disjoint (except possibly at a common endpoint), the linear-work lower bound we mentioned earlier for detecting intersections between polygonal chains [65] holds even for two chains that intersect each other no more than twice. The solution in [36] exploits the fact that the two chains are subchains of a simple chain in order to get around the lower bound when solving the two-intersection case. Specifically, it shows how to compute, for each C_i, the (at most two) portions of $\mathit{VIS}(C_i)$ that are hidden. Once one has done this (in parallel) for every $i \in \{1, \ldots, g\}$, it is easy to "stitch" the resulting $g' \leq 3g$ pieces of $\mathit{VIS}(C)$ and create $\mathit{VIS}(C)$: first split the trees representing $\mathit{VIS}(C_1)$, $\mathit{VIS}(C_2)$, $\ldots$, $\mathit{VIS}(C_g)$, in order to discard all portions of the $\mathit{VIS}(C_i)$'s that are invisible in $\mathit{VIS}(C)$; then the problem essentially becomes that of sorting (by the polar angles) the $O(g)$ endpoints of those portions of the $\mathit{VIS}(C_i)$'s that are visible in $\mathit{VIS}(C)$, which can be done in time $O(\log m)$ using $O(g)$ EREW PRAM processors [87]. There are g^4 processors available, more than enough to do this sorting.

6 Cascading

Cascading is a sampling and iterative refinement method that was introduced by Cole [87] for the sorting problem, and was further developed in [37, 130] for the solutions of geometric problems. It has proved to be a fundamental technique, one that enables optimal solutions when most other techniques fail. Since its details are intricate even for sorting, the next subsection gives a rough sketch – too broad to be useful but enough to give the reader a feel for the main idea (later subsections give a more detailed description and illustrate with examples).

6.1 A Rough Sketch of Cascading

Since this technique works best for problems that are solved sequentially by divide and conquer, we use such a hypothetical problem to illustrate the discussion: Consider an $O(n \log n)$ time sequential algorithm that works by recursively solving two subproblems of size $n/2$ each, followed by an $O(n)$ time conquer stage. Let T be the tree of recursive calls for this algorithm, i.e., a node of this recursion tree at height h corresponds to a subproblem of size equal to the number of leaves in its subtree ($= 2^h$). A "natural" way of parallelizing such an algorithm would be to mimic it by using n processors to process T in a bottom up fashion, one level at a time, completing level h before moving to level $h+1$ of T (where by "level h", we mean the set of nodes of T whose height is h). Such a parallelization will yield an $O(\log n)$ time algorithm only if processing each level can be done in constant time. It can be quite nontrivial to process one level in constant time, so this natural parallelization can be challenging. However, it is frequently the case that processing one level cannot be done in constant time, and it is precisely in these situations that the cascading idea can be useful. In order to be more specific when sketching this idea, we assume that the hypothetical problem being solved is about a set S of n points, with the points stored in the leaves of T.

In a nutshell, the general idea of cascading is as follows. The computation proceeds in a logarithmic number of stages, each of which takes constant time. Each stage involves activity by the n processors at more than one level, so the computation diffuses up the tree T, rather than working on only one level at a time. For each node $v \in T$, let $h(v)$ be the height of v in T, $L(v)$ be the points stored in the leaves of the subtree of v in T, and let $I(L(v))$ be the information we seek to compute for node v (the precise definition of $I(\cdot)$ varies from problem to problem). The ultimate goal is for every $v \in T$ to compute the $I(L(v))$ array. Each $v \in T$ lies "dormant" and does nothing until the stage number exceeds a certain value (usually $h(v)$), at which time node v "wakes up" and starts computing, from stage to stage, $I(L')$ for a progressively larger subset L' of $L(v)$, a subset L' that (roughly) doubles in size from one stage to the next of the computation. $I(L')$ can be thought of as an approximation of the desired $I(L(v))$, an approximation that starts out being very rough (when L' consists of, say, a single point) but gets repeatedly refined from one stage to the next. When L' eventually becomes equal to $L(v)$, node v becomes inactive for all the future stages (i.e., it is done with its computation, since it now has $I(L(v))$). There are many (often intricate) implementation details that vary from one problem to the next, and many times the scheme substantially deviates from the above rough sketch, but the above nevertheless gives the general idea of cascading.

The cascading technique has been used to solve many problems (not just geometric ones). The next section explains in more detailed the basic version of the technique, for the problem of sorting. Some of the geometric problems for which it has been used are covered later, in order to illustrate variants of the method. The exposition follows [37], with proofs and details omitted.

6.2 Cascading Merge Sort

In this section we sketch the cascading technique for the merge sorting problem; later we show how it can be modified to solve geometric problems. Suppose we are given a binary tree T (not necessarily complete) with items, taken from some total order, placed at the leaves of T, so that each leaf contains at most one item. For simplicity, we assume that the items are distinct. We wish to compute for each internal node $v \in T$ the sorted list $U(v)$ that consists of all the items stored in descendant nodes of v. 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. Without loss of generality, we assume that every internal node v of T has two children (otherwise we can add an extra leaf child that is empty, i.e., does not store any items from the total order).

Let two sorted (nondecreasing) lists $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ be given. We define the *rank* of an element a_i in B to be the number of elements in B that are less than a_i. 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 $A \cup B$ 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 that are not in B.

Let T be a binary tree. For any node v in T the *altitude*, denoted $alt(v)$, is $height(T) - depth(v)$. $Desc(v)$ denotes the set of descendant nodes of v (including v itself).

Let a sorted list L and a sorted list J be given. 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. We let

$SAMP_c(L)$ denote the sorted list consisting of every c-th element of L, and call this list 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. Intuitively, in each stage we will be performing a portion of the merge of $U(lchild(v))$ and $U(rchild(v))$ to give the list $U(v)$. After performing a portion of this merge we will gain some insight into how to perform the merge at v's parent. Consequently, we will pass some of the elements formed in the merge at v to v's parent, so we can begin performing the merge at v's parent.

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

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

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

Per-Stage Computation$(v, s+1)$**:**

Form the two lists $U'_{s+1}(lchild(v))$ and $U'_{s+1}(rchild(v))$, and compute the new list $U_{s+1}(v) := U'_{s+1}(lchild(v)) \cup U'_{s+1}(rchild(v))$.

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

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

Lemma 6.1: *For any stage $s \geq 0$ and any node $v \in T$, $|U_{s+1}(v)| \leq 2|U_s(v)| + 4$.*

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

Lemma 6.2: *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 $s \geq 1$.*

Corollary 6.3: *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. In addition to this corollary, we also need to maintain the following rank information at the start of each stage s:

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

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

The lemma that follows states 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 6.4 (The Merge Lemma): *Suppose one is given sorted lists A_s, A'_{s+1}, B'_s, B'_{s+1}, C'_s, and C'_{s+1}, where the following (input) conditions are true:*

1. *$A_s = B'_s \cup C'_s$;*

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

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

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

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

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

7. *B'_s and C'_s are cross ranked.*

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

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

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

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

We apply this lemma by setting $A_s = U_s(v)$, $A'_{s+1} = U'_{s+1}(v)$, $A_{s+1} = U_{s+1}(v)$, $B'_s = U'_s(x)$, $B'_{s+1} = U'_{s+1}(x)$, $C'_s = U'_s(y)$, and $C'_{s+1} = U'_{s+1}(y)$, where $x = \mathit{leftchild}(v)$ and $y = \mathit{rightchild}(v)$. Note that assigning the lists of Lemma 6.4 in this way satisfies input conditions 1–4 from the definitions. The ranking information we maintain from stage to stage satisfies input conditions 5–7. Thus, in each stage s we can construct the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors. Also, the new ranking information (of output computations 2 and 3) gives us the input conditions 5–7 for the next stage. By Corollary 6.3 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 lists for $s - 1$; we can discard any lists for stages previous to that.

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

It will often be more convenient to relax the condition that there is at most one item stored at each leaf. So, suppose there is an unsorted set $A(v)$ (which may be empty) stored at each leaf. In this case one can construct a tree T' from T by replacing each leaf v of

T with a complete binary tree with $|A(v)|$ leaves, and associating each item in $A(v)$ with one of these leaves. T' would now satisfy the conditions of the method outlined above. We incorporate this observation in the following theorem, which summarizes the discussion of this section:

Theorem 6.5: *Suppose one is given a binary tree T such that there is an unsorted set $A(v)$ (which may be empty) stored at each leaf. Then one 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. This computation can be implemented in $O(height(T) + \log(\max_v |A(v)|))$ time using a total of $n + N$ processors in the CREW PRAM computational model, where n is the number of leaves of T and N is the total number of items stored in T.*

The next three sections give specific examples of other applications and extensions of the cascading technique.

7 Fractional Cascading in Parallel

Fractional cascading is a fundamental technique that has a large number of different applications in geometry [67]. First we briefly review the problem. Given a directed graph $G = (V, E)$, such that every node v contains a sorted list $C(v)$ (called a *catalogue*), the fractional cascading problem is to construct an $O(n)$ space data structure that, given a walk $(v_1, v_2, \ldots, v_m)$ in G and an arbitrary element x, enables a single processor to locate x quickly in each $C(v_i)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. Fractional cascading problems arise naturally from a number of computational geometry problems. As a simple example of a fractional cascading problem, suppose one has five different English dictionaries and would like to build a data structure that would allow one to look up a word w in all the dictionaries. Chazelle and Guibas give an elegant $O(n)$ time sequential method for constructing a fractional cascading data structure from any graph G, as described above, achieving a search time of $O(\log n + m \log d(G))$, where $d(G)$ is the maximum degree of any node in G. However, their approach does not appear to be "parallelizable."

In this section, which follows [37], we show how to construct a data structure achieving the same performance as that of Chazelle and Guibas in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors. The method begins with a preprocessing step similar to one used by Chazelle and Guibas where we "expand" each node of G into two binary trees—one for its in-edges and one for its out-edges—so that each node in our graph has in-degree and out-degree at most 2. We then perform a cascading merge procedure in stages on this graph. Each catalogue $C(v)$ is "fed into" the node v in samples that double in size with each stage and these lists are in turn sampled and merged along the edges of G. Lists continue to be sampled and "pushed" across the edges of G (even in cycles) for a logarithmic number of stages, at which time we stop the computation and add some links between elements in adjacent lists. We show that this gives us a fractional cascading data structure, and that the computation can be implemented in $O(\log n)$ time and $O(n)$ space using $\lceil n/\log n \rceil$ processors.

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

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 by $d(v)$, is defined as $d(v) = \max\{|In(v, G)|, |Out(v, G)|\}$. The *degree* of G, denoted by $d(G)$, is defined as $d(G) = \max_{v \in V}\{d(v)\}$. A sequence $(v_1, v_2, \ldots, v_m)$ of vertices is a *walk* if $(v_i, v_{i+1}) \in E$ for all $i \in \{1, 2, \ldots, m-1\}$.

As mentioned above, we begin the construction by preprocessing the directed graph G to convert it into a directed graph $\hat{G} = (\hat{V}, \hat{E})$ such that $d(\hat{G}) \leq 2$ and such that an edge (v, w) in G corresponds to a path in $\hat{G}$ of length at most $O(\log d(G))$. Specifically, for each node $v \in V$ we construct two complete binary trees T_v^{in} and T_v^{out}. Each leaf in T_v^{in} (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. Note that $d(\hat{G}) = 2$. We assume that for each node v we have access to the nodes in $In(v, \hat{G})$ as well as those in $Out(v, \hat{G})$. We structure fan-out trees so that a processor needing to go from v to w in $\hat{G}$, with $(v, w) \in E$, can correctly determine the path down T_v^{out} to the leaf corresponding to (v, w). More specifically, the leaves of each fan-out tree are ordered so that they are listed from left to right by increasing destination name, i.e., if the leaf in T_v^{out} for $e = (v, u)$ is to the left of the leaf for $f = (v, w)$, then $u < w$. (The leaves of T_v^{in} need not be sorted, since all edges are directed towards the root of that tree.) If we are not given the $Out(v, G)$ sets in sorted order, then we must perform a sort as a part of the T_v^{out} construction, which can be done in $O(\log d(G))$ time using n processors using Cole's merge sorting algorithm [87]. We also store in each internal node z of T_v^{out} the leaf node u that 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. This is where the resemblance to the sequential algorithm ends, however.

The goal for the rest of the computation is to construct a special sorted list $B(v)$, which we call the *bridge list*, for every node $v \in \hat{V}$. We shall define these bridge lists so that $B(v) = C(v)$ if v is in V, 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 samples of the catalogue $C(v)$ that double in size with each stage. These samples are then cascaded back into the gateway v' for v and from there back through the fan-in tree for v. We will also be merging any samples "passed back" from the fan-out tree for v with $B_s(v')$, and cascading these values back through the fan-in tree for v as well. We iterate the per-stage computation for $\lceil \log N \rceil$ stages, where N is the size of the largest catalogue in G. We will show that after we have completed the last stage, and updated some ranking pointers, $\hat{G}$ will be a fractional cascading data structure for G. More details follow.

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

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

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

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

Per-Stage Computation$(v, s+1)$**:**

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

The function $c(s)$ is defined so that if $v \in V$, then as the computation proceeds the list $B'_{s+1}(v)$ will be empty for a while, then at some stage $s+1$ it will consist of a single element of $C(v)$ (the $(2^{\lceil \log N \rceil - s})$-th element), in stage $s+2$ at most three elements (evenly sampled), in stage $s+3$ at most five elements, in stage $s+4$ at most nine elements, and so on. This continues until the final stage (stage $\lceil \log N \rceil$), when $B'_{s+1}(v) = C(v)$. Intuitively, the $c(s)$ function is a mechanism for synchronizing the processes of "feeding" the $C(v)$ lists into the nodes of $\hat{G}$ so that all the processes complete at the same time. We show below that each stage can be performed in $O(1)$ time, resulting in a running time of the cascading computations that is $O(\log N)$ (plus the time it takes time to compute the value of N: namely, $O(\log n)$). The following important lemma is similar to Lemma 6.1 in that it guarantees that the bridge lists do not grow "too much" from one stage to another.

Lemma 7.1: *For any stage* $s \geq 0$ *and any node* $v \in T$, $|B_{s+1}(v)| \leq 2|B_s(v)| + 4$.

Corollary 7.2: *The list* $(-\infty, B'_s(v), \infty)$ *is a* 4*-cover for* $B'_{s+1}(v)$*, for* $s \geq 0$.

Corollary 7.3: *The list* $(-\infty, B_s(v), \infty)$ *is a* 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 6.4) 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 lists for $s-1$; we can discard any lists for stages previous to that, as in the generalized cascading merge.

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

Given a walk $W = (v_1, \ldots, v_m)$, and an arbitrary element x, the query that asks for locating x in every $C(v_i)$ is called the *multilocation* of x in $(v_1, \ldots, v_m)$. To perform a multilocation of x in a walk $(v_1, \ldots, v_m)$ we extend the walk W in G to its corresponding walk $\hat{W} = (\hat{v}_1, ..., \hat{v}'_m)$ in $\hat{G}$ and perform the corresponding multilocation in $\hat{G}$, similar to the method given by Chazelle and Guibas for performing multilocations in their data structure. The multilocation begins with the location of x in $B(\hat{v}_1) = B(v'_1)$, the gateway bridge list for v_1, by binary search. For each other vertex in this walk we can locate the position of x in $B(\hat{v}_i)$ given its position in $B(\hat{v}_{i-1})$ in $O(1)$ time. The method is to follow the pointer from x's predecessor in $B(\hat{v}_{i-1})$ to the predecessor of that element in $B(\hat{v}_i)$ and then locate x in $B(\hat{v}_i)$ by a linear search from that position (which will require at most 13 comparisons by Corollary 7.3). In addition, if $\hat{v}_i$ corresponds to a gateway v', then we can locate x in $C(v)$ in $O(1)$ time given its position in $B(v')$ by a similar argument. Since each edge in the walk W corresponds to a path in $\hat{W}$ of length at most $O(\log d(G))$, this implies that we can perform the multilocation of x in $(v_1, \ldots, v_m)$ in $O(\log |B(v'_1)| + m \log d(G))$ time. In other words, $\hat{G}$ is a fractional cascading data structure. The next lemma states that $\hat{G}$ uses $O(n)$ space.

Lemma 7.4: *Let n_v denote the amount of space that is added to $\hat{G}$ because of the presence of a particular catalogue $C(v)$, $v \in V$. Then $n_v \le 2|C(v)|$.*

Corollary 7.5: *The total amount of space used by the fractional cascading data structure is $O(n)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$.*

Note that the upper bound on the space of the fractional cascading data structure holds even if $\hat{G}$ contains cycles. This corollary, then, implies that we can construct a fractional cascading data structure $\hat{G}$ from any catalogue graph G in $O(\log n)$ time and $O(n)$ space using n processors, even if G contains cycles. See [37] for details of how to assign these n processors to their respective jobs. In fact [37] establishes the following stronger result.

Theorem 7.6: *Given a catalogue graph $G = (V, E)$, such that $d(G)$ is $O(1)$ or one is given each $Out(v, G)$ set in sorted order, one can build a fractional cascading data structure for G in $O(\log n)$ time and $O(n)$ space using $\lceil n/\log n \rceil$ processors in the CREW PRAM model, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. This bound is optimal.*

8 Cascading with Labeling Functions

The merging procedure of Section 6.2 can be combined with divide-and-conquer strategies based on merging lists with labels defined on their elements. This gives rise to variants of cascading that resulted in efficient parallel solutions to many geometric problems. We illustrate some of these here. For most of these problems this approach gives an efficient sequential alternative to the known sequential algorithms (which typically use the plane-sweeping paradigm), as well as giving rise to efficient parallel algorithms. We begin with the 3-dimensional maxima problem.

8.1 The 3-Dimensional Maxima Problem

Let $V = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathbf{R}^3$. 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 method is based on cascading a divide-and-conquer strategy in which the merging step involves the computation of two labeling functions for each point. The labels we use are motivated by the optimal sequential plane-sweeping algorithm of Kung, Luccio, and Preparata [163]. Specifically, for each point p_i we compute the maximum z-coordinate from all points which 1-dominate p_i and use that label to also compute the maximum z-coordinate from all points which 2-dominate p_i. We can then test if p_i is a maximum point by comparing $z(p_i)$ to this latter label.

Without loss of generality, assume the input points are given sorted by increasing y-coordinates, i.e., $y(p_i) < y(p_{i+1})$. Let T be a complete binary tree with leaf nodes $v_1, v_2, \ldots, v_n$ (in this order). In each leaf node v_i we store the list $B(v_i) = (-\infty, p_i)$, where $-\infty$ is a special symbol such that $x(-\infty) < x(p_j)$ and $y(-\infty) < y(p_j)$ for all points p_j in V. Initializing T in this way can be done in $O(\log n)$ time using n processors. We then perform a generalized cascading-merge from the leaves of T as in Theorem 6.5, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). Using the notation of Section 6.2, we let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store two labels: $zod(p_i, v)$ and $ztd(p_i, v)$, where $zod(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that 1-dominate p_i, and $ztd(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that 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 $x(p_i)$). As we are performing the cascading-merge, we update the labels zod and ztd based on the equations in the following lemma:

Lemma 8.1: *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following:*

$$zod(p_i, v) = \begin{cases} \max\{zod(p_i, u),\ zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u) \\ \max\{zod(\text{pred}(p_i, u), u),\ zod(p_i, w)\} & \text{if } p_i \in U(w) \end{cases} \quad (1)$$

$$ztd(p_i, v) = \begin{cases} \max\{ztd(p_i, u),\ zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u) \\ ztd(p_i, w) & \text{if } p_i \in U(w) \end{cases} \quad (2)$$

We use these equations during the cascading merge to maintain the labels for each point. By Lemma 8.1, 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. Moreover, after v's parent becomes full we no longer need $U(v)$ any more, and can deallocate the space it occupies, resulting in an $O(n)$ space algorithm, as outlined in Section 6.2. After we complete the merge, and have computed $U(root(T))$, along with all the labels for the points in $U(root(T))$, note that a point $p_i \in U(root(T))$ is a maximum if and only if $ztd(p_i, root(T)) \leq z(p_i)$ (there is no point that 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 8.2: *Given a set V of n points in $\mathbf{R}^3$, one can construct the set M of maximal points in V in $O(\log n)$ time and $O(n)$ space using n processors in the CREW PRAM model, and this is optimal.*

(The optimality follows from the fact that [163] have shown that this problem has an $\Omega(n \log n)$ sequential lower bound.)

It is worth noting that one 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 for $k \geq 3$. The resulting time and space complexities are $O((\log n)^{k-2})$ time using n processors in the CREW PRAM model.

Next, we address the two-set dominance counting problem.

8.2 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 $B = \{r_1, r_2, \ldots, r_l\}$ of points in the plane, and wish to know for each point r_i in B the number of points in A that are 2-dominated by r_i. For simplicity, we assume that the points have distinct x (resp. y) coordinates. The approach to this problem is similar to that of the previous subsection, in that we will be performing a cascading merge procedure while maintaining two labeling functions for each point. In this case the labels maintain for each point p_i (from A or B) how many points of A are 1-dominated by p_i and also how many points of A are 2-dominated by p_i. As in the previous solution, the first label is used to maintain the second. The details follow.

Let $Y = \{p_1, p_2, \ldots, p_{l+m}\}$ be the union of A and B with the points listed by increasing y-coordinate, i.e., $y(p_i) < y(p_{i+1})$. We can construct Y in $O(\log n)$ time using n processors [87], where $n = l + m$. The 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 $v_1, v_2, \ldots, v_n$, in this order, and in each leaf node v_i we store the list $U(v_i) = (-\infty, p_i)$, ($-\infty$ still being a special symbol such that $x(-\infty) < x(p_i)$ and $y(-\infty) < y(p_i)$ for all points p_i in Y). We then perform a generalized cascading-merge from the leaves of T as in Theorem 6.5, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). We let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinate. For each point p_i in $U(v)$ we store two labels: $nod(p_i, v)$ and $ntd(p_i, v)$ The label $nod(p_i, v)$ is the number of points in $U(v)$ that are in A and are 1-dominated by p_i, and the label $ntd(p_i, v)$ is the number of points in $U(v)$ that are in A and are 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 $\Psi_A(p_i)$ as follows: $\Psi_A(p_i) = 1$ if $p_i \in A$, and $\Psi_A(p_i) = 0$ 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 following lemma.

Lemma 8.3: *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following:*

$$nod(p_i, v) = \begin{cases} nod(p_i, u) + nod(\text{pred}(p_i, w), w) + \Psi_A(\text{pred}(p_i, w)) \\ \qquad\qquad\qquad\qquad\qquad \text{if } p_i \in U(u) \\ nod(\text{pred}(p_i, u), u) + nod(p_i, w) + \Psi_A(\text{pred}(p_i, u)) \\ \qquad\qquad\qquad\qquad\qquad \text{if } p_i \in U(w) \end{cases} \tag{3}$$

$$ntd(p_i, v) = \begin{cases} ntd(p_i, u) \qquad\qquad\qquad\qquad \text{if } p_i \in U(u) \\ nod(\text{pred}(p_i, u), u) + ntd(p_i, w) + \Psi_A(\text{pred}(p_i, u)) \\ \qquad\qquad\qquad\qquad\qquad \text{if } p_i \in U(w) \end{cases} \tag{4}$$

By Lemma 8.3, 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 $U(root(T))$, then we are done. We summarize in the following theorem:

Theorem 8.4: *Given a set A of l points in the plane and a set B of m points in the plane, one can compute for each point p in B the number of points in A 2-dominated by p in $O(\log n)$ time and $O(n)$ space using n processors in the CREW PRAM model, where $n = l + m$, and this is optimal.*

There are a number of other problems that can be reduced to two-set dominance counting, and can thus be solved within the same bounds (i.e., in logarithmic time with linear space and processors). We mention two here, the first being the multiple range-counting problem: Given a set V of l points in the plane and a set R of m isothetic rectangles (ranges) the multiple range-counting problem is to compute the number of points interior to each rectangle. Another problem that reduces to two-set dominance counting is rectilinear segment intersection counting: Given a set S of n rectilinear line segments in the plane, determine for each segment the number of other segments in S that intersect it.

8.3 Other Applications of Cascading

Cascading can be used to solve other problems, too many to cover in detail. We mention the following:

- Trapezoidal decomposition. Given a set S of n planar line segments, determine for each segment endpoint p the first segment encountered by starting at p and walking vertically upward (or downward). An $O(\log n)$ time and n-processor CREW PRAM algorithm is known [37]. This implies similar bounds for the polygon triangulation problem [130, 132, 227].

- Topological sorting of n non-intersecting line segments. This is the problem of ordering the segments so that, if a vertical line l intersects segments s_i and s_j and $i < j$, then the intersection between l and s_i is above the intersection between l and s_j. An $O(\log n)$ time, n-processor CREW PRAM algorithm is easily obtained by implementing the main idea of the mesh algorithm of [45] (which reduces the problem to a trapezoidal decomposition computation followed by a tree computation).

- Planar point location. Given a subdivision of the plane into polygons called faces, build a data structure that enables one processor to quickly locate, for any query point, the face containing it. Using n processors, cascading can be used to achieve $O(\log n)$ time for both construction and query [37, 214, 93]. The planar point location problem itself tends to arise rather frequently, even in geometric problems apparently unrelated to it.

- Intersection detection, all nearest neighbors, visibility from a point. For these problems, cascading can be used to achieve $O(\log n)$ time with n processors [37, 88].

Alternative approaches to cascading have been proposed for some of the above problems; for example, see [53, 197, 226] and also the elegant parallel hierarchical approach of Dadoun and Kirkpatrick [100, 99].

9 Cascading Without Concurrent Reads

In this section we briefly note that the same techniques as employed by Cole in [87] 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, and many others, resulting in EREW PRAM algorithms for these problems. Applying Cole's techniques to these algorithms results in EREW PRAM algorithms with the same asymptotic bounds except that the space bounds for the problems addressed in Section 8 all become $O(n \log n)$. The reason that his techniques increase the space complexity of these problems is because of our use of labeling functions. Specifically, it is not clear how to perform the merges on-line and still update the labels in $O(1)$ time after a node becomes full. This is because a label whose value changes on level l may have to be broadcast to many elements in level $l-1$ to update their labels, which would require $\Omega(\log n)$ time in this model if there were $O(n)$ such elements.

One can get around the problems arising from the labeling functions, however [37]. For the 3-dimensional maxima problem and the two-set dominance counting problem we separate the computation of the $U(v)$ lists and computation of the labeling functions into two separate steps, rather than "dove-tailing" the two computations as before. Each of the labeling functions we used for these two problems can re-defined so as to be EREW-computable. Specifically, the label for an element p in $U(v)$, on level l, can be expressed in terms of a label $pref(p, v)$ and a label $up(p, v)$, where $pref(p, v)$ can be computed by performing a parallel prefix computation [159, 164] in $U(v)$ and $up(p, v)$ can be defined in terms of $pref(pred(p, lchild(v)), lchild(v))$, $pref(pred(p, rchild(v)), rchild(v))$, and the up label p had on level $l+1$ (say, in $U(rchild(v))$ if $p \in U(rchild(v))$). In particular, for the 3-dimensional maxima problem $pref(p,v) = zod(p,v)$ and $up(p,v) = ztd(p,v)$, and for the two-set dominance counting problem $pref(p,v) = nod(p,v)$ and $up(p,v) = ntd(p,v)$. One can compute all the $pref(p,v)$ labels in $O(\log n)$ time using n processors by assigning $\lceil |U(v)|/\log n \rceil$ processors to each node v [159]. One can then broadcast each $pref(p,v)$ label to the successor of v in $sibling(v)$, which takes $O(\log n)$ time using n processors by assigning $\lceil |U(v)|/\log n \rceil$ processors to each node v. Finally, one can compute all the $up(p,v)$ labels in $O(\log n)$ additional time by assigning a single processor to each point p and tracing the path in the tree from the leaf node which contains p up to the root. This is an EREW operation because computing all the $up(p,v)$ labels only depends upon accessing memory locations associated with the point p.

The EREW solution to the visibility from a point problem requires $O(n \log n)$ space for a different reason (cf. [37]).

10 Matrix Searching Techniques

A significant contribution to computational geometry (both sequential and parallel) is the formulation of many geometric problems as searching problems in monotone matrices [8, 6]. Geometric problems amenable to such a formulation include the largest empty rectangle [10], various area minimization problems [8, 9, 206] (such as finding a minimum area circumscribing d-gon of a polygon), perimeter minimization [8] (finding a minimum perimeter triangle circumscribing a polygon), the layers of maxima problem [8, 35], rectilinear shortest paths in the presence of rectangular obstacles [29, 30], longest increasing chains [31, 35], approximation of certain geometric shapes used in VLSI design [77], and shortest paths in certain planar graphs with geometric structures [34, 204]. Many more problems are likely to be formulated as such matrix searching problems in the future. This section discusses the current best deterministic parallel solutions to these monotone matrix searching problems.

10.1 Row Minima

An important matrix searching technique for solving geometric problems was introduced by Aggarwal *et al.* in [6], where a linear time sequential solution was also given. The technique has myriads of applications to geometric and combinatorial problems [8, 6].

Suppose we have an $m \times n$ matrix A and we wish to compute the array θ_A such that, for every row index r $(1 \leq r \leq m)$, $\theta_A(r)$ is the smallest column index c that minimizes $A(r,c)$

(that is, among all the c's that minimize $A(r,c)$, $\theta_A(r)$ is the smallest). If θ_A satisfies the following *sorted* property:

$$\theta_A(r) \leq \theta_A(r+1),$$

and if for every submatrix A' of A, $\theta_{A'}$ also satisfies the sorted property, then matrix A is said to be *totally monotone* [8, 6].

Given a totally monotone matrix A, the problem of computing array θ_A is known as that of "computing the row minima" of A [8]. The best EREW PRAM algorithm for this problem runs in $O(\log n)$ time and n processors [46] (where $m = n$).

Theorem 10.1: *The row minima (that is, the array θ_A) of an $m \times n$ totally monotone matrix A can be computed in $O(\log m + \log n)$ time with $O(m + n)$ processors in the EREW PRAM model.*

Any improvement in these parallel complexity bounds will also imply corresponding improvements on the parallel complexities of many geometric applications of this problem. In fact [46] proves a somewhat stronger result: an implicit description of θ_A can be computed, within the same time bound as in the above theorem, by $O(n)$ processors. From this implicit description, a single processor can obtain any particular $\theta_A(r)$ value in $O(\log n)$ time. One important ingredient that is crucial to the overall scheme is an $O(\max(n, \log m))$ time algorithm using $\min(n, \log m)$ processors in the EREW PRAM model. The heart of the CREW method uses a new kind of sampling and pipelining, where samples are evenly spaced (and progressively finer) *clusters* of elements and where the "help" for computing the information at a node does not come only from its children (as it did in Cascading) but also from some of its subtree's leaves. This CREW algorithm is later improved into an EREW algorithm; this is achieved by (i) storing each leaf solution in a suitable parallel data structure, and (ii) re-defining the nature of the information stored at the internal nodes. A more specific overview follows.

Since the matrix A is understood, we henceforth use θ as a shorthand for θ_A. Let R denote the set of m row indices of A, C its n column indices. An *interval* of rows or columns is a non-empty set of *contiguous* (row or column) indices $[i,j] = \{i, i+1, \cdots, j\}$. We imagine row indices to lie on a horizontal line, so that a row is *to the left of* another row if and only if it has a smaller index (similarly "left of" is defined for columns). We say that interval I_1 is to the left of interval I_2, and I_2 is to the right of I_1, if the largest index of I_1 is smaller than the smallest index of I_2. Let I be a column interval, and let A_I be the $m \times |I|$ submatrix of A consisting of the columns of A in I. We use θ_I as a shorthand for θ_{A_I}. That is, if r is a row index, then $\theta_I(r)$ denotes the smallest $c \in I$ for which $A(r, c)$ is minimized. Note that $\theta_I(r)$ usually differs from $\theta(r)$, since we are minimizing only over I rather than C. Instead of storing θ_I directly, we shall instead store a function π_I which is an implicit description of θ_I.

Definition 1: *For any column interval I and any column index c, $\pi_I(c)$ is the row interval such that, for every row index r in that interval, we have $\theta_I(r) = c$; $\pi_I(c)$ is empty if no such r exists.*

Note that the monotonicity of A implies that, if $c_1 < c_2$, then $\pi_I(c_1)$ is to the left of $\pi_I(c_2)$.

Note that each $\pi_I(c)$ can be stored in $O(1)$ space, since we need only store the beginning and end of that row interval. We actually use π_I as an implicit description of θ_I. The advantage of doing so is that we use $O(|I|)$ storage instead of the $O(|R|)$ that would be needed for explicitly storing θ_I. The disadvantage is that, given a row index r, a processor needs to binary search in the π_I array for the position of row index r in order to determine $\theta_I(r)$. Had we stored directly θ_I, $\theta_I(r)$ would be readily available in constant time. From now on, we consider our problem to be that of computing the π_C array. Once we have π_C, it is easy to do a postprocessing computation that obtains (explicitly) θ from π_C with m processors: each column c gets assigned $|\pi_C(c)|$ processors which set $\theta(r) = c$ for every $r \in \pi_C(c)$. Therefore Theorem 10.1 would easily follow if we can establish the following.

Theorem 10.2: *π_C can be computed in $O(\log m + \log n)$ time and $O(n)$ processors in the EREW PRAM model.*

The following lemma is an important ingredient in the overall solution.

Lemma 10.3: *The π_C array can be computed in $O(\max(n, \log m))$ time with $\min(n, \log m)$ processors in the EREW PRAM model.*

The bounds of the above lemma might look unappealing at first sight, but their significance lies in the fact that m can be much larger than n. In fact the overall solution uses this lemma on problems of size $m \times (\log n)$.

Simple-minded approaches to proving the above lemma, like "use one processor to binary search for $\pi_C(c)$ in parallel for each $c \in C$" do not work, the difficulty being that we do not know which $\pi_C(c)$'s are empty. In fact, if we knew which $\pi_C(c)$'s are empty then we could easily achieve $O(\log m)$ time with n processors (by using the above-mentioned straightforward binary search—e.g., binary search for the right endpoint of $\pi_C(c)$ by doing $\log m$ comparisons involving the two columns c and c', where c' is the nearest column to the right of c having a nonempty $\pi_C(c')$). The proof of the above lemma is in fact quite elaborate (cf. [46] for its details).

In addition to using the above lemma, the overall CREW algorithm for this problem also uses sampling and iterative refinement methods that are reminiscent of cascading, and are similar to those used in [25] for solving in parallel the string editing problem. As in cascading, it is useful to think of the computation as progressing through the nodes of a tree T which we now proceed to define.

Partition the column indices into $n/\log n$ adjacent intervals $I_1, \cdots, I_{n/\log n}$ of size $\log n$ each. Call each such interval I_i a *fat column*. Imagine a complete binary tree T on top of these fat columns, and associate with each node v of this tree a *fat interval* $I(v)$ (i.e., an interval of fat columns) in the following way: The fat interval associated with a leaf is simply the fat column corresponding to it, and the fat interval associated with an internal node is the union of the two fat intervals of its children. Thus a node v at height h has a fat interval $I(v)$ consisting of $|I(v)| = 2^h$ fat columns. The storage representation we use for a fat interval $I(v)$ is a list containing the indices of the fat columns in it; we also call that list $I(v)$, in order to avoid introducing extra notation. For example, if v is the left child of the root, then the $I(v)$ array contains $(1, 2, \cdots, n/(2\log n))$. Observe that $\sum_{v \in T} |I(v)| = O(|T| \log |T|) = O((n/\log n) \log n) = O(n)$.

The ultimate goal is to compute $\pi_C(c)$ for every $c \in C$.

Let *leaf problem* I_i be the problem of computing $\pi_{I_i}(c)$ for all $c \in I_i$. Thus a leaf problem is a subproblem of size $m \times \log n$. From Lemma 10.3 it easily follows that the leaf problems can all be solved in $O(\log n + \log m)$ time by n processors. It remains to show that an additional $O(\log n + \log m)$ time with $O(n)$ processors is enough for obtaining π_C.

Definition 2: *Let $J(v)$ be the interval of original columns that belong to fat intervals in $I(v)$ (hence $|J(v)| = |I(v)| \cdot \log n$). For every $v \in T$, fat column $f \in I(v)$, and subset R' of R, let $\psi_v(R', f)$ be the interval in R' such that, for every r in that interval, $\theta_{J(v)}(r)$ is a* *column in fat column f. We use "$\psi_v(R', *)$" as a shorthand for "$\psi_v(R', f)$ for all $f \in I(v)$)".*

We henceforth focus on the computation of the $\psi_{root(T)}(R, *)$ array, where $root(T)$ is the root node of T. Once we have the array $\psi_{root(T)}(R, *)$, it is easy to compute the required π_C array within the prescribed complexity bounds: For each fat column f, we replace the $\psi_{root(T)}(R, f)$ row interval by its intersection with the row intervals in the π_{I_f} array (which are already available at the leaf I_f of T).

Lemma 10.4: *$\psi_v(R, *)$ for every $v \in T$ can be computed by a CREW PRAM in time $O(height(T) + \log m)$ and $O(n)$ processors, where $height(T)$ is the height of T ($= O(\log n)$).*

Since $\sum_{v \in T}(|I(v)| + \log n) = O(n)$, we have enough processors to assign $|I(v)| + \log n$ of them to each $v \in T$. The computation proceeds in $\log m + height(T) - 1$ *stages*, each of which takes constant time. Each $v \in T$ will compute $\psi_v(R', *)$ for progressively larger subsets R' of R, subsets R' that double in size from one stage to the next of the computation. We now state precisely what these subsets are. Let R_i denote the $(m/2^i)$-sample of R, so that $|R_i| = 2^i$.

At the t-th stage of the algorithm, a node v of height h in T will use its $|I(v)| + \log n$ processors to compute, in constant time, $\psi_v(R_{t-h}, *)$ if $h \le t \le h + \log m$. It does so with the help of information from $\psi_v(R_{t-1-h}, *)$, $\psi_{LeftChild(v)}(R_{t-h}, *)$, and $\psi_{RightChild(v)}(R_{t-h}, *)$, all of which are available from the previous stage $t - 1$ (note that $(t-1) - (h-1) = t - h$). If $t < h$ or $t > h + \log m$ then node v does nothing during stage t. Thus before stage h the node v lies "dormant", then at stage $t = h$ it first "wakes up" and computes $\psi_v(R_0, *)$, then at the next stage $t = h + 1$ it computes $\psi_v(R_1, *)$, etc. At stage $t = h + \log m$ it computes $\psi_v(R_{\log m}, *)$, after which it is done.

See [46] for the details of what information v stores and how it uses its $|I(v)| + \log n$ processors to perform stage t in constant time. In [46] a method is also given for avoiding the use of the "concurrent read" capability of the CREW PRAM, thus making the results hold in the EREW PRAM model.

10.2 Tube Minima

In what can be viewed as the three-dimensional version of the above row minima problem [8], one is given an $n_1 \times n_2 \times n_3$ matrix A and one wishes to compute, for every $1 \le i \le n_1$ and $1 \le j \le n_3$, the $n_1 \times n_3$ matrix θ_A such that $\theta_A(i, j)$ is the smallest index k that minimizes

$A(i,k,j)$ (that is, among all the k's that minimize $A(i,k,j)$, $\theta_A(i,j)$ is the smallest). The matrix A is such that θ_A satisfies the following *sorted* property:

$$\theta_A(i,j) \leq \theta_A(i,j+1),$$

$$\theta_A(i,j) \leq \theta_A(i+1,j).$$

Furthermore, for any submatrix A' of A, $\theta_{A'}$ also satisfies the sorted property.

Given such a matrix A, the problem of computing array θ_A is called by Aggarwal and Park [8] "computing the tube minima" of A. Many geometric applications of this problem are mentioned in [8]. There are many non-geometric applications to this problem as well. These include parallel string editing [25], constructing Huffmann codes in parallel [48], and other tree-construction problems. (In [48], the problem was given the name "multiplying two concave matrices".) The best CREW PRAM algorithms for this problem run in $O(\log n)$ time and $n^2/\log n$ processors [8, 25], and the best CRCW PRAM algorithm runs in $O(\log\log n)$ time and $n^2/\log\log n$ processors [27] (where $n = n_1 = n_2 = n_3$). Both the CREW and the CRCW bounds are easily seen to be optimal.

We sketch the CRCW solution, which is an example of rootish divide and conquer. A crucial ingredient in the solution is the judicious use of the "aspect ratio condition": intuitively, the idea is to allow the aspect ratio of the subproblems solved recursively to deteriorate, but not too much, and in a controlled fashion.

We sketch a version of the algorithm that has the right time complexity, but does too much work (hence uses too many processors).

The procedure is recursive, and requires that A be $\ell \times h \times \ell$ with $h \leq \ell^2$. We call this last condition the *aspect ratio* requirement; we assume it to be true initially, and we maintain it through the recursion (doing so without damaging the time complexity or the work complexity is, in fact, the main difficulty in this preliminary algorithm). The preliminary CRCW PRAM algorithm runs in $O(\log\log \ell)$ time and has work (that is, number of operations) complexity $O((\ell h + \ell^2)(\log \ell)^2)$.

Before describing the algorithm, we need a few definitions and a review of some properties.

Let X (resp., Y) be a subset of the row (resp., height) indices of A, and let Z be a contiguous interval of the column indices of A. The *problem induced by the triplet* (X, Z, Y) is that of finding $\theta_{A'}$ for the $|X| \times |Z| \times |Y|$ submatrix A' of A induced by X, Z and Y. That is, it consists of finding, for each pair u, v with $u \in X$ and $v \in Y$, the smallest index $k \in Z$ such that $A(u, k, v)$ is minimized. This k need not equal $\theta_A(u, v)$ since we are minimizing only over Z. However, the following property holds [8, 25]. Assume X (resp., Y) is a contiguous interval of row (resp., height) indices of A. Let x, z, y (resp., x', z', y') be the smallest (resp., largest) indices in X, Z, Y respectively. If $\theta_A(x, y) = z$ and $\theta_A(x', y') = z'$, then the solution to the triplet (X, Y, Z) gives the correct value of $\theta_A(u, v)$ for all $u \in X$ and $v \in Y$ (this follows from the sortedness of θ_A).

The first stage of the computation partitions the row indices of A into $\ell^{1/3}$ contiguous intervals $X_1, X_2, \cdots, X_{\ell^{1/3}}$ of size $\ell^{2/3}$ each. Similarly, the height indices of A are partitioned into $\ell^{1/3}$ contiguous intervals $Y_1, Y_2, \cdots, Y_{\ell^{1/3}}$ of size $\ell^{2/3}$ each. An *endpoint* of an interval X_i (resp., Y_i) is the largest or smallest index in it. For each pair v, w such that v is an endpoint of X_i and w is an endpoint of Y_j, we assign $h^{1+(1/6)}$ processors which compute, in constant time, the index $\theta_A(v, w)$. (Computing the minimum of h entries using $h^{1+(1/6)}$ processors is known to take constant time [211].) The total number of processors used in this step of the algorithm is $O(\ell^{1/3}\ell^{1/3}h^{1+(1/6)})$, which is $O(\ell h)$ because $h \leq \ell^2$.

Let x (resp., x') be the smallest (resp., largest) index in X_i, and let y (resp., y') be the smallest (resp., largest) index in Y_j. Let $Z_{i,j}$ be the interval $[a, b]$ where $a = \theta_A(x, y)$ and $b = \theta_A(x', y')$. In the sequel, when we want to define such a $Z_{i,j}$, we shall simply say "let $Z_{i,j}$ denote the interval of column indices of A defined by the set $\theta_A(v, w)$ such that $v \in X_i$ and $w \in Y_j$"; we do so for simplicity of expression, although it is an abuse of language (because $Z_{i,j}$ might include an index k that is not the $\theta_A(u, v)$ of any pair $u \in X_i, v \in Y_j$).

After the above stage of the computation we know the beginning and end of each such interval $Z_{i,j}$. As already observed, for any pair of indices u, v where $u \in X_i$ and $v \in Y_j$, we have $\theta_A(u, v) \in Z_{i,j}$. Thus it suffices to solve all of the subproblems defined by triplets $(X_i, Z_{i,j}, Y_j)$. However, some of these triplets might violate the aspect ratio condition because their $Z_{i,j}$ is too large (larger than $|X_i|^2 = \ell^{4/3}$): Each such troublesome triplet (we call it a *bad* triplet) will be further partitioned into $k_{i,j} = \lceil |Z_{i,j}|/\ell^{4/3} \rceil$ smaller subproblems, by partitioning $Z_{i,j}$ into $k_{i,j}$ pieces of size $\ell^{4/3}$ each (except that possibly the $k_{i,j}$-th piece might be smaller). Specifically, if $Z_{i,j}^{(k)}$ denotes the k-th piece from this partition of $Z_{i,j}$, then the k-th subproblem *spawned* by the bad triplet $(X_i, Z_{i,j}, Y_j)$ is $(X_i, Z_{i,j}^{(k)}, Y_j)$. Of course such a spawned subproblem $(X_i, Z_{i,j}^{(k)}, Y_j)$ no longer has the property that $\theta_A(u, v) \in Z_{i,j}^{(k)}$ for $u \in X_i$ and $v \in Y_j$. However, the answer returned by solving such an $(X_i, Z_{i,j}^{(k)}, Y_j)$ is not meaningless: We can obtain $\theta_A(u,v)$ for $u \in X_i$ and $v \in Y_j$ by choosing the best among the $k_{i,j}$ candidates returned by the $k_{i,j}$ subproblems $(X_i, Z_{i,j}^{(k)}, Y_j)$, $1 \le k \le k_{i,j}$. We are now ready to give the details of the second stage of the computation.

The second stage of the computation "fills in the blanks" by doing one parallel recursive call on a number of problems, defined as follows. In what follows, we describe these problems one at a time, but one should keep in mind that they are all solved in parallel. The first class of problems to be solved recursively are the good ones, those defined by triplets $(X_i, Z_{i,j}, Y_j)$ where $|Z_{i,j}| \le \ell^{4/3}$. The $Z_{i,j}$ of such a good problem is not large enough to violate the aspect ratio constraint (because it satisfies $|Z_{i,j}| \le |X_i|^2$). The second class of problems to be solved recursively are those spawned by the bad triplets $(X_i, Z_{i,j}, Y_j)$, namely subproblems $(X_i, Z_{i,j}^{(k)}, Y_j)$, $1 \le k \le k_{i,j}$. By definition, each such $(X_i, Z_{i,j}^{(k)}, Y_j)$ satisfies the aspect ratio requirement. When these recursive calls return, we need not do further work for the good triplets, but for the bad ones we only have the answers for the $k_{i,j}$ subproblems they spawned. We can use these $k_{i,j}$ subanswers to get the correct answers in constant time, however. For each bad triplet $(X_i, Z_{i,j}, Y_j)$, we need to compute, for every pair $u \in X_i$ and $v \in Y_j$, the minimum among $k_{i,j}$ entries. We do so by using $k_{i,j}h^{1/6}$ processors for each such pair $u \in X_i$ and $v \in Y_j$ (this is enough, since we are then computing the minimum of $k_{i,j}$ entries using $\ge k_{i,j}^{1+(1/6)}$ processors). Since there are $\ell^{4/3}$ such u, v pairs per bad triplet, the total work done for this "bad triplet postprocessing" is upper-bounded by $\ell^{4/3}h^{1/6}\sum_{i,j} k_{i,j}$; now, since

$$\sum_{i,j} k_{i,j} = \sum_{\beta=-\ell^{1/3}+1}^{\ell^{1/3}-1} \sum_i k_{i,i+\beta} \le \sum_{\beta=-\ell^{1/3}+1}^{\ell^{1/3}-1} (2h/\ell^{4/3}) = O(h/\ell),$$

this work is $O(\ell^{4/3}h^{1/6}h/\ell) = O(h\ell)$ (where the fact that $h \le \ell^2$ was used).

The bottom of the recursion is as usual: We stop when ℓ is some small enough constant (note that, by the aspect ratio condition, a constant ℓ implies a constant h, since $h \le \ell^2$).

Analysis of the above shows that the time complexity satisfies $T(\ell, h) = O(\log\log\ell)$, and the work complexity satisfies $W(\ell, h)O((\ell h + \ell^2)(\log\ell)^2)$.

The above algorithm does too much work. We refer the reader to [27] for a solution that uses the above algorithm to achieve an optimal amount of work without requiring any aspect ration condition.

10.3 Generalized Monotone Matrices

Aggarwal *et al.* [7] considered some generalizations of monotone matrices. An $m \times n$ array A is called *staircase monotone* if the following conditions hold:

1. Every entry of A is either a real number or ∞.

2. $A[i,j] = \infty$ implies that $A[i,l] = \infty$ for all l with $l > j$, and $A[k,j] = \infty$ for all k with $k > i$.

3. For any $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, $A[i,j] + A[k,l] \leq A[i,l] + A[k,j]$ if all the four entries involved are finite.

Clearly, monotone matrices are a special case of staircase monotone matrices. It was indicated in [7] that staircase monotone matrices, like monotone matrices, appear in numerous applications (see [7] for references).

Efficient parallel algorithms for computing row-maxima in $n \times n$ staircase monotone matrices were presented in [7]. In particular, the algorithms take $O(\log n)$ time and n processors on the CRCW PRAM, and $O(\log n \log\log n)$ time and $n/\log\log n$ processors on the CREW PRAM. Using these parallel row-maxima algorithms for two-dimensional staircase monotone matrices, several geometric problems were solved efficiently, such as the largest-area empty rectangle among points contained in a given rectangle, the nearest-visible or farthest-visible neighbors between two non-intersecting convex polygons, and other related problems [7].

11 Other Useful PRAM Techniques

There are other geometric techniques on the PRAM that we did not describe in detail, yet are also very useful. In particular, the following techniques are noteworthy.

11.1 Geometric Hierarchies

The paradigm of geometric hierarchies has proved extremely useful and general in computational geometry, both sequential [157, 116, 117, 118] and parallel [100, 99]. Generally speaking, the method consists of building a sequence of descriptions of the geometric objects under consideration, where an element of the sequence is simpler and smaller (by a constant factor) than its predecessor, and yet "close" enough that information about it can be used to obtain in constant time information about the predecessor. This "information" could be, for example, the location of a query point in the planar subdivision, assuming that the elements of the sequence are progressively simpler subdivisions of the plane (in that case, pointers exist between faces of a subdivision and those of its predecessor — these pointers are part of the data structure representing the sequence of subdivisions). This technique turns out to be useful for other models than the PRAM.

11.2 From CREW to EREW

In order to turn a CREW algorithm into an EREW one, one needs to get rid of the *read conflicts*, the simultaneous reading from the same memory cell by many processors. Such read conflicts often occur in the conquer stage, and can take the form of concurrent searching of a data structure by many processors (e.g., see [36, 68, 71]). To avoid read conflicts during such concurrent searching, the scheme of [190] can be helpful:

Lemma 11.1 (Paul, Vishkin, Wagener): *Suppose T is a 2-3 tree with m leaves, suppose a_1, a_2, ..., a_k are data items that may or may not be stored in (the leaves of) T, and suppose each processor P_j wants to search for a_j in T, $j = 1, 2, \ldots, k$. Then in $O(\log m + \log k)$ time, the k processors can perform their respective searches without read conflicts.*

Many types of searches can be accommodated by the above lemma. The following tends to occur in geometric applications:

- Type 1: Searching for a particular item in the tree, and
- Type 2: Searches of the type "find the t-th item starting from item p in the tree".

The search tree need not be a 2-3 tree: The requirements for the concurrent searching scheme of [190] to be applicable are that (i) each node of the tree has $O(1)$ children, and (ii) the k searches should be "sortable" according to their ranks in the sorted order of the leaves of the tree. (The scheme of [190] has other requirements, but they are needed only for the concurrent insertions and deletions that it can also handle, not for searching.) Requirement (i) is usually satisfied in geometric applications. Requirement (ii) is also clearly satisfied for the searches of Type 1. It can be made to be satisfied for searches of Type 2 by sorting the queries according to the leaf orders of their targets (this requires first doing a search of Type 1 to determine the leaf order of p).

11.3 Array-of-Trees

The array-of-trees data structure was originally developed in a non-geometric framework [47] for designing parallel algorithms, and was later generalized in [133, 140] for parallelizing a number of sweeping algorithms for geometric problems.

The array-of-trees data structure can be viewed as a parallel version of the persistence paradigm of Driscoll *et al.* [121]. Suppose a sequence S of n operations is to be performed in an on-line fashion, and S contains $m \leq n$ set-modifying operations (e.g., insert, delete, extract-minimum, etc). The idea in [47] is to maintain m relevant "snapshots" of a sequential data structure for evaluating S on-line, with $O(\log m)$ time per operation. Of course, storing explicitly the m copies of the sequential data structure (with one copy for each set-modifying operation) would be very expensive. Hence, a single "compressed" structure T for the m logical data structures is used, such that this structure can be built and processed in $O(\log m)$ time with m processors. Actually, the structure T can be considered as an array of m trees that share nodes, such that the t-th tree stores the information that is present right after the t-th set-modifying operation in S is performed (hence the name "array-of-trees"). T is represented as a complete binary tree with m leaves. A node of T, called a *super-node*, contains a number of *minor-nodes* that are nodes of the m individual trees represented by T.

The array-of-trees data structures were generalized in [133, 140], becoming a general technique for parallelizing sweeping algorithms for geometric problems. Note that the "sweeping" paradigm has been widely used in designing sequential geometric algorithms [124, 165, 193]. The array-of-trees based parallel methods in [133, 140] address those geometric problems for which the sweeping can be described either as a single sequence of data operations or as a related collection of operation sequences. These methods establish efficient parallel bounds for a number of geometric problems, such as computing the intersections of line segments in the plane [133], several types of hidden-surface elimination, CSG evaluation, and construction of the contour of a collection of rectangles [140].

11.4 Deterministic Geometric Sampling

A general type of geometric structure that has proven very useful in designing efficient randomized as well as deterministic geometric algorithms is the *geometric partition* (e.g., see [82, 84, 148]). The general framework is that given a collection X of n geometric objects in the d-dimensional space $\mathbf{R}^d$ (e.g., lines or hyperplanes) and a parameter r, one is to construct a partitioning of the space $\mathbf{R}^d$ into $O(r^d)$ small sized cells, such that each cell intersects as few objects of X as possible. This structure is particularly amenable to the recursive design of divide and conquer based geometric data structures and algorithms. As shown in [82, 84, 148], it is often possible to apply random sampling to construct a geometric partitioning of space, such that each cell intersects $O(\frac{n \log r}{r})$ objects on average. Further, Chazelle and Friedman [66] proved that such a partitioning can be obtained deterministically in polynomial time (through derandomization), and it was also shown that in certain situations even NC implementations are possible [51, 182]. Derandomization of geometric algorithms based on random samples is often achieved by quantifying the combinatorial properties used by the random samples and by showing that the sets having such combinatorial properties can be constructed efficiently in the deterministic fashion. Most of the combinatorial properties needed by geometric random samples can be described by the notions of the ϵ-approximation [173, 223] and the ϵ-net [148, 173].

Deterministic geometric sampling techniques have also been generalized by Amato, Goodrich, and Ramos to designing parallel geometric algorithms, achieving parallel bounds for a number of important problems that are more efficient than the previously known solutions [18, 19, 22, 135, 136, 138, 142, 196]. These include 3-dimensional convex hull [18, 136], d-dimensional convex hull for any fixed $d \geq 3$ [18], d-dimensional linear programming for any fixed $d \geq 2$ [138, 142], intersections of planar line segments [19], arrangements of lines and hyperplanes [135], lower envelope of constant degree algebraic functions of one variable [196], various Voronoi diagrams [22, 196], etc. More interestingly, the parallel geometric algorithms based on deterministic geometric sampling techniques often are also efficient in the sense of being *output-sensitive* (see Subsection 11.5 for more discussion of this type of parallel geometric algorithms).

11.5 Output-Sensitive Algorithms

Parallel output-sensitive algorithms are those whose complexity bounds depend on not only the size of the input but also the size of the output. For example, when computing the intersections of n line segments in the plane, the time and number of processors used by an algorithm may depend on n as well as the number of the actual intersections between the given line segments [19, 133, 202]. There have been quite a few interesting deterministic output-sensitive PRAM algorithms for geometric problems. Notable examples of such algorithms are 3-dimensional convex hull [18], intersections of planar line segments [19, 133, 202], lower envelope of constant degree algebraic functions of one variable [196], various Voronoi diagrams [22, 196], rectilinear hidden-surface elimination [140], contour construction for a collection of rectangles [140], construction of the visibility graph of a simple polygon [143], etc.

The above parallel output-sensitive geometric algorithms are in general based on the paradigm that the pool of virtual processors (in a slightly different PRAM model) can grow as the computation progresses, provided that the processor allocation happens globally [133, 202]. In this paradigm, r new processors are allowed to be allocated in time t only if an r-element array that stores pointers to the r tasks on which these new processors are to begin performing in time $t+1$ has been constructed. This PRAM model is essentially the same as the "standard" PRAM, with the exception that in the "standard" PRAM, processors are requested and allocated only once (at the beginning of the computation).

11.6 Stratified Decomposition Trees

The stratified decomposition tree was used in [143] for solving efficiently in parallel visibility and shortest path problems on a simple polygon. These polygon problems include shortest path queries, visibility from an edge, visibility graph, finding the convex rope, finding all-farthest neighbors, etc [143].

Given a triangulation of an n-vertex simple polygon P, let T be a *centroid decomposition tree* of the graph-theoretic planar dual of the triangulation (the centroid decomposition tree T can be constructed from the triangulation of P in $O(\log n)$ time and $n/\log n$ processors [91]). The tree T represents a decomposition of P into subpolygons. Let d be a chosen integer parameter. A *contraction* operation is first performed on T, such that for each node v of T, the subtree T_v of T rooted at v is "contracted" into a single node at v if and only if the subtree T_w of T rooted at every child w of v has less than d vertices but T_v has at least d vertices. Let T'' be the contracted tree obtained from T. Then mark every vertex v of T'' if v is at a level $i * d$ of T'' for $i = 0, 1, \ldots,$ or if v is a leaf of T''. Let the marked vertices form a "compressed" version $S(T'')$ of T'' based on their ancestor-descendant relation in T''. The tree $S(T'')$ so obtained is the stratified decomposition tree for P (the name "stratified decomposition tree" refers to the "layering" of the marked vertices in T''). By storing appropriate information about certain shortest paths in the subpolygons associated with the vertices of $S(T'')$, the stratified decomposition tree can be used for answering shortest path queries in P and for solving other problems [143]. By choosing $d = r\lceil \log^4 n \rceil$ for a certain constant r, the stratified decomposition tree can be built in $O(\log n)$ time using $n/\log n$ CREW PRAM processors [143].

11.7 Prune-and-Search

Prune-and-search techniques have been used by many sequential geometric algorithms [193]. In parallel, Cole [86] gave an efficient prune-and-search algorithm for solving the selection problem [57, 95] on the EREW PRAM. Cole's parallel selection algorithm takes $O(\log n \log^* n)$ time and $O(n)$ work. When the k-th smallest element is to be selected from an n-element set S, Cole's algorithm [86] performs the following main steps:

1. Use k to find in S two "good" approximations to the k-th smallest element with ranks k_1 and k_2 respectively, such that $k_1 \leq k \leq k_2$ and $k_2 - k_1$ is sufficiently small.

2. Eliminate from S all the elements whose ranks are not in the range of $[k_1, k_2]$, and recursively solve the problem on the remaining elements in S.

3. When $|S|$ is $O(n/\log n)$, use Cole's parallel merge sort [87] to finish the selection.

Cole's selection algorithm consists of $O(\log^* n)$ recursion levels, with $O(\log n)$ time per level.

A key feature of Cole's parallel selection algorithm is that the approximations are determined based on an already known rank k. But, this feature could limit the applicability of Cole's parallel prune-and-search technique. For certain problems such as two-variable linear programming [122, 174] and weighted selection [95], finding in parallel good approximations based on rank information as in [86] can be difficult, since useful rank information is not as readily available as for the selection problem.

Actually, an EREW PRAM algorithm for two-variable linear programming can be obtained by a straightforward parallelization of a sequential prune-and-search algorithm [122, 174]: Select the median of the candidate elements, eliminate a constant fraction of these elements, and recurse on the remaining elements. By using Cole's EREW PRAM algorithm for approximate median selection [86], one can simulate in parallel this sequential prune-and-search procedure, until the number of the remaining elements is $O(n/\log n)$ (at that point, one can simply apply a parallel algorithm for computing common intersection of half-planes such as [71]). Such a parallel prune-and-search algorithm for two-variable linear programming takes $O(\log n \log\log n)$ time and $O(n)$ work on the EREW PRAM. In fact, Deng [113] gave an $O(\log n)$ time, $O(n)$ work algorithm on the CRCW PRAM for two-variable linear programming that is based on such a parallelization of the sequential prune-and-search procedure.

A more general parallel prune-and-search technique was recently developed in [78]. This technique is different from Cole's [86] in that it does not depend on any given rank information. This new parallel prune-and-search technique has led to $O(\log n \log^* n)$ time, $O(n)$ work algorithms for two-variable linear programming and other problems on the EREW PRAM. This is an improvement over the EREW PRAM algorithms of [123, 142] when applied to the two-variable case (the algorithms in [123, 142] are based on more sophisticated parallel techniques). A key procedure used in the prune-and-search algorithms [78] is the parallel algorithm of Chen *et al.* for partitioning sorted sets [75].

12 Further Remarks

In addition to the open problems in parallel computational geometry that we already mentioned earlier, the following open problems are likely to receive attention in the future:

- $O(\log n)$ time, optimal-work deterministic construction on the CREW or EREW PRAM for Voronoi diagrams in the plane. The current best parallel bounds are $O(\log n \log\log n)$ time and $\frac{n \log n}{\log\log n}$ CREW PRAM processors [89], or $O(\log^2 n)$ time and $n/\log n$ EREW PRAM processors [22, 196].

- $O(\log n)$ time, optimal-work deterministic construction on the CREW or EREW PRAM for 3-dimensional convex hull. The current best parallel bounds are $O(\log n)$ time and $O(n^{1+\epsilon})$ work on the CREW PRAM [21] (for any constant $\epsilon > 0$), or $O(\log^2 n)$ time and $O(n \log n)$ work on the EREW PRAM [18, 136].

- $O(\log n)$ time, optimal-work deterministic EREW PRAM solutions for linear programming in any fixed dimension $d \geq 2$. The current best EREW PRAM bounds are $O(\log n \log^* n)$ time and $O(n)$ work for $d = 2$ [78], and $O(\log n (\log\log n)^{d-1})$ time and $O(n)$ work for any fixed $d \geq 3$ [142].

- $O(\log n)$ time, optimal-work deterministic EREW PRAM solution for the row minima in an $n \times n$ totally monotone matrix. The current best EREW PRAM bounds are $O(\log n)$ time and $O(n \log n)$ work [46] (and $O(\log n)$ time and $O(n)$ work, but a randomized solution [195]).

The following are additional promising directions for future research:

- Output-sensitive PRAM algorithms. Most geometric problems remain open when being looked at from the parallel output-sensitive perspective, that is, time and work optimal algorithms are yet to be found. This is the case with the known deterministic output-sensitive PRAM solutions [18, 19, 22, 133, 140, 143, 196, 202]. Recently, there are also several randomized output-sensitive PRAM algorithms [83, 127, 134, 146, 196].

- Robust parallel algorithms. Recall that robust algorithms are such that their correctness is not destroyed by roundoff errors. Most existing parallel geometric algorithms misbehave if implemented with rounded arithmetic. There has been recently an increasing interest in designing efficient and robust sequential algorithms for geometric problems (see [169] for a list of references), and it is natural for this activity to spread to the design of parallel geometric algorithms as well.

References

[1] K. Abrahamson, N. Dadoun, D. A. Kirkpatrick, and T. Przytycka, "A Simple Parallel Tree Contraction Algorithm," *J. of Algorithms*, Vol. 10, 1989, pp. 287–302.

[2] P. K. Agarwal, "A Deterministic Algorithm for Partitioning Arrangements of Lines and Its Applications," *Proc. 5th Annual ACM Symp. on Computational Geometry*, 1989, pp. 11–22.

[3] P. K. Agarwal, *Intersection and Decomposition Algorithms for Planar Arrangements*, Cambridge University Press, New York, 1991.

[4] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, Vol. 3, 1988, pp. 293–328.

[5] A. Aggarwal and M.-D. Huang, "Network Complexity of Sorting and Graph Problems and Simulating CRCW PRAMS by Interconnection Networks," *Lecture Notes in Computer Science Vol. 319: VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing*, Springer-Verlag, 1988, pp. 339–350.

[6] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric Applications of a Matrix Searching Algorithm," *Algorithmica*, Vol. 2, 1987, pp. 209–233.

[7] A. Aggarwal, D. Kravets, J. K. Park, and S. Sen, "Parallel Searching in Generalized Monge Arrays with Applications," *Proc. 2nd ACM Symp. Parallel Algorithms and Architectures*, 1990, pp. 259–268.

[8] A. Aggarwal and J. Park, "Parallel Searching in Multidimensional Monotone Arrays," *Proc. 29th Annual IEEE Symp. on Foundations of Computer Science*, 1988, pp. 497–512. (To appear in *J. of Algorithms*.)

[9] A. Aggarwal, B. Schieber and T. Tokuyama, "Finding a Minimum Weight k-Link Path in Graphs with Monge Property and Applications," *Proc. 9th Annual Symp. on Computational Geometry*, 1993, pp. 189–197.

[10] A. Aggarwal and S. Suri, "Fast Algorithms for Computing the Largest Empty Rectangle," *Proc. 3rd ACM Symp. on Computational Geometry*, 1987, pp. 278–290.

[11] A. Aggarwal and J. S. Vitter, “The Input/Output Complexity of Sorting and Related Problems,” *Communications of the ACM*, Vol. 31, 1988, pp. 1116–1127.

[12] M. Ajtai, J. Komlos, and E. Szemeredi, “Sorting in $c \log n$ Parallel Steps,” *Combinatorica*, Vol. 3, 1983, pp. 1–19.

[13] M. Ajtai and N. Megiddo, “A Deterministic Poly($\log\log N$)-Time N-Processor Algorithm for Linear Programming in Fixed Dimension,” *SIAM J. Comput.*, Vol. 25, 1996, pp. 1171-1195.

[14] S. G. Akl, “A Constant-Time Parallel Algorithm for Computing Convex Hulls,” *BIT*, Vol. 22, 1982, pp. 130–134.

[15] S. G. Akl and K. A. Lyons, *Parallel Computational Geometry*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[16] N. Alon and N. Megiddo, “Parallel Linear Programming in Fixed Dimension Almost Surely in Constant Time,” *Proc. 31st Annual IEEE Symp. on Foundation of Computer Science*, 1990, pp. 574–582.

[17] N. M. Amato, “Finding a Closest Visible Vertex Pair Between Two Polygons,” *Algorithmica*, Vol. 14, 1985, pp. 183–201.

[18] N. M. Amato, M. T. Goodrich, and E. A. Ramos, “Parallel Algorithms for Higher-Dimensional Convex Hulls,” *Proc. 35th IEEE Symp. on Foundations of Computer Science*, 1994, pp. 683–694.

[19] N. M. Amato, M. T. Goodrich, and E. A. Ramos, “Computing Faces in Segment and Simplex Arrangements,” *Proc. 27th ACM Symp. on Theory of Computing*, 1995, pp. 672–682.

[20] N. M. Amato and F. P. Preparata, “The Parallel 3D Convex-Hull Problem Revisited,” *Int. J. Computational Geometry & Applications*, Vol. 2, 1992, pp. 163–174.

[21] N. M. Amato and F. P. Preparata, “A Time-Optimal Parallel Algorithm for Three-Dimensional Convex Hulls,” *Algorithmica*, Vol. 14, 1995, pp. 169–182.

[22] N. M. Amato and E. A. Ramos, “On Computing Voronoi Diagrams by Divide-Prune-and-Conquer,” *Proc. 12th Annual ACM Symp. on Computational Geometry*, 1996, pp. 166–175.

[23] R. Anderson, P. Beame, E. Brisson, “Parallel Algorithms for Arrangements,” *Proc. 2nd ACM Symp. Parallel Algorithms and Architectures*, 1990, pp. 298–306.

[24] R. Anderson and G. L. Miller, “Deterministic Parallel List Ranking,” *Algorithmica*, Vol. 6, 1991, pp. 859–868.

[25] A. Apostolico, M. J. Atallah, L. Larmore, and H. S. McFaddin, “Efficient Parallel Algorithms for String Editing and Related Problems,” *SIAM J. Comput.*, Vol. 19, 1990, pp. 968–988.

[26] M. J. Atallah, “Parallel Techniques for Computational Geometry,” *Proceedings of the IEEE*, Vol. 80, 1992, pp. 1425–1448.

[27] M. J. Atallah, “A Faster Parallel Algorithm for a Matrix Searching Problem,” *Algorithmica*, Vol. 9, 1993, pp. 156–167.

[28] M. J. Atallah, P. Callahan, and M. T. Goodrich, “P-Complete Geometric Problems,” *Int. J. Computational Geometry & Applications*, Vol. 3, 1993, pp. 443–462.

[29] M. J. Atallah and D. Z. Chen, “Parallel Rectilinear Shortest Paths with Rectangular Obstacles,” *Computational Geometry: Theory and Applications*, Vol. 1, 1991, pp. 79–113.

[30] M. J. Atallah and D. Z. Chen, “On Parallel Rectilinear Obstacle-Avoiding Paths,” *Computational Geometry: Theory and Applications*, Vol. 3, 1993, pp. 307–313.

[31] M. J. Atallah and D. Z. Chen, “Computing the All-Pairs Longest Chains in the Plane,” *Int. J. Computational Geometry & Applications*, Vol. 5, 1995, pp. 257–271.

[32] M. J. Atallah and D. Z. Chen, "Optimal Parallel Hypercube Algorithms for Polygon Problems," *IEEE Trans. on Computers*, Vol. 44, 1995, pp. 914–922.

[33] M. J. Atallah and D. Z. Chen, "Parallel Computational Geometry," *Current and Future Trends in Parallel and Distributed Computing*, A. Zomaya (Ed.), International Thomson, 1996, pp. 162–197.

[34] M. J. Atallah, D. Z. Chen, and O. Daescu, "Efficient Parallel Algorithms for Planar *st*-Graphs," *Proc. 8th Annual Int. Symp. on Algorithms and Computation*, 1997, pp. 223–232.

[35] M. J. Atallah, D. Z. Chen, and K. S. Klenk, "Parallel Algorithms for Longest Increasing Chains in the Plane and Related Problems," *Proc. of the Ninth Canadian Conference on Computational Geometry*, 1997, pp. 59–64.

[36] M. J. Atallah, D. Z. Chen, and H. Wagener, "An Optimal Parallel Algorithm for the Visibility of a Simple Polygon from a Point," *J. ACM*, Vol. 38, 1991, pp. 516–533.

[37] M. J. Atallah, R. Cole, and M. T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM J. Comput.*, Vol. 18, 1989, pp. 499–532.

[38] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay, "Multisearch Techniques for Implementing Data Structures on a Mesh-Connected Computer," *J. of Parallel and Distributed Computing*, Vol. 20, 1994, pp. 1–13.

[39] M. J. Atallah and A. Fabri, "On the Multisearching Problem for Hypercubes," Technical Report 93-029, Dept. of Computer Sciences, Purdue University, 1993.

[40] M. J. Atallah, G. N. Frederickson, and S. R. Kosaraju, "Sorting with Efficient Use of Special-Purpose Sorters," *Information Processing Letters*, Vol. 27, 1988, pp. 13–15.

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

[42] M. J. Atallah and M. T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *J. of Parallel and Distributed Computing*, Vol. 3, 1986, pp. 492–507.

[43] M. J. Atallah and M. T. Goodrich, "Parallel Algorithms for Some Functions of Two Convex Polygons," *Algorithmica*, Vol. 3, 1988, pp. 535–548.

[44] M. J. Atallah and S. E. Hambrusch, "Solving Tree Problems on a Mesh-Connected Processor Array," *Information and Control*, Vol. 69, 1986, pp. 168–187.

[45] M. J. Atallah, S. E. Hambrusch, and L. E. TeWinkel, "Parallel Topological Sorting of Features in a Binary Image," *Algorithmica*, Vol. 6, 1991, pp. 762–769.

[46] M. J. Atallah and S. R. Kosaraju, "An Efficient Parallel Algorithm for the Row Minima of a Totally Monotone Matrix," *J. of Algorithms*, Vol. 13, 1992, pp. 394–413.

[47] M. J. Atallah, S. R. Kosaraju, and M. T. Goodrich, "Parallel Algorithms for Evaluating Sequences of Set Manipulation Operations," *J. ACM*, Vol. 41, 1994, pp. 1049–1088.

[48] M. J. Atallah, S. R. Kosaraju, L. Larmore, G. L. Miller, and S. Teng, "Constructing Trees in Parallel," *Proc. 1st Annual ACM Symp. on Parallel Algorithms and Architectures*, Santa Fe, New Mexico, 1989, pp. 421–431.

[49] M. J. Atallah and J.-J. Tsay, "On the Parallel-Decomposibility of Geometric Problems," *Algorithmica*, Vol. 8, 1992, pp. 209–231.

[50] R. Beigel and J. Gill, "Sorting n Objects with a k-Sorter," *IEEE Trans. on Computers*, Vol. 39, 1990, pp. 714–716.

[51] B. Berger, J. Rompel, and P. W. Shor, "Efficient NC Algorithms for Set Cover with Applications to Learning and Geometry," *Proc. 30th Annual IEEE Symp. Found. Comput. Sci.*, 1989, pp. 54–59.

[52] O. Berkman, B. Schieber, and U. Vishkin, "A Fast Parallel Algorithm for Finding the Convex Hull of a Sorted Point Set," *Int. J. Computational Geometry & Applications*, Vol. 6, 1996, pp. 231–242.

[53] P. Bertolazzi, C. Guerra, and S. Salza, "A Parallel Algorithm for the Visibility Problem from a Point," *J. of Parallel and Distributed Computing*, Vol. 9, 1990, pp. 11-14.

[54] G. Bilardi and A. Nicolau, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines," *SIAM J. Comput.*, Vol. 18, 1989, pp. 216–228.

[55] G. E. Blelloch, "Scans as Primitive Parallel Operations," *Proc. Int. Conf. on Parallel Processing*, 1987, pp. 355-362.

[56] G. E. Blelloch, "Scan Primitives and Parallel Vector Models," Ph.D. Thesis, Massachusetts Institute of Technology, 1988.

[57] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *J. of Computer and System Sciences*, Vol. 7, 1973, pp. 448–461.

[58] A. Borodin and J. E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *J. of Computer and System Sciences*, Vol. 30, 1985, pp. 130–145.

[59] L. Boxer and R. Miller, "Dynamic Computational Geometry on Meshes and Hypercubes," *J. of Supercomputing*, Vol. 3, 1989, pp. 161–191.

[60] R. P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM*, Vol. 21, 1974, pp. 201–206.

[61] P. B. Callahan, "Optimal Parallel All-Nearest-Neighbors Using the Well-Seated Pair Decomposition," *Proc. 34th Annual IEEE Sympos. Found. Comput. Sci.*, 1993, pp. 332–340.

[62] V. Chandru, S. K. Ghosh, A. Maheshwari, V. T. Rajan, and S. Saluja, "NC-Algorithms for Minimum Link Path and Related Problems, *J. of Algorithms*, Vol. 19, 1995, pp. 173–203.

[63] B. M. Chazelle, "On the Convex Layers of the Planar Set," *IEEE Trans. on Information Theory*, Vol. IT-31, 1985, pp. 509–517.

[64] B. M. Chazelle, "Computational Geometry on a Systolic Chip," *IEEE Trans. on Computers*, Vol. C-33, 1984, pp. 774–785.

[65] B. M. Chazelle, and D. P. Dobkin, "Intersection of Convex Objects in Two and Three Dimensions." *J. ACM*, Vol. 34, 1987, pp. 1–27.

[66] B. M. Chazelle and J. Friedman, "A Deterministic View of Random Sampling and Its Use in Geometry," *Combinatorica*, Vol. 10, 1990, pp. 229–249.

[67] B. M. Chazelle and L. J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica*, Vol. 1, 1986, pp. 133–162.

[68] D. Z. Chen, "Parallel Techniques for Paths, Visibility, and Related Problems," Ph.D. thesis, Technical Report 92-051, Department of Computer Sciences, Purdue University, July 1992.

[69] D. Z. Chen, "Determining Weak External Visibility of Polygons in Parallel," *Proc. 6th Canadian Conf. on Computational Geometry*, 1994, pp. 375–380.

[70] D. Z. Chen, "Optimal Hypercube Algorithms for Triangulating Classes of Polygons and Related Problems," *Proc. 7th Int. Conf. on Parallel and Distributed Computing Systems*, 1994, pp. 174–179.

[71] D. Z. Chen, "Efficient Geometric Algorithms on the EREW-PRAM," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, 1995, pp. 41–47.

[72] D. Z. Chen, "An Optimal Parallel Algorithm for Detecting Weak Visibility of a Simple Polygon," *Int. J. Computational Geometry & Applications*, Vol. 5, 1995, pp. 93–124.

[73] D. Z. Chen, "Optimally Computing the Shortest Weakly Visible Subedge of a Simple Polygon," *J. of Algorithms*, Vol. 20, 1996, pp. 459–478.

[74] D. Z. Chen, "Determining Weak Visibility of a Polygon from an Edge in Parallel," *Int. J. Computational Geometry & Applications*, to appear.

[75] D. Z. Chen, W. Chen, K. Wada, and K. Kawaguchi, "Parallel Algorithms for Partitioning Sorted Sets and Related Problems," *Proc. 4th Annual European Symp. on Algorithms*, 1996, pp. 234–245.

[76] D. Z. Chen and S. Guha, "Testing a Simple Polygon for Monotonicity Optimally in Parallel," *Information Processing Letters*, Vol. 47, 1993, pp. 325–331.

[77] D.Z. Chen and X. Hu, "Efficient Approximation Algorithms for Floorplan Area Minimization," *Proc. of 33rd ACM/IEEE Design Automation Conf.*, 1996, pp. 483–486.

[78] D. Z. Chen and J. Xu, "Two-Variable Linear Programming in Parallel," 1998, submitted.

[79] W. Chen, K. Wada, and K. Kawaguchi, "Parallel Robust Algorithms for Constructing Strongly Convex Hulls," *Proc. 12th Annual ACM Symp. on Computational Geometry*, 1996, pp. 133–140.

[80] W. Chen, K. Wada, K. Kawaguchi, and D. Z. Chen, "Finding the Convex Hull of Discs in Parallel," *Int. J. Computational Geometry & Applications*, to appear.

[81] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 1980.

[82] K. L. Clarkson, "New Applications of Random Sampling in Computational Geometry," *Discrete & Computational Geometry*, Vol. 2, 1987, pp. 195–222.

[83] K. L. Clarkson, R. Cole, and R. E. Tarjan, "Randomized Parallel Algorithms for Trapezoidal Diagrams," *Int. J. on Computational Geometry & Applications*, Vol. 2, 1992, pp. 117–133.

[84] K. L. Clarkson and P. W. Shor, "Applications of Random Sampling in Computational Geometry, II," *Discrete & Computational Geometry*, Vol. 4, 1989, pp. 387–421.

[85] E. Cohen, R. Miller, E.M. Sarraf, and Q.F. Stout, "Efficient Convexity and Domination Algorithms for Fine- and Medium-Grain Hypercube Computers," *Algorithmica*, Vol. 7, 1992, pp. 51–75.

[86] R. J. Cole, "An Optimally Efficient Selection Algorithm," *Information Processing Letters*, Vol. 26 1987/1988, pp. 295–299.

[87] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, Vol. 17, 1988, pp. 770–785.

[88] R. Cole and M.T. Goodrich, "Optimal Parallel Algorithms for Point-Set and Polygon Problems," *Algorithmica*, Vol. 7, 1992, pp. 3–23.

[89] R. Cole, M. T. Goodrich, and C. Ó'Dúnlaing, "A Nearly Optimal Deterministic Parallel Voronoi Diagram Algorithm," *Algorithmica*, Vol. 16, 1996, pp. 569–617.

[90] R. Cole and U. Vishkin, "Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems," *Proc. 27th Annual IEEE Symp. on Foundations of Comp. Sci.,* 1986, pp. 487–491.

[91] R. Cole and U. Vishkin, "The Accelerated Centroid Decomposition Technique for Optimal Parallel Tree Evaluation in Logarithmic Time," *Algorithmica*, Vol. 3, 1988, pp. 329–346.

[92] R. Cole and U. Vishkin, "Faster Optimal Parallel Prefix Sums and List Ranking," *Information and Control*, Vol. 81, 1989, pp. 334–352.

[93] R. Cole and O. Zajicek, "An Optimal Parallel Algorithm for Building a Data Structure for Planar Point Location," *J. of Parallel and Distributed Computing*, Vol. 8, 1990, pp. 280–285.

[94] S. Cook and C. Dwork, "Bounds on the Time for Parallel RAM's to Compute Simple Functions," *Proc. 14th ACM Annual Symp. on Theory of Computing,* 1982, pp. 231–233.

[95] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[96] R. Cypher and C. G. Plaxton, "Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers," *J. Computer and System Sciences*, Vol. 47, 1993, pp. 501-548.

[97] R. Cypher and J. L. C. Sanz, "Optimal Sorting on Feasible Parallel Computers," *Proc. International Conference on Parallel Processing*, 1988, pp. 339-350.

[98] J. Czyzowicz, I. Rival, and J. Urrutia, "Galleries, Light Matchings, and Visibility Graphs," *Proc. Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science Vol. 382*, Springer-Verlag, 1989, pp. 316–324.

[99] N. Dadoun, "Geometric Hierarchies and Parallel Subdivision Search," Ph.D. Thesis, U. of British Columbia, 1990.

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

[101] N. Dadoun and D. Kirkpatrick, "Parallel Construction of subdivision hierarchies," *J. of Computer and System Sciences*, Vol. 39, 1989, pp. 153–165.

[102] A. Datta, A. Maheshwari, and J.-R. Sack, "Optimal Parallel Algorithms for Direct Dominance Problems," *Nordic Journal on Computing*, Vol. 3, 1996, pp. 71–87.

[103] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwartzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 1997.

[104] F. Dehne, "Solving Visibility and Separability Problems on a Mesh of Processors," *The Visual Computer*, Vol. 3, 1988, pp. 356–370.

[105] F. Dehne, "Computing the Largest Empty Rectangle on One and Two Dimensional Processor Arrays," *J. of Parallel and Distributed Computing*, Vol. 9, 1990, pp. 63–68.

[106] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable Parallel Computational Geometry for Coarse Grained Multicomputers," *Int. J. on Computational Geometry & Applications*, Vol. 6, 1996, pp. 379–400.

[107] F. Dehne, A. Ferreira, and A. Rau-Chaplin, "Parallel Fractional Cascading on a Hypercube Multiprocessor," *Computational Geometry: Theory and Applications*, 1992, Vol. 2, pp. 141-167.

[108] F. Dehne, A.-L. Hassenklover, and J.-R. Sack, "Computing the Configuration Space for a Robot on a Mesh of Processors," *Parallel Computing*, Vol. 12, 1989, pp. 221–231.

[109] F. Dehne, A.-L. Hassenklover, J.-R. Sack, and N. Santoro, "Computational Geometry Algorithms for the Systolic Screen," *Algorithmica*, Vol. 6, 1991, pp. 734–761.

[110] F. Dehne and A. Rau-Chaplin, "Implementing Data Structures on a Hypercube Multiprocessor, with Applications in Parallel Computational Geometry," *J. Parallel and Distributed Computing*, Vol. 8, 1990, pp. 367–375.

[111] F. Dehne, J.-R. Sack, and I. Stojmenovic, "A Note on Determining the 3-Dimensional Convex Hull of a Set of Points on a Mesh of Processors," *Proc. Scandinavian Workshop on Algorithms and Theory*, 1988, pp. 154–162.

[112] F. Dehne and I. Stojmenovic, "An $O(\sqrt{n})$ Time Algorithm for the ECDF Searching Problem for Arbitrary Dimensions on a Mesh of Processors," *Information Processing Letters*, Vol. 28, 1988, pp. 67–70.

[113] X. Deng, "An Optimal Parallel Algorithm for Linear Programming in the Plane," *Information Processing Letters*, Vol. 35, 1990, pp. 213-217.

[114] A. Dessmark, A. Lingas, and A. Maheshwari, "Multi-List Ranking: Complexity and Applications," *Theoretical Computer Science*, Vol. 141, 1995, pp. 337–350.

[115] O. Devillers and A. Fabri, "Scalable Algorithms for Bichromatic Line Segment Intersection Problems on Coarse Grained Multicomputers," *Int. J. Computational Geometry & Applications*, Vol. 6, 1996, pp. 487–506.

[116] D. P. Dobkin and D. G. Kirkpatrick, "Fast Detection of Polyhedral Intersections," *Theoretical Computer Science*, Vol. 27, 1983, pp. 241–253.

[117] D. P. Dobkin and D. G. Kirkpatrick, "A Linear Time Algorithm for Determining the Separation of Convex Polyhedra," *J. of Algorithms,* Vol. 6, 1985, pp. 381–392.

[118] D. P. Dobkin and D. G. Kirkpatrick, "Determining the Separation of Preprocessed Polyhedra – A Unified Approach," *Proc. of the Int. Colloq. on Automata, Lang., and Programming,* 1990, pp. 154–165.

[119] D. P. Dobkin, R. J. Lipton, and S. Reiss, "Linear Programming Is Log-Space Hard for P," *Information Processing Letters,* Vol. 9, 1979, pp. 96–97.

[120] D. P Dobkin and S. Reiss, "The Complexity of Linear Programming," *Theoretical Computer Science*, Vol. 11, 1980, pp. 1–18.

[121] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making Data Structures Persistent," *Proc. 18th Annual ACM Symp. on Theory of Computing*, 1986, pp. 109–121.

[122] M. E. Dyer, "Linear Time Algorithms for Two- and Three-Variable Linear Programs," *SIAM J. Computing*, Vol. 13, 1984, pp. 31–45.

[123] M.E. Dyer, "A Parallel Algorithm for Linear Programming in Fixed Dimension," *Proc. 11th Annual Symp. on Computational Geometry*, 1995, pp. 345–349.

[124] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, NY, 1987.

[125] H. ElGindy and M. T. Goodrich, "Parallel Algorithms for Shortest Path Problems in Polygons," *The Visual Computer*, Vol. 3, 1988, pp. 371–378.

[126] G. N. Frederickson and D. B. Johnson, "The Complexity of Selection and Ranking in $X+Y$ and Matrices with Sorted Columns," *J. of Computer and System Sciences*, Vol. 24, 1982, pp. 197–208.

[127] M. Ghouse and M. T. Goodrich, "Fast Randomized Parallel Methods for Planar Convex Hull Construction," *Computational Geometry: Theory and Applications*, to appear.

[128] A. Gibbons and W. Rytter, "An Optimal Parallel Algorithm for Dynamic Expression Evaluation and Its Applications," *Proc. Symp. on Found. of Software Technology and Theoretical Comp. Sci.*, Springer-Verlag, 1986, pp. 453–469.

[129] L. M. Goldschlager, "The Monotone and Planar Circuit Value Problems Are Log Space Complete for P," *SIGACT News*, Vol. 9, 1977, pp. 25–29.

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

[131] M. T. Goodrich, "Finding the Convex Hull of a Sorted Point Set in Parallel," *Information Processing Letters*, Vol. 26, 1987, pp. 173–179.

[132] M. T. Goodrich, "Triangulating a Polygon in Parallel," *J. of Algorithms*, Vol. 10, 1989, pp. 327–351.

[133] M. T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," *SIAM J. Comput.*, Vol. 20, 1991, pp. 737–755.

[134] M. T. Goodrich, "Using Approximation Algorithms to Design Parallel Algorithms That May Ignore Processor Allocation," *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, 1991, pp. 711–722.

[135] M. T. Goodrich, "Constructing Arrangements Optimally in Parallel," *Discrete and Computational Geometry*, Vol. 9, 1993, pp. 371–385.

[136] M. T. Goodrich, "Geometric Partitioning Made Easier, Even in Parallel," *Proc. 9th Annual ACM Symp. Computational Geometry*, 1993, pp. 73–82.

[137] M. T. Goodrich, "Planar Separators and Parallel Polygon Triangulation," *J. Computer and System Sciences*, Vol. 51, 1995, pp. 374–389.

[138] M. T. Goodrich, "Fixed-Dimensional Parallel Linear Programming via Relative Epsilon-Approximations," *Proc. 7th Annual ACM-SIAM Symp. Discrete Algorithms*, 1996, pp. 132–141.

[139] M. T. Goodrich, "Parallel Computational Geometry," *CRC Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke (eds.), CRC Press, Inc., 1997, pp. 669–682.

[140] M. T. Goodrich, M. Ghouse, and J. Bright, "Sweep Methods for Parallel Computational Geometry," *Algorithmica*, Vol. 15, pp. 126–153.

[141] M. T. Goodrich, C. Ó'Dúnlaing, and C. Yap, "Computing the Voronoi Diagram of a Set of Line Segments in Parallel," *Algorithmica*, Vol. 9, 1993, pp. 128–141.

[142] M. T. Goodrich and E .A. Ramos, "Bounded-Independence Derandomization of Geometric Partitioning with Applications to Parallel Fixed-Dimensional Linear Programming," to appear in *Discrete and Computational Geometry*.

[143] M. T. Goodrich, S. B. Shauck, and S. Guha, "Parallel Method for Visibility and Shortest Path Problems in Simple Polygons," *Algorithmica*, Vol. 8, 1992, pp. 461–486.

[144] S. Guha, "Parallel Computation of Internal and External Farthest Neighbours in Simple Polygons," *Int. J. Computational Geometry & Applications*, Vol. 2, 1992, pp. 175–190.

[145] S. Guha, "Optimal Mesh Computer Algorithms for Simple Polygons," *Proc. 7th International Parallel Processing Symp.*, Newport Beach, California, April 1993, pp. 182–187.

[146] N. Gupta and S. Sen, "Faster Output-Sensitive Parallel Convex Hulls for $d \leq 3$: Optimal Sublogarithmic Algorithms for Small Outputs," *Proc. 12th Annual ACM Symp. Computational Geometry*, 1996, pp. 176–185.

[147] T. Hagerup and C. Rüb, "Optimal Merging and Sorting on the EREW-PRAM," *Inform. Processing Letters*, Vol. 33, 1989, pp. 181–185.

[148] D. Haussler and E. Welzl, "Epsilon-Nets and Simplex Range Queries," *Discrete & Computational Geometry*, Vol. 2, 1987, pp. 127–151.

[149] J. Hershberger, "Upper Envelope Onion Peeling," *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, Springer-Verlag, 1990, pp. 368–379.

[150] J. Hershberger, "Optimal Parallel Algorithms for Triangulated Simple Polygons," *Proc. 8th Annual ACM Symp. Computational Geometry*, 1992, pp. 33–42.

[151] J. A. Holey and O. H. Ibarra, "Triangulation in a Plane and 3-D Convex Hull on Mesh-Connected Arrays and Hypercubes," Tech. Rep., University of Minnesota, Dept. of Computer Science, 1990.

[152] J.-W. Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th Annual ACM Symp. on Theory of Computing*, 1981, pp. 326–333.

[153] J. Jájá, *An Introduction to Parallel Algorithms,* Addison Wesley, Reading, MA, 1992.

[154] C. S. Jeong and D. T. Lee, "Parallel Geometric Algorithms for a Mesh-Connected Computer," *Algorithmica*, Vol. 5, 1990, pp. 155–177.

[155] S. L. Johnsson, "Combining Parallel and Sequential Sorting on a Boolean n-Cube," *Proc. International Conf. on Parallel Processing*, 1984, pp. 444-448.

[156] R. M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," *Handbook of Theoretical Computer Science*, Edited by J. van Leeuwen, Volume 1, Elsevier Science Publishers, 1990.

[157] D. G. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM J. Comput.*, Vol. 12, 1983, pp. 28–35.

[158] S. R. Kosaraju and A. Delcher, "Optimal Parallel Evaluation of Tree-Structured Computations by Raking," *Lecture Notes in Computer Science Vol. 319: VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing*, Springer-Verlag, 1988, pp. 101–110.

[159] C. P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *IEEE Trans. on Computers*, Vol. C-34, 1985, pp. 965–968.

[160] C. P. Kruskal, L. Rudolph, and M. Snir, "A Complexity Theory of Efficient Parallel Algorithms," *Lecture Notes in Computer Science Vol. 317, Proc. 15th Coll. on Autom., Lang., and Prog.*, Springer-Verlag, 1988, pp. 333–346.

[161] V. Kumar and V. Sinch, "Scalability of Parallel Algorithms for the All-Pairs Shortest-Path Problem," *J. of Parallel and Distributed Computing*, Vol. 13, 1991, pp. 124–138.

[162] M. Kunde, "Optimal Sorting on Multidimensional Mesh-Connected Computers," *Proc. 4th Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science*, Springer, 1987, pp. 408–419.

[163] 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.

[164] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM*, Vol. 27, 1980, pp. 831–838.

[165] D. T. Lee and F. P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, Vol. C-33, 1984, pp. 872–1101.

[166] D. T. Lee, F. P. Preparata, C. S. Jeong, and A. L. Chow, "SIMD Parallel Convex Hull Algorithms," Tech. Rep. AC-91-02, Northwestern University, Dept. of Electrical Eng. and Computer Science, 1991.

[167] F. T. Leighton, *An Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[168] C. Levcopoulos, J. Katajainen, and A. Lingas, "An Optimal Expected Time Algorithm for Voronoi Diagrams," *Proc. 1st Scandinavian Workshop on Algorithm Theory*, Springer-Verlag, 1988.

[169] Z. Li and V. Milenkovic, "Constructing Strongly Convex Hulls Using Exact or Rounded Arithmetic," *Algorithmica*, Vol. 8, 1992, pp. 345–364.

[170] A. Lingas, A. Maheshwari, and J.-R. Sack, "Optimal Parallel Algorithms for Rectilinear Link Distance Problems," *Algorithmica*, Vol. 14, 1995, pp. 261–289.

[171] P. D. MacKenzie and Q. Stout, "Asymptotically Efficient Hypercube Algorithms for Computational Geometry," *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, 1990, pp. 8–11.

[172] J. M. Marberg and E. Gafni, "Sorting in Constant Number of Row and Column Phases on a Mesh," *Proc. 24th Annual Allerton Conf. on Communication, Control, and Computing*, Monticello, Illinois, 1986, pp. 603–612.

[173] J. Matoušek, "Epsilon-Nets and Computational Geometry, *New Trends in Discrete and Computational Geometry*, J. Pach (eds.), *Algorithms and Combinatorics*, Vol. 10, Springer-Verlag, 1993, pp. 69–89.

[174] N. Megiddo, "Linear Time Algorithms for Linear Programming in R^3 and Related Problems," *SIAM J. Computing*, Vol. 12, 1983, pp. 759–776.

[175] G. L. Miller and J. H. Reif, "Parallel Tree Contraction and Its Applications," *Proc. 26th IEEE Symp. on Foundations of Comp. Sci.,* 1985, pp. 478–489.

[176] R. Miller and S. E. Miller, "Convexity Algorithms for Digitized Pictures on an Intel iPSC Hypercube," *Supercomputer Journal*, Vol. 31, VI-3, 1989, pp. 45–53.

[177] R. Miller and Q. F. Stout, "Geometric Algorithms for Digitized Pictures on a Mesh-Connected Computer," *IEEE Trans. PAMI*, Vol. 7, 1985, pp. 216–228.

[178] R. Miller and Q. F. Stout, "Efficient Parallel Convex Hull Algorithms," *IEEE Trans. on Computers*, Vol. C-37, 1988, pp. 1605–1618.

[179] R. Miller and Q. F. Stout, "Mesh Computer Algorithms for Computational Geometry," *IEEE Trans. on Computers*, Vol. C-38, 1989, pp. 321–340.

[180] R. Miller and Q. F. Stout, *Parallel Algorithms for Regular Architectures,* The MIT Press, Cambridge, Massachusetts, 1991.

[181] S. Miyano, S. Shiraishi, and T. Shoudai, "A List of P-Complete Problems," Technical Report RIFIS-TR-CS-17, 1989, Kyushu University.

[182] R. Motwani, J. Naor, and M. Naor, "The Probabilistic Method Yields Deterministic Parallel Algorithms," *Proc. 30th Annual IEEE Symp. Found. Comput. Sci.*, 1989, pp. 8–13.

[183] H. Mueller, "Sorting Numbers Using Limited Systolic Coprocessors," *Information Processing Letters*, Vol. 24, 1987, pp. 351–354.

[184] K. Mulmuley, *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice Hall, New Jersey, 1994.

[185] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers," *IEEE Trans. on Computers*, Vol. 30, 1981, pp. 101–106.

[186] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, 1987.

[187] J. O'Rourke, "Computational Geometry," *Ann. Rev. Comp. Sci.*, Vol. 3, 1988, pp. 389–411.

[188] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1993.

[189] I. Parberry, *Parallel Complexity Theory*, Pitman, London, 1987.

[190] W. Paul, U. Vishkin, and H. Wagener, "Parallel Dictionaries on 2-3 Trees," *Proc. 10th Coll. on Autom., Lang., and Prog., Lecture Notes in computer Science Vol. 154*, Springer, Berlin, 1983, pp. 597–609.

[191] C. G. Plaxton, "Load Balance, Selection, and Sorting on the Hypercube," *Proc. 1st Annual ACM Symp. on Parallel Algorithms and Architectures*, 1989, pp. 64–73.

[192] C. G. Plaxton, "On the Network Complexity of Selection," *Proc. 30th Annual IEEE Symp. on Foundations of Computer Science*, 1989, pp. 396–401.

[193] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[194] F. P. Preparata and R. Tamassia, "Fully Dynamic Techniques for Point Location and Transitive Closure in Planar Structures," *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 1988, pp. 558–567.

[195] R. Raman and U. Vishkin, "Optimal Parallel Algorithms for Totally Monotone Matrix Searching," *Proc. 5th Annual ACM-SIAM Symp. on Discrete Algorithms*, 1994, pp. 613–621.

[196] E. A. Ramos, "Construction of 1-d Lower Envelopes and Applications," *Proc. 13th Annual ACM Symp. Computational Geometry*, 1997, pp. 57–66.

[197] J. H. Reif and S. Sen, "Optimal Randomized Parallel Algorithms for Computational Geometry," *Algorithmica*, Vol. 7, 1992, pp. 91–117.

[198] J. H. Reif and S. Sen, "Polling: A New Random Sampling Technique for Computational Geometry," *SIAM J. Comput.*, Vol. 21, 1992, pp. 466–485.

[199] J. H. Reif and S. Sen, "Randomized Algorithms for Binary Search and Load Balancing on Fixed Connection Networks with Geometric Applications," *SIAM J. Comput.*, Vol. 23, 1994, pp. 633–651.

[200] J. H. Reif and Q. F. Stout, manuscript.

[201] J. H. Reif and L. Valiant, "A Logarithmic Time Sort for Linear Size Networks," *J. ACM*, Vol. 34, 1987, pp. 60–76.

[202] C. Rüb, "Line-Segment Intersection Reporting in Parallel," *Algorithmica*, Vol. 8, 1992, pp. 119–144.

[203] K. W. Ryu and J. Jájá, "Efficient Algorithms for List Ranking and for Solving Graph Problems on the Hypercube," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, 1990, pp. 83–90.

[204] S. Sairam, R. Tamassia, and J. S. Vitter, "An Efficient Parallel Algorithm for Shortest Paths in Planar Layered Digraphs," *Algorithmica*, Vol. 14, 1995, pp. 322–339.

[205] J. L. C. Sanz and R. Cypher, "Data Reduction and Fast Routing: A Strategy for Efficient Algorithms for Message-Passing Parallel Computers," *Algorithmica*, Vol. 7, 1992, pp. 77–89.

[206] B. Schieber, "Computing a Minimum-Weight k-Link Path in Graphs with the Concave Monge Property," *Proc. 6th Annual ACM-SIAM Symp. on Discrete Algorithms*, 1995, pp. 405–411.

[207] C. P. Schnorr and A. Shamir, "An Optimal Sorting Algorithm for Mesh-Connected Computers," *Proc. 18th ACM Symp. on Theory on Computing,* 1986, pp. 255–261.

[208] S. Sen, "A Deterministic Poly($\log\log n$) Time Optimal CRCW PRAM Algorithm for Linear Programming in Fixed Dimension," Technical Report 95-08, Dept. of Computer Science, University of Newcastle, 1995.

[209] S. Sen, "Parallel Multidimensional Search Using Approximation Algorithms: With Applications to Linear-Programming and Related Problems," *Proc. 8th ACM Symp. Parallel Algorithms and Architectures*, 1996, pp. 251–260.

[210] M. Sharir and P. K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.

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

[212] I. Stojmenovic, manuscript, 1988.

[213] Q. F. Stout, "Constant-Time Geometry on PRAMs," *Proc. 1988 Int'l. Conf. on Parallel Computing*, Vol. III, IEEE, pp. 104-107.

[214] R. Tamassia and J. S. Vitter, "Parallel Transitive Closure and Point Location in Planar Structures," *SIAM J. Comput.*, Vol. 20, (1991), pp. 708-725.

[215] R. Tamassia and J. S. Vitter, "Optimal Cooperative Search in Fractional Cascaded Data Structures," *Algorithmica*, Vol. 15, 1996, pp. 154–171.

[216] R. E. Tarjan and U. Vishkin, "Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time," *SIAM J. Comput.*, Vol. 14, 1985, pp. 862–874.

[217] C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM*, Vol. 20, 1977, pp. 263–271.

[218] G. T. Toussaint, "Solving Geometric Problems with Rotating Calipers," *Proc. IEEE MELECON '83*, Athens, Greece, May 1983.

[219] J.-J. Tsay, "Optimal Medium-Grained Parallel Algorithms for Geometric Problems," Technical Report 942, Dept. of Computer Sciences, Purdue University, 1990.

[220] J.-J. Tsay, "Parallel Algorithms for Geometric Problems on Networks of Processors," *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, Dallas, Texas, Dec. 1993, pp. 200–207.

[221] P. Vaidya, personal communication.

[222] L. Valiant, "Parallelism in Comparison Problems," *SIAM J. Comput.*, Vol. 4, 1975, pp. 348–355.

[223] V. N. Vapnik and A. Y. Chervonenkis, "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities," *Theory Probab. Appl.*, Vol. 16, 1971, pp. 264–280.

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

[225] H. Wagener, "Optimal Parallel Hull Construction for Simple Polygons in $O(\log\log n)$ Time," *Proc. 33rd Annual IEEE Sympos. Found. Comput. Sci.*, 1992, pp. 593–599.

[226] D. E. Willard and Y. C. Wee, "Quasi-Valid Range Querying and Its Implications for Nearest Neighbor Problems," *Proc. 4th Annual ACM Symp. on Computational Geometry*, 1988, pp. 34–43.

[227] C. K. Yap, "Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map," *Algorithmica*, Vol. 3, 1988, pp. 279–288.