Parallel Techniques for Computational Geometry

MIKHAIL J. ATALLAH, SENIOR MEMBER, IEEE

Invited Paper

A survey of techniques for solving geometric problems in parallel is given, both for shared memory parallel machines and for networks of processors. Open problems are also discussed, as well as directions for future research.

I. Introduction

Many of the problems in computational geometry come from applications in pattern recognition, computer graphics, statistics, operations research, computer-aided design, robotics, etc. The problems which arise in these areas can come from real-time applications and hence need to be solved as fast as possible. For many of these problems, however, we are already at the limits of what can be achieved through sequential computation. Such sequential methods can be inadequate for situations in which the input consists of a large number of geometric objects. Thus, it is natural to study the kinds of speedups that can be achieved through parallel computing. As an indication of the importance of this research direction, we note that four of the 11 problems used as benchmark problems to evaluate parallel architectures for the DARPA Architecture Workshop Benchmark Study of 1986 were computational geometry problems.

Unfortunately, many of the techniques used to find efficient sequential algorithms for computational geometry problems do not translate well into a parallel setting. That is, while providing elegant paradigms for designing sequential algorithms, these techniques use methods which seem to be inherently sequential. Therefore, one needs to develop new paradigms for computational geometry, paradigms better suited for a parallel processing environment. This article is a survey of the main known algorithmic techniques for solving computational geometry problems efficiently in parallel. Since the focus is on general algorithmic techniques rather than on specific problems, no attempt is made to list all of the known parallel complexity bounds for geometric problems (there are too many of them).

The rest of the paper is organized as follows. Section II briefly reviews parallel models, Section III discusses basic subproblems that tend to arise in the solution of geometric problems on *any* parallel model, Section IV discusses PRAM techniques, Section V discusses techniques for mesh-connected arrays of processors, Section VI deals with the hybrid RAM/ARRAY model and its connection to I/O complexity, Section VII mentions some experimental work, and Section VIII concludes.

II. Parallel Models

This section briefly reviews the models of parallel computation for which parallel geometric algorithms have been designed.

A. PRAM Models

The PRAM (parallel random access machine) model of parallel computation is the shared-memory model where the processors operate synchronously. A *step* in a PRAM consists of each processor reading the content of a cell in the shared memory, writing something in a cell of the shared memory, or performing a computation within its own registers. Thus all communication is done via the shared memory. The PRAM comes in many flavors. 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

Manuscript received September 24, 1990; revised June 27, 1991. This work was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

The author is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 9204296.

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 so far been the main vehicle used to study the parallel algorithmics of geometric problems, and much of this survey (Section IV) will deal with PRAM techniques.

B. Networks of Processors

A network of processors is modeled as a graph where the nodes represent processors and the edges represent communication lines. All the network models we consider are synchronous, and a *step* of such a network of processors consists either of each processor communicating with a neighbor by sending/receiving the contents of a register (a *data movement step*), or of each processor performing a computation within its own registers (a *computation step*). We next briefly review some network models.

1) The Mesh: In a d-dimensional mesh of processors, the processors operate synchronously and are positioned on an $h_1 \times \cdots \times h_d$ grid, one processor per grid point. A processor is denoted by its position in the grid, a typical one being denoted by $(i_1, \cdots, i_d)$, where $1 \leq i_k \leq h_k$ for every $k \in \{1, \cdots, d\}$. Processors $(i_1, \cdots, i_d)$ and $(j_1, \cdots, j_d)$ are *neighbors* if and only if $|i_1 - j_1| + |i_2 - j_2| + \cdots + |i_d - j_d| = 1$. Note that a processor cannot have more than $2d$ neighbors (processors at the boundary have fewer). A processor has a fixed (i.e., $O(1)$) number of storage registers. Some researchers assume that a register can store up to $\log n$ bits, while others limit the size of a register to $O(1)$ bits: here we assume the former model.

2) The Hypercube: Every processor in the k-dimensional hypercube H is labeled as $b_0 b_1 \cdots b_{k-1}$, where $b_i \in \{0, 1\}$ for $0 \leq i \leq k-1$. A processor with label $b_0 b_1 \cdots b_{k-1}$ is connected to k processors, having labels $b_0 b_1 \cdots \bar{b}_s \cdots b_{k-1}$, for $0 \leq s \leq k-1$ (where $\bar{b}_s$ denotes the complement of b_s). An edge (v_1, v_2) of H is said to be of dimension s if v_1 and v_2 differ in bit position s, i.e., $v_1 = b_0 b_1 \cdots b_s \cdots b_{k-1}$ and $v_2 = b_0 b_1 \cdots \bar{b}_s \cdots b_{k-1}$.

3) Other Network Models: Some geometric algorithms were designed for a number of other networks, which we shall not cover in any detail. These include the tree of processors, the pyramid, and the mesh of trees. Although the general algorithmic techniques for solving geometric problems on these models can be quite similar to the techniques used for other models (such as the mesh), there are significant differences in the way processors communicate (these networks have smaller diameter than the mesh). Generally speaking, the tree of processors has been used more for parallel information storage and retrieval than for solving geometric problems. The pyramid has been used mostly for image processing applications. For the reader interested in learning more about these and other network models, see the forthcoming books by Leighton [94] and by Miller and Stout [98].

C. Hybrid Models

These are models consisting of more than one type of machine, and the main one for which geometric problems have been considered consists of a sequential computer to which a mesh is attached. We postpone the description of this model until Section VI.

Although geometric algorithms have been designed for all of the network models mentioned above, there are far fewer geometric algorithms for network models than for PRAM's. Furthermore, among the network models, more geometric algorithms have been designed for the mesh than for any of the other network models, perhaps because the parallel complexity of such basic operations as sorting and list ranking is well understood for the mesh. Other models, such as the hypercube, are just as important, but the complexity of the most basic operations on them is still open. For this reason, among all the network models, we shall focus on the mesh (in Section V) and on hybrid variants of the mesh (in Section VI). We also briefly discuss the connections between these and the I/O complexity of geometric problems (in Section VI).

III. Basic Subproblems

This section reviews some basic subproblems that are ubiquitous in the design of parallel geometric algorithms, no matter what parallel model is used. In many models the complexity of these basic subproblems is well understood, but for some models (such as the hypercube) the complexity of some of these (such as sorting and list ranking) is still open, and in such situations no final statement about the complexity of the most common geometric problems can be made until these issues are resolved (especially since many geometric problems are related to sorting). These basic operations are reviewed below.

A. Sorting, Merging

Sorting is probably the most frequently used subroutine in parallel geometric algorithms. Fortunately, for PRAM models and for the mesh, we know how to sort optimally: $O(\log n)$ time and n processors on the EREW-PRAM [41], [8], $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh [95], [105], [87]. The parallel complexity of sorting on the hypercube is not known (the current best hypercube bound is $O(\log n (\log \log n)^2)$ with n processors [49]). On the mesh, the complexity of merging is the same as that of sorting, but on the hypercube and PRAM it is easier than sorting [117], [127], [29], [79]; it is $O(\log n)$ time on an n-processor hypercube, and on the PRAM it is $O(\log \log n)$ time with n processors or, alternatively, $O(\log n)$ time with $n / \log n$ processors.

B. Parallel Prefix

Given an array A of n elements and an associative operation $+$, the parallel prefix problem is that of computing the array B of n elements such that $B(i) = \sum_{k=1}^{i} A(k)$. Parallel prefix can be solved in $O(\log n)$ time and $n / \log n$ processors on an EREW-PRAM [90], $O(\log n / \log \log n)$

time and $n \log\log n/\log n$ processors on a CRCW-PRAM [45], $O(\sqrt{n})$ time on the mesh (trivial), and in $O(\log n)$ time on an n-processor hypercube (trivial). Computing the smallest element in the A array is a special case of parallel prefix; for the CRCW model it can be done faster than general parallel prefix—in $O(\epsilon^{-1})$ 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 [117].

C. 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 the previous one, and most tree computations as well as many graph computations reduce, via the *Euler tour technique* [122], to solving that problem. EREW-PRAM algorithms that run in $O(\log n)$ time and $n/\log n$ processors are known [44], [10]. An $O(\sqrt{n})$ time mesh algorithm is also known [22]. Its complexity on the hypercube is still an open problem.

D. Tree Contraction

Given a (not necessarily balanced) rooted tree, the problem is to reduce it to a single node by a sequence of node removals, where a node v can be removed if it is not the root and either (i) it is a leaf or (ii) it has only one child. In case (ii) the removal of v is accomplished by "bypassing it," i.e., making v's child the child of v's parent. In a parallel setting, many nodes can be removed simultaneously so long as they are independent, in the sense that the parent of a node being removed cannot be removed at the same time. This problem is an abstraction of many other problems, including that of evaluating an arithmetic expression tree [102]. Many elegant optimal EREW-PRAM algorithms for it are known [1], [44], [71], [84], running in $O(\log n)$ time with $n/\log n$ processors. It is easy to implement these in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh by using the techniques in [22].

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 (they tend to be model-dependent).

IV. PRAM Techniques

The PRAM has been extensively used in theoretical studies as a uniform vehicle for designing parallel algorithms. The PRAM is generally considered to be a rather unrealistic model of parallel computation. However, although there are no PRAM's commercially available, algorithms designed for PRAM's 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 subsection) 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).

A. Inherently Sequential Geometric Problems

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 method 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 in 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, [108] or [83]. A proof establishing the P-completeness of a problem is viewed as strong evidence that the problem is "inherently sequential." 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 structure to allow membership in NC. Even the otherwise P-complete problem of linear programming [65], [66] is in NC when restricted to the plane. In the rest of this subsection 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. Each of the problems known to be P-complete involves a collection of line segments in the plane.

- *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 [67] and [110]), and it clearly has a polynomial time sequential solution.
- *Weighted planar partitioning:* One is given a collection of n nonintersecting 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 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" [48], [106], and is P-complete even if there are only three possible orientations for the line segments. It is straightforward to solve it sequentially in $O(n \log^2 n)$ time (using the dynamic point-location data structure of [111]), and in $O(n \log \log n)$ time by a more sophisticated method [48].

- *Visibility layers*: One is give a collection of n nonintersecting 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 $(0, +\infty)$, 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 [35], [92], [107], and is P-complete even if all the segments are horizontal.

The P-completeness proofs of the above problems were given in [13]; for the third problem see also [78]. The proofs consist in giving NC reductions from the monotone circuit value problem and planar circuit value problem, which are known to be P-complete [70], [88], [103]. These reductions typically involve the use of geometry to simulate a circuit, by using the relative positions of objects in the plane.

Perhaps the most famous open problem in the area of geometric P-completeness is that of the convex layers problem [35]: given n points in the plane, mark the points on their convex hull as being layer zero, then remove layer zero and repeat the process, generating layers $1, 2, \cdots$, etc. In view of the P-completeness of the above-mentioned visibility layers problem, it is reasonable to conjecture that the convex layers problem is also P-complete.

B. "Fast" and "Efficient"

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 should match the lower bound (assuming such a lower bound is known), and the *time × processors* product should match the sequential time complexity of the problem. A parallel lower bound for a geometric problem is usually established by showing that it can be used to solve some other (perhaps nongeometric) problem having that lower bound. For example, it is well known [47] 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 on 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 [20], [128].

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 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 "nonparallelizable." One can indeed argue [126] that 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.

C. 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 (these include the convex hull problem [20], [128]). The time and processor complexities 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 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 two approaches often work.

D. "Rootish" Divide and Conquer

By "rootish" we mean partitioning the problem into $n^{1/k}$ subproblems to be solved recursively in parallel, for some 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 constants. These imply that $T(n) = O(\log n)$ and $P(n) = n$.

The problems that can be solved using rootish divide and conquer include the convex hull [2], [19], the visibility of nonintersecting planar segments from a point [28], and the visibility of a polygonal chain from a point [15]. The scheme is useful in various ways and forms, and sometimes with recurrences very different from the above-mentioned ones. For example, it was used in the form of a fourth-root divide and conquer to obtain (in a rather involved way) an optimal EREW algorithm for the visibility of a simple polygon from a point [15] (that is, $O(\log n)$ time with $n/\log n$ processors).

There are instances where 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. For example, in [15], the recursive procedure takes two parameters (one of which is problem size) and uses either fourth-root divide and conquer or two-way divide and conquer, depending on the relative sizes of these two input parameters.

E. Cascading

This sampling and iterative refinement method was introduced by Cole [41] for the sorting problem, and was further developed in [16] and [72] for the solution of geometric problems. It has proved to be a fundamental technique, one that allows optimal solutions when most other techniques fail. Its details are intricate even for sorting, but the gist of it can easily be described. Since the 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 the processing of 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 where 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, let $L(v)$ 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 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 our purpose was only to give the general idea of cascading.

The cascading technique has been used to solve many problems (not just geometric ones). Some of the geometric problems for which it has been used are as follows:

- *Fractional cascading*: Given a directed graph G in which every node v contains a sorted list $C(v)$, construct a linear space data structure (that is, one whose size is at most a constant factor larger than the space taken by the input) that enables one processor to quickly locate any x in all the sorted lists stored along a given path $(v_1, v_2, \cdots, v_k)$ in G (by "quickly" we mean in $O(\log |C(v_1)| + k)$ time). This problem was introduced by Chazelle and Guibas [37], who gave an elegant optimal sequential algorithm. An optimal $O(\log n)$ time and $n/\log n$ processor parallel algorithm for this problem was given in [16].
- *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 [16]. This implies similar bounds for the polygon triangulation problem [72], [74], [130].
- *Topological sorting of n nonintersecting line segments*: This is the problem of ordering the segments so that, if a vertical line l intersect 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 agorithm of [23] (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, 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 [16], [121], [46]. The planar point location problem itself tends to arise rather frequently, even in geometric problems apparently unrelated to it.
- *Intersection detection, three-dimensional maxima, two-set dominance counting, visibility from a point, all nearest neighbors*: For all of these problems, cascading can be used to achieve $O(\log n)$ time with n processors [16], [42].

Alternative approaches to cascading have been proposed for some of the above problems; for example, see [28], [113], [129] and also the elegant parallel hiererchical approach of Dadoun and Kirkpatrick, which is discussed next.

F. Geometric Hierarchies

This paradigm has proved extremely useful and general in computational geometry, both sequential [85], [62], [63], [64] and parallel [51], [52]. Generally speaking, the method consists in building a sequence of descriptions of the geometric object 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 subdivision, assuming 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.) The technique turns out to be useful for other models than the PRAM (see subsection V-B).

G. Brent's Theorem

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

Theorem 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 [34] before one can apply it to a PRAM: (i) at the beginning of the ith 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.

H. 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 (see, e.g., [15]). To avoid read conflicts during such concurrent searching, the scheme of [109] can be helpful:

Lemma 1 (Paul, Vishkin, Wagener [109]) Suppose T is a 2–3 tree with m leaves, suppose $a_1, a_2, \cdots, a_k$ 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, $j = 1, 2, \cdots, k$. 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 tend to occur in geometric applications:

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

The search tree need not be a 2–3 tree: the requirements for the concurrent searching scheme of [109] to be applicable are that (i) each node of the tree have $O(1)$ children and (ii) the k searches be "sortable" according to their ranks in the sorted order of the leaves of the tree. (The scheme of [109] 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).

I. Matrix Searching Techniques

A significant contribution to computational geometry (both sequential and parallel) is the formulation of many of its problems as searching problems in monotone matrices [5], [3]. Geometric problems amenable to such a formulation include the largest empty rectangle [6], various area minimization problems [5] (such as finding a minimum area circumscribing d-gon of a polygon), perimeter minimization [5] (finding a minimum perimeter triangle circumscribing a polygon), the layers of maxima problem [5], and rectilinear shortest paths in the presence of rectangular obstacles [14]. Many more problems are likely to be formulated as such matrix searching problems in the future. We briefly review these matrix searching formulations next.

1) Row Minima: An important matrix searching technique for solving geometric problems was introduced by

Aggarwal *et al.* in [3], where a linear time sequential solution was also given. The technique, which we review next, has myriads of applications to geometric and combinatorial problems [5], [3].

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 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 [5], [3].

Given a totally monotone matrix A, the problem of computing the θ_A array is known as that of computing the row minima of that matrix [5]. The best EREW-PRAM algorithm for this problem runs in $O(\log n)$ time and n processors [24] (where $m = n$). Any improvement in this parallel complexity bound will also imply corresponding improvements on the parallel complexities of the many geometric applications of this problem.

2) Tube Minima: In what can be viewed as the three-dimensional version of the above row minima problem [5], one is given an $n_1 \times n_2 \times n_3$ matrix A and one wishes to compute, for every $1 \leq i \leq n_1$ and $1 \leq j \leq n_3$, 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 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 the θ_A array is called by Aggarwal and Park [5] computing the tube minima of that matrix. Many geometric applications of this problem are mentioned in [5]. There are many nongeometric applications to this problem as well. These include parallel string editing [11], constructing Huffmann codes in parallel [25], and other tree-construction problems. (In [25] the problem was referred to as multiplying two concave matrices.) The best CREW-PRAM algorithms for this problem run in $O(\log n)$ time and $n^2/\log n$ processors [5], [11], and the best CRCW-PRAM algorithm runs in $O(\log\log n)$ time and $n^2/\log\log n$ processors [12] (where $n = n_1 = n_2 = n_3$). Both the CREW and the CRCW bounds are easily seen to be optimal.

J. Randomization

Reif and Sen [113]–[115] have modified and applied to parallel geometric computation the randomization techniques that had proved their worth in sequential geometric computing (cf. the works of K. Clarkson, Haussler and Welzl, and Mulmuley) as well as in areas other than computational geometry. Recall that a randomized algorithm is one which bases some of its decisions on the outcomes of coin flips. Thus for a particular input, there are many possible executions of a randomized algorithm (which one actually happens depends on the outcomes of the coin flips). A good randomized algorithm must ensure that the number of "bad" possible executions (e.g., those that take too long to terminate) is a small fraction of all the possible executions. Algorithms that are not randomized are *deterministic* (although this adjective is usually omitted when the context does not leave room for confusion). Some deterministic algorithms (such as the two-dimensional parallel Voronoi diagram algorithm given in [91]) have efficient expected time behavior for a randomly chosen set of input points, whereas randomized algorithms make no assumption about the input distribution.

Randomized algorithms have the disadvantage that they might fail, but if the probability of failure is made small enough then they can have advantages over deterministic ones: they are typically very simple (which makes them easy to program and to comprehend), and the multiplicative constant in their time complexity is usually small. For example, the algorithms given by Reif and Sen in [113] have a running time of $O(\log n)$ with n processors, *with high probability* (i.e., a probability that approaches 1 for very large n). The problems they deal with include planar point location and trapezoidal decomposition. The techniques they use there (and also in [115]) are somewhat reminiscent of the Flashsort algorithm of Reif and Valiant [116]. The *polling* technique of Reif and Sen [114] has yielded optimal randomized parallel bounds for two problems that continue to frustrate deterministic approaches, namely, the problems of computing the three-dimensional convex hull and the two-dimensional Voronoi diagram.

K. Other PRAM Techniques

There are other techniques that we did not describe in detail because of their somewhat specialized nature. One such technique is the "array of trees" parallel data structure, originally designed in a nongeometric framework [21] but later used in [76] to establish geometric parallel bounds for such problems as hidden-line elimination, CSG evaluation, and computing the contour of a collection of rectangles. Another technique is the "stratified decomposition tree," used in [77] in the parallel solution of visibility and path problems in polygons.

V. Mesh Techniques

In this section, for convenience, we limit the discussion to two-dimensional (i.e., $\sqrt{n} \times \sqrt{n}$) meshes, but most of the results and techniques mentioned are known to easily generalize to higher dimensional meshes as well. The geometric objects under consideration (e.g., points) are initially stored in the mesh, one object per processor. Therefore we are implicitly assuming that the mesh has enough processors to store the problem description—the important case where the problem size is too large to fit in the mesh is discussed in the next section.

Since it is known how to sort n items optimally (i.e., in $O(\sqrt{n})$ time) on a $\sqrt{n} \times \sqrt{n}$ mesh, sorting is not a bottleneck when trying to design $O(\sqrt{n})$ time solutions to geometric problems on the mesh. (Contrast this with the situation for the hypercube, a network in which the complexity of sorting is still unknown.) In fact many of the classical problems of computational geometry have been shown to be solvable on the mesh within the optimal $O(\sqrt{n})$ time bound (we mention some of these later). Most of these problems have an $O(n \log n)$ sequential time complexity, and since the mesh $time \times processors$ product is proportional to $n\sqrt{n}$, one might think that the word "optimal" is being abused here. However, this is not the case: any nontrivial problem on a $\sqrt{n} \times \sqrt{n}$ mesh requires $\Omega(\sqrt{n})$ time (since it can take that long for two processors to communicate), and there is usually no hope of using $o(n)$ processors because of the already mentioned $O(1)$ storage limitation per processor. (It takes $\Omega(n)$ space, and hence $\Omega(n)$ processors, just to store the input.)

A. Mesh Divide and Conquer

Many geometric algorithms on the mesh use some form of divide-and-conquer: the problem gets partitioned into (e.g.) four pieces of size $n/4$ each; each piece is then moved into one of the four quadrants of the mesh where it is solved recursively by the $(\sqrt{n}/2) \times (\sqrt{n}/2)$ quadrant, after which the answers returned by the four recursive calls are combined to obtain the overall solution. The "conquer" stage as well as the various bookkeeping steps usually involves sorting and takes $O(\sqrt{n})$ time. Thus the time recurrence of this scheme generally ends up being of the form

$$T(n) = T(n/4) + c\sqrt{n},$$

where c is a constant, which implies $T(n) = O(\sqrt{n})$. An example of this is the convex hull algorithm of [100]:

1) If n is small (say, $n \leq 4$) then solve the problem directly by brute force; otherwise proceed to step 2 below.
2) Sort the n points whose convex hull we seek by x coordinates. Put those with the smallest $n/4$ x coordinates in one of the four quadrants, those with the next $n/4$ smallest x coordinates into another quadrant, etc. In fact, the sorting itself can be done so that each quadrant automatically contains the appropriate $n/4$ points; i.e., no separate data movement is needed other than sorting (see [100] for details).
3) Recursively solve the problem for each of the four quadrants.
4) Combine the solutions returned by the four recursive calls into the hull of the whole point set. This involves finding the common tangents betweens pairs of disjoint convex polygons in $O(\sqrt{n})$ time.

The nontrivial part is usually the "combine" part (i.e., step 4 in the above example). The data movement techniques of [105] often play a role in that stage, and sometimes the tree computation technique of [22] is needed (e.g., in [82] and [23]).

B. Multisearching

The following problem is often the bottleneck in the parallel solution of geometric problems on a network of processors. It is a generalization of the problem described in Subsection IV-H: given a search structure modeled as a graph G with n constant-degree nodes, and given $O(n)$ search processes on that structure, the *multisearch* problem is that of performing as fast as possible all of the search processes on that structure. The searches need not be processed in any particular order, and can simultaneously be processed in parallel by using, for example, one processor for each. However, the path that a search query will trace in G is *not* known ahead of time, and must instead be determined "on-line": only when a search query is at (say) node v of G can it determine which node of G it should visit next (it does so by comparing its own search key with the information stored at v—the nature of this information and of the comparison performed depend on the specific problem being solved).

The multisearch problem is a useful abstraction that can be used to solve many problems (more on this later). It is a challenging problem both for EREW-PRAM's and for networks of processors, since many searches might want to visit a single node of G, creating a "congestion" problem (with the added complication that one cannot even tally ahead of time how much congestion will occur at a node, since one does not know ahead of time the full search paths, only the nodes of G at which they start). When the parallel model used to solve the problem is a network of processors, the graph G is initially stored in the network in the natural way, with each processor containing one node of G and that node's adjacency list. It is important to keep in mind that the computational network's topology is *not* the same as the search structure G, so that a neighbor of node v in G need not be stored in a processor adjacent to the one containing v. Each processor also contains initially (at most) one of the search queries to be processed (in which case that search does not necessarily start at the node of G stored in that processor).

In the EREW-PRAM, the difficulty comes from the "exclusive read" restriction of the model: if k processes were to simultaneously access node v's information, the k processors assigned to these k search processes would apparently be unable to simultaneously access v's information. We have already mentioned, in Lemma 1, an elegant way around this problem, designed by Paul, Vishkin, and Wagener [109] for the case where G is a 2–3 tree (although they assume a linear ordering on the search keys, something which usually does not hold in a geometric framework involving multidimensional search keys).

The multisearch problem is even more challenging for networks of processors. In such models, data are not stored in a shared memory, but are distributed over a network and require considerable time to be permuted to allow different processors access to different data items. Furthermore, each memory location can be accessed by only $O(1)$ query processes at a time, since a processor containing (say) node v's information would be unable to simultaneously store more than a constant number of search queries.

In [125] the multisearch problem is solved in $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh-connected computer, where r is the length of the longest search path associated with a query. For most geometric data structures, the search path traversed when answering a query has length $r = O(\log n)$; hence the time complexity is $O(\sqrt{n})$ time, which is asymptotically optimal. The classes of graphs for which this result holds contain most of the important cases of G that arise in practice, ranging from simple trees to the powerful Kirkpatrick hierarchical search DAG [85], which is so important in both sequential and parallel computational geometry (see subsection IV-F). Applications include interval trees and the related multiple interval intersection search, as well as hierarchical representations of polyhedra and its many applications, among them lines-polyhedron intersection queries, multiple tangent plane determination, three-dimensional convex hull, and intersecting convex polyhedra.

A special case of the multisearching problem for hypercube multiprocessors was studied in [59].

C. Mesh Prune and Search

Some mesh algorithms use the parallel equivalent of what has been called, in sequential computation, the prune and search paradigm [92]. This paradigm consists in throwing away a subset of the input (after determining that it does not contribute to the answer) and then recursively searching the surviving portion of the input. The portion of the input thrown away is a fixed fraction of the input (i.e., a subset of size cn of an input of size n, where c is a positive constant). Mesh implementations of this idea have the intriguing feature of advantageously keeping many of the processors idle during much of the computation. This is because, after doing the "pruning" (= decreasing problem size by a constant factor), the resulting (smaller) problem is compressed into a smaller submesh of the original mesh, where it is recursively solved while the processors not in this smaller submesh remain idle.

Mesh algorithms might involve a sequence of many recursive calls (occurring after one another rather than in parallel) and still run in $O(\sqrt{n})$ time. So long as these successive recursive calls are on problems of sizes $c_1n, c_2n, \cdots, c_kn$ where k as well as the c_i's are constants and $\sqrt{c_1} + \sqrt{c_2} + \cdots + \sqrt{c_k} < 1$, the time complexity is $O(\sqrt{n})$ (assuming that setting up each recursive call is done in $O(\sqrt{n})$ time, and that the other bookkeeping and combining of subsolutions also takes $O(\sqrt{n})$ time). As an example, see the algorithm in [100] for computing the closest pair among a set of n planar points.

The parallel version of the prune and search technique has been far more useful for the mesh than for the PRAM, because in the mesh we can afford to prune in $O(\sqrt{n})$ time and still end up with an optimal algorithm, whereas in the PRAM model the technique typically yields superlogarithmic time bounds ([61] being one of the few instances where it was used for a PRAM geometric algorithm, and that was for the CRCW model).

D. Some Known Bounds and Open Problems for the Mesh

We now mention some problems for which $O(\sqrt{n})$ time mesh algorithms are known, as well as some open problems for the mesh.

The following problems have known $O(\sqrt{n})$ time solutions on the mesh (the list is not exhaustive).

- Convex hull and all nearest neighbor problems for planar point sets [100].
- Voronoi diagram of a planar set of n points [82]. This remained an open problem for a while, until Jeong and Lee gave their elegant algorithm achieving an optimal time bound.
- Minimum distance spanning tree for planar point sets. This follows from the above-mentioned Voronoi diagram result of Jeong and Lee, and the fact that a minimum spanning tree of an e-edge undirected graph can be computed in $O(\sqrt{e})$ time on a $\sqrt{e} \times \sqrt{e}$ mesh [112].
- Trapezoidal decomposition of a set of n (possibly intersecting) segments [82]. Note that visibility is a special case of trapezoidal decomposition.
- Polygon triangulation. This follows from the above-mentioned trapezoidal decomposition result of Jeong and Lee, and the fact that polygon triangulation can be solved by two calls to the trapezoidal decomposition procedure [130].
- Topological sorting of nonintersecting line segments. This follows from [23] and the above-mentioned trapezoidal decomposition result.
- The area of the union of iso-oriented rectangles [100].
- Intersection detection between n planar line segments [82], [100].
- Computing the largest empty rectangle [53]. This is the problem of computing the largest-area iso-oriented rectangle that is constrained to lie in a given iso-oriented rectangular region and not to contain any of n given points.
- Three-dimensional convex hull, computing the intersection of two three-dimensional convex polyhedra [125], [80], [93].

Other geometric problems considered in the literature include the computation of robot configuration space [55], visibility and separability [54], ECDF searching [60], and multipoint and planar point location [82].

The following problems remain open on the mesh, in the sense that no $O(\sqrt{n})$ time algorithm on a $\sqrt{n} \times \sqrt{n}$ mesh is known for them.

- Convex layers in the plane. (See the end of subsection IV-A for a definition of this problem.)
- The layers of maxima in the plane. This is defined in a similar way to convex layers, but with the words "convex hull" replaced by "maximal elements."

The most interesting open geometric problems on the mesh are in the hybrid model described in the next section, and will be mentioned there.

Another framework in which geometric problems have been considered on the mesh is that in which the input geometric figure is a binary image stored in the mesh in the natural way (the (i, j)th pixel is stored in the processor at row i and column j). The techniques needed in this image processing framework can be quite different from those we mentioned above and are not within the scope of this survey (see, for example, [101] [56]).

VI. A Hybrid Model: The RAM/ARRAY

The main justification for the hybrid model that is the subject of this section is that many existing parallel machines have a "front end" that is a conventional sequential computer, and the number of processors in the parallel machine itself is typically the fixed number purchased rather than a function of the problem size n.

Suppose we have a parallel machine (such as a d-dimensional mesh-connected array of p processors) that can solve a problem of size p in time $O(p^{1/d})$ (this includes the time to input the data to the array as well as the actual computation time, a standard assumption in the literature of mesh-connected arrays, and certainly a reasonable one for the case $d = 1$). Suppose such a mesh-connected array of processors is attached to a conventional random access machine (RAM) that wishes to solve a problem of size $n > p$. We call such a machine a RAM/ARRAY(d). It is important to realize that the mesh alone cannot even store the description of the geometric problem, because of the limitation that each processor has $O(1)$ storage registers; hence the sequential "front end" must play a role in the solution process. If the problem's sequential time complexity is, say, $\Theta(n \log n)$, then the mesh gives a factor of $s(p) = p^{1-1/d} \log p$ speedup *for a problem of size p*. However, if the RAM/ARRAY(d) is trying to solve a problem of size n, $n > p$, then it is not clear how it should use the mesh to achieve the factor of $s(p)$ speedup and obtain $O(n \log n/s(p))$ time performance. Actually, it is not even clear whether maintaining the $s(p)$ speedup is at all possible. Identifying the problems for which this optimal $O(n \log n/s(p))$ time can be achieved is an interesting question that was originally posed, for sorting in the case $d = 1$, by Mueller [104], who also gave a partial solution. The question has been answered in the affirmative in [17] and [27]. This result immediately implies an affirmative answer on a RAM/ARRAY(1) for the geometric problems that can be solved in linear time after a preprocessing sorting step, for example the planar convex hull and maximal elements problems. The $O(n \log n/s(p))$ time bound can also be achieved on a RAM/ARRAY(1) for the following geometric problems [26]: all nearest neighbors of a planar set of points, the measure and perimeter of a union of rectangles, the visibility of a set of nonintersecting line segments from a point, three-dimensional maxima, and dominance counting between two sets of points (hence the related problem of counting intersections between rectilinear rectangles). Essentially the same method as for the RAM/ARRAY(1) establishes that all these problems can be solved in $O(n \log n/s(p))$ on a RAM/ARRAY(d), with $s(p) = p^{1-1/d} \log p$ [26].

We illustrate the technique for the case $d = 1$ and for geometric problems whose sequential time complexity is $\Theta(n \log n)$, i.e., when the task is to design $O(n \log n/ \log p)$ time algorithms on a RAM/ARRAY(1). In that case the algorithm usually follows the p-way divide-and-conquer paradigm (there is an alternative method, using a "lazy B-tree" approach [17], which we do not discuss). That is, the algorithm divides the problem into p subproblems. Then it recursively solves each of the p subproblems, one after the other. After the p recursive calls return, it combines the subsolutions to form the final solution. The main difficulty is how to perform the combining step in $O(n)$ time. If the combining step can be performed in $O(n)$ time, then the overall time complexity $T(n)$ satisfies the recurrence $T(n) = p \cdot T(n/p) + O(n)$, which implies that $T(n)$ is $O(n \log n/ \log p)$. In the case of a RAM/ARRAY(d) where $d > 1$, instead of partitioning the problem into p subproblems, the problem gets partitioned into $p^{1/(d+1)}$ subproblems. In that case the $p^{1/(d+1)}$ subsolutions must be "combined" in $O(n/p^{1-1/d})$ time.

The following result from sequential computation, reported by Frederickson and Johnson [69], is often useful in the RAM/ARRAY(d) framework. Given an $a \times b$ matrix ($b \leq a$) whose columns are sorted, the kth smallest element can be selected in time $O(b + m \log(k/m))$, where $m = \min\{k, b\}$, if the matrix is already in the memory, or if any element of the matrix can be produced in constant time. This implies that the bth element can be selected from the matrix in $O(b)$ time. This selection algorithm has been used in one of the two schemes given in [17] to establish the optimal sorting algorithm for this model, and it turned out to be a crucial tool for many geometric problems [26].

The question whether the speedup of $s(p)$ that the$d$-dimensional array makes possible for a problem of size p can be carried over to larger problems is really dealing with the fundamental issue of the *parallel decomposability* of the problem at hand: given that a problem of size p can be solved on a parallel machine faster by a factor of (say) $s(p)$ than on a RAM alone, then that problem is *fully parallel decomposable* if a RAM to which the parallel machine is attached can solve arbitrarily large instances of that problem with a speedup of $s(p)$ when compared with a RAM alone. Although it is known that for some geometric problems the speedup of $s(p)$ for a problem of size p can indeed also be translated into a speedup of $s(p)$ for problems of size n, $n > p$, this question remains open for many other classical geometric problems, such as:

- Topological sorting of nonintersecting line segments.

- Trapezoidal decomposition. However, if the nonintersecting line segments $s_1, s_2, \cdots, s_n$ are given in topologically sorted order, then it is known how to solve it in $O(n \log n / s(p))$ time on a RAM/ARRAY(d) [26]. What makes the problem easier in that case is the fact that, if one partitions the problem into p equal-sized subproblems according to their topological order, then the "interaction" between subproblems is encapsulated by their visibilities from a point at infinity. In particular, it is known for the case where the line segments are horizontal because sorting them by y coordinates is like sorting them topologically.
- Voronoi diagram of a planar point set.
- Three-dimensional convex hull, computing the intersection of two three-dimensional convex polyhedra.

Negative results would also be interesting: which problems are inherently such that it is impossible to maintain the same speedup for $n > p$ as for $n = p$?

The techniques developed for RAM/ARRAY(d)'s have also been used in [124] to achieve linear speedups on several hypercube-related computers which consist of p processors, each containing $O(n/p)$ local memory, provided that $n > p^{1+\epsilon}$ for some constant $\epsilon > 0$. The same speedup is known for sorting [4], [50].

Finally, there are close connections between the work on parallel decomposability and the work on I/O complexity [7], [81]. In the study of I/O complexity, one is given a sequential computer which has a small main memory and a large secondary storage, and one is interested in solving problems of arbitrarily large size. The input of the problem is initially stored in the secondary storage and the output has to be written to the secondary storage. The limitation that the size of the main memory is small is similar to the limitation that the size of the attached parallel machine is small. The major concern in the study of I/O complexity is to minimize the amount of I/O between the main memory and the secondary storage. To achieve the best I/O performance, the algorithm is allowed arbitrarily long computation times for scheduling the I/O's (i.e., only the amount of I/O matters). On the other hand, the time to decompose the computation into subcomputations and to schedule the subcomputations must be counted in the study of parallel decomposability. The techniques developed for the study of geometric parallel decomposability can be used to obtain I/O complexity bounds for the geometric problem considered [124].

VII. Experimental Work

Much of the work in parallel computational geometry has been theoretical in nature, but some researchers have implemented geometric algorithms on various parallel architectures and reported interesting results.

Blelloch [30], [31] has implemented parallel geometric algorithms on the Connection Machine (CM),[2] including convex hull (the $\sqrt{n}$-divide-and-conquer method we mentioned earlier). Blelloch argued that in the CM architecture, *scan* operations (essentially, parallel prefix) are implemented so efficiently that one should solve problems on the CM architecture by using, whenever possible, calls to these built-in routines. In fact he went as far as assuming the cost of a parallel prefix to be $O(1)$, and gave a detailed study of the implications of such an assumption in solving various problems. The experimental data obtained by Blelloch and by other researchers seem to confirm that Blelloch's assumption is quite reasonable.

[2]"Connection Machine" and "CM" are registered trademarks of the Thinking Machines Corporation.

Cohen, Miller, Sarraf, and Stout have implemented parallel geometric algorithms on hypercube architectures such as the iPSC, including convex hulls and domination [40], and convex hulls of digitized pictures [97].

The above-mentioned experimental work demonstrates, among other things, that algorithmic ideas developed for abstract parallel models can be useful when programming "real" parallel machines.

Generally speaking, work in parallel computational geometry continues to be mostly theoretical, with experimental work being the exception rather than the rule. Perhaps this will change as researchers gain increased access to parallel machines.

VIII. Further Remarks

In view of the importance of the hypercube, surprisingly few geometric algorithms have been designed for this parallel model (see [33], [58], [59], [97], [99], and [119] for some of these). We believe that, once the complexity of such basic operations as sorting and list ranking is settled for the hypercube model, algorithm design for geometric problems on that model will probably receive increased attention. An important step in this direction has recently been taken in the new sorting algorithm of Cypher and Plaxton [49]. One way around the "sorting bottleneck" for the hypercube would be to take the randomization approach, as Reif and Sen did [113] (sorting is then no longer a bottleneck, since there is an optimal randomized sorting algorithm for the hypercube [116]).

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

- Optimal deterministic PRAM construction of Voronoi diagrams in the plane. The current best bounds are, in the CREW-PRAM model, $O(\log n \log \log n)$ time and $n \log n / \log \log n$ processors or, alternatively, $O(\log^2 n)$ time and $n / \log n$ processors (see [43]).
- Optimal deterministic PRAM construction of three-dimensional convex hull.
- Optimal EREW-PRAM solution to linear programming in the plane (an algorithm exists in the CRCW model [61]).

The following are additional promising directions for future research:

- Output-sensitive PRAM algorithms—where the complexity depends on the size of the output (for example, in [75], the number of processors needed depends on the number of intersections). Most geometric problems remain open when looked at from this perspective (even the planar convex hull problem).

- Robust parallel algorithms. Recall that robust algorithms are such that their correctness is not destroyed by roundoff error. Most existing parallel geometric algorithms misbehave if implemented with rounded arithmetic. There has recently been a flurry of activity in designing efficient and robust sequential algorithms for geometric problems (see [96] for a list of references), and we expect this important activity to spread to the design of parallel geometric algorithms as well.

Acknowledgment

It is a pleasure to acknowledge the helpful comments of D. Chen on a earlier draft of this survey.

References

[1] K. Abrahamson, N. Dadoun, D. A. Kirpatrick, and T. Przytycka, "A simple parallel tree contraction algorithm," *J. Algorithms*, vol. 10, pp. 287–302, 1989.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel computational geometry," *Algorithmica*, vol. 3, pp. 293–328, 1988.

[3] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric applications of a matrix searching algorithm," *Algorithmica*, vol. 2, pp. 209–233, 1987.

[4] A. Aggarwal and M.-D. Huang, "Network complexity of sorting and graph problems and simulating {CRCW PRAMS} by interconnection networks," in *Lecture Notes in Computer Science, 319: VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88*, 1988, pp. 339–350.

[5] A. Aggarwal and J. Park, "Parallel searching in multidimensional monotone arrays," in *Proc. 29th Ann. IEEE Symp. Foundations of Computer Science*, 1988, pp. 497–512. (To appear in *J. Algorithms.*)

[6] A. Aggarwal and S. Suri, "Fast algorithms for computing the largest empty rectangle," in *Proc. 3rd ACM Symp. Computational Geometry*, 1987, pp. 278–290.

[7] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. Ass. Comput. Mach.*, vol. 31, pp. 1116–1127, 1988.

[8] M. Ajtai, J. Komlos, and E. Szemeredi, "Sorting in $c \log n$ parallel steps," *Combinatorica*, vol. 3, pp. 1–19, 1983.

[9] S. G. Akl, "A constant-time parallel algorithm for computing convex hulls," *BIT*, vol. 22, pp. 130–134, 1982.

[10] R. Anderson and G. L. Miller, "Deterministic parallel list ranking," in *Lecture Notes in C. S. 319: 3rd AWOC*, 1988, pp. 81–90.

[11] 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, pp. 968–988, 1990.

[12] M. J. Atallah, "A faster parallel algorithm for a matrix searching problem," in *Proc. 2d Scandinavian Workshop on Algorithm Theory*, 1990, pp. 192–200. (To appear in *Algorithmica.*)

[13] M. J. Atallah, P. Callahan, and M. T. Goodrich, "P-complete geometric problems," in *Proc. 2d Ann. ACM Symp. Parallel Algorithms and Architectures*, 1990, pp. 317–326.

[14] M. J. Atallah and D. Z. Chen, "Parallel rectilinear shortest paths with rectangular obstacles," in *Proc. 2d Ann.ACM Symp. Parallel Algorithms and Architectures*, 1990, pp. 270–279.

[15] M. J. Atallah, D. Z. Chen, and H. Wagener, "An optimal parallel algorithm for the visibility of a simple polygon from a point," *J. Ass. Comput. Mach.*, to be published; a preliminary version appeared in *Proc. 5th Ann. ACM Symp. Computational Geometry* (Saarbrucken, Federal Republic of Germany), 1989, pp. 114–123.

[16] M. J. Atallah, R. Cole, and M. T. Goodrich, "Cascading divide-and-conquer: A technique for designing parallel algorithms," *SIAM J. Comput.*, vol. 18, pp. 499–532, 1989.

[17] M. J. Atallah, G. N. Frederickson, and S. R. Kosaraju, "Sorting with efficient use of special-purpose sorters," *Inform. Process. Lett.*, vol. 27, pp. 13–15, 1988.

[18] M. J. Atallah and M. T. Goodrich, "Efficient plane sweeping in parallel," in *Proc. 2nd Ann. ACM Symp. Computational Geometry* (Yorktown Heights, NY), 1986, pp. 216–225.

[19] M. J. Atallah and M. T. Goodrich, "Efficient parallel solutions to some geometric problems," *J. Parallel and Distributed Computing*, vol. 3, pp. 492–507, 1986.

[20] M. J. Atallah and M. T. Goodrich, "Parallel algorithms for some functions of two convex polygons," *Algorithmica*, vol. 3, pp. 535–548, 1988.

[21] M. J. Atallah, S. R. Kosaraju and M. T. Goodrich, "On the parallel complexity of evaluating some sequences of set manipulation operations," in *Lecture Notes in Computer Science, 319: VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88*, 1988, pp. 1–10.

[22] M. J. Atallah and S. E. Hambrusch, "Solving tree problems on a mesh-connected processor array," *Info. and Control*, vol. 69, pp. 168–187, 1986.

[23] M. J. Atallah, S. E. Hambrusch, and L. E. TeWinkel, "Parallel topological sorting of features in a binary image," in *Proc. 26th Ann. Allerton Conf. Communication, Control, and Computing* (Monticello, IL) 1988, pp. 1114–1115. (To appear in *Algorithmica.*)

[24] M. J. Atallah and S. R. Kosaraju, "An efficient parallel algorithm for the row minima of a totally monotone matrix," in *Proc. 2nd ACM-SIAM Symp. Discrete Algorithms*, 1991, pp. 394–403.

[25] M. J. Atallah, S. R. Kosaraju, L. Larmore, G. L. Miller and S. Teng, "Constructing trees in parallel," in *Proc. 1st Ann. ACM Symp. Parallel Algorithms and Architectures* (Santa Fe, NM), 1989, pp. 421–431.

[26] M. J. Atallah and J.-J. Tsay, "On the parallel-decomposibility of geometric problems," in *Proc. 5th Ann. ACM Symp. Computational Geometry*, 1989, pp. 104–113.

[27] R. Beigel and J. Gill, "Sorting n objects with a k-sorter," *IEEE Trans. Comput.*, to be published.

[28] P. Bertolazzi, C. Guerra, and S. Salza, "A parallel algorithm for the visibility problem from a point," *J. Parallel and Distributed Computing*, vol. 9, pp. 11–14, 1990.

[29] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines," *SIAM J. Comput.*, vol. 18, pp. 216–228, 1989.

[30] G. E. Blelloch, "Scan primitives and parallel vector models," Ph.D. thesis, Massachusetts Institute of Technology, 1988.

[31] G. E. Blelloch, "Scans as primitive parallel operations," in *Proc. Int. Conf. Parallel Processing*, 1987, pp. 355–362.

[32] A. Borodin and J. E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *J. Computer and System Sciences*, vol. 30, pp. 130–145, 1985.

[33] L. Boxer and R. Miller, "Dynamic computational geometry on meshes and hypercubes," *J. Supercomputing*, vol. 3, pp. 161–191, 1989.

[34] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. Ass. Comput. Mach.*, vol. 21, pp. 201–206, 1974.

[35] B. M. Chazelle, "Optimal algorithms for computing depths and layers," in *Proc. 20th Allerton Conf. Communications, Control and Computing*, 1983, pp. 427–436.

[36] B. M. Chazelle, "Computational geometry on a systolic chip," *IEEE Trans. Comput.*, vol. 33, pp. 774–785, 1984.

[37] B. Chazelle and L. J. Guibas, "Fractional cascading: I. A data structuring technique," *Algorithmica*, vol. 1, pp. 133–162.

[38] D. Z. Chen, "Efficient geometric algorithms in the EREW-PRAM," in *Proc. 28th Ann. Allerton Conf. Communication, Control, and Computing* (Monticello, IL), 1990, pp. 818–827.

[39] A. Chow, "Parallel algorithms for geometric problems," Ph.D. thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 1980.

[40] E. Cohen, R. Miller, E. M. Sarraf and Q. F. Stout, "Efficient convexity and domination algorithms for fine- and medium-grain hypercube computers," *Algorithmica*, to be published.

[41] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, pp. 770–785, 1988.

[42] R. Cole and M. T. Goodrich, "Optimal parallel algorithms for point-set and polygon problems," in *Proc. 4th Ann. ACM Symp. Computational Geometry*, 1988, pp. 201–210.

[43] R. Cole, M. T. Goodrich and C. O'Dunlaing, "Merging free

trees in parallel for efficient Voronoi diagram construction," in *Proc. 17th Int. Colloq. Automata, Lang., and Programming*, 1990, pp. 432–445.

[44] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems," in *Proc. 27th Ann. IEEE Symp. Foundations of Comp. Sci.*, 1986, pp. 487–491.

[45] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking," *Info. and Control*, vol. 81, pp. 334–352, 1989.

[46] R. Cole and O. Zahicek, "An optimal parallel algorithm for building a data structure for planar point location," Courant Inst. Tech. Rep. 316, 1987. (To appear in *J. Parallel and Distributed Computing*.)

[47] S. Cook and C. Dwork, "Bounds on the time for parallel RAM's to compute simple functions," in *Proc. 14th ACM Annual Symp. Theory of Computing,* 1982, pp. 231–233.

[48] J. Czyzowicz, I. Rival, and J. Urrutia, "Galleries, light matchings, and visibility graphs," in *Lecture Notes in CS: 382, Proc. WADS 89*, 1989, pp. 316–324.

[49] R. Cypher and C. G. Plaxton, "Deterministic sorting in nearly logarithmic time on the hypercube and related computers," in *Proc. 22d Ann. ACM Symp. Theory of Computing*, 1990, pp. 193–203.

[50] R. Cypher and J. L. C. Sanz, "Optimal sorting on feasible parallel computers," in *Proc. Int. Conf. Parallel Processing*, 1988, pp. 339–350.

[51] N. Dadoun and D. Kirkpatrick, "Parallel processing for efficient subdivision search," in *Proc. 3rd ACM Symp. Computational Geom.*, 1987, pp. 205–214.

[52] N. Dadoun, "Geometric hierarchies and parallel subdivision search," Ph.D. thesis, U. of British Columbia, 1990.

[53] F. Dehne, "Computing the largest empty rectangle on one and two dimensional processor arrays," *J. Parallel Dist. Comput.*, to be published.

[54] F. Dehne, "Solving visibility and separability problems on a mesh of processors," *Visual Computer,* vol. 3, pp. 356–370, 1988.

[55] F. Dehne, A.-L. Hassenklover, and J.-R. Sack, "Computing the configuration space for a robot on a mesh of processors," *Parallel Computing*, to be published.

[56] F. Dehne, A.-L. Hassenklover, J.-R. Sack, and N. Santoro, "Computational geometry algorithms for the systolic screen," *Algorithmica*,, to be published.

[57] 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," in *Proc. 1988 Scandinavian Workshop on Algorithms and Theory,* pp. 154–162.

[58] F. Dehne, A. Ferreira and A. Rau-Chaplin, "Parallel fractional cascading on a hypercube multiprocessor," in *Proc. 27th Ann. Allerton Conf. Communication, Control, and Computing* (Monticello, IL), 1989, pp. 1084–1093.

[59] F. Dehne and A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor, with applications in parallel computational geometry," *J. Parallel Distrib. Computing*, vol. 8, pp. 367–375, 1990.

[60] F. Dehne and I. Stojmenovic, "An $O(\sqrt{n})$ time algorithm for the ECDF searching problem for arbitrary dimensions on a mesh of processors," *Inform. Process. Lett.*, vol. 28, pp. 67–70, 1988.

[61] X. Deng, "An optimal parallel algorithm for linear programming in the plane," *Inform. Process. Lett.*, vol. 35, pp. 213–217, 1990.

[62] D. P. Dobkin and D. G. Kirkpatrick, "Fast detection of polyhedral intersections," *Theor. Comp. Sci.*, vol. 27, pp. 241–253, 1983.

[63] D. P. Dobkin and D. G. Kirkpatrick, "A linear time algorithm for determining the separation of convex polyhedra," *J. Algorithms,* vol. 6, pp. 381–392, 1985.

[64] D. P. Dobkin and D. G. Kirkpatrick, "Determining the separation of prepprocessed polyhedra – A unified approach," in *Proc. Int. Colloq. Automata, Lang., and Programming,* 1990, pp. 154–165.

[65] D. Dobkin, R. J. Lipton, and S. Reiss, "Linear programming is log-space hard for P" *Inform. Process. Lett.*, vol. 9, pp. 96–97, 1979.

[66] D. Dobkin and S. Reiss, "The complexity of linear programming," *Theor. Comp. Sci.,* vol. 11, pp. 1–18, 1980.

[67] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*. New York: Springer-Verlag, 1987.

[68] H. ElGindy and M. T. Goodrich, "Parallel algorithms for shortest path problems in polygons," *The Visual Computer: International J. Computer Graphics*, vol. 3, pp. 371–378, 1988.

[69] G. N. Frederickson and D. B. Johnson, "The complexity of selection and ranking in $\{X + Y\}$ and matrices with sorted columns," *J. Computer and System Sciences*, vol. 24, pp. 197–208, 1982.

[70] L. M. Goldschlager, "The monotone and planar circuit value problems are log space complete for P," *SIGACT News*, vol. 9, pp. 25–29, 1977.

[71] A. Gibbons and W. Rytter, "An optimal parallel algorithm for dynamic expression evaluation and its applications," in *Proc. Symp. Found. Software Technology and Theoretical Comp. Sci.*, 1986, pp. 453–469.

[72] M. T. Goodrich, "Efficient parallel techniques for computational geometry," Ph.D. thesis, Department of Computer Science, Purdue University, 1987.

[73] M. T. Goodrich, "Finding the convex hull of a sorted point set in parallel," *Inform. Process. Lett.*, vol. 26, pp. 173–179, 1987.

[74] M. T. Goodrich, "Triangulating a polygon in parallel," *J. Algorithms*, in press.

[75] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," Tech. Rep. 88–27, Johns Hopkins Univ. 1988.

[76] M. T. Goodrich, M. Ghouse, and J. Bright, "Generalized sweep methods for parallel computational geometry," in *Proc. 2d Ann. ACM Symp. Parallel Algorithms and Architectures*, 1990, pp. 280–289.

[77] M. T. Goodrich, S. B. Shauck, and S. Guha, "Parallel method for visibility and shortest path problems in simple polygons," in *Proc. 6th Ann. ACM Symp. Computational Geometry*, 1990, pp. 73–82.

[78] J. Hershberger, "Upper envelope onion peeling," in *Proc. 2d Scandinavian Workshop on Algorithm Theory*, 1990, pp. 368–379.

[79] T. Hagerup, and C. Rub. "Optimal merging and sorting on the EREW PRAM," *Inform. Process. Lett.*, vol. 33, pp. 181–185, 1989,

[80] J. A. Holey and O. H. Ibarra, "Triangulation in a plane and 3-D convex hull on mesh-connected arrays and hypercubes," Tech. Rep., Dept. of Computer Science, Univ. of Minnesota, 1990.

[81] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proc. 13th Ann. ACM Symp. Theory of Computing*, 1981, pp. 326–333.

[82] C. S. Jeong and D. T. Lee, "Parallel geometric algorithms on a mesh connected computer," Tech. Rep, 87–02-FC-01, Dept. EE/CS, Northwestern Univ., 1987. To appear in *Algorithmica*.

[83] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," Tech. Rep. CSD-TR 88/408, U. C. Berkeley, Mar. 1988.

[84] S. R. Kosaraju and A. Delcher, "Optimal parallel evaluation of tree-structured computations by raking," *Lecture Notes in CS 319: AWOC 88*, Springer Verlag, 1988, pp. 101–110.

[85] D. G. Kirkpatrick. "Optimal search in planar subdivisions," *SIAM J. Comput.*, vol. 12, pp. 28–35, 1983.

[86] M. Kunde, "Optimal sorting on multidimensional mesh-connected computers," in *Proc. STACS 1987* (Lecture Notes in Computer Science) 1987, pp. 408–419.

[87] H. T. Kung and C. D. Thompson, "Sorting on a mesh-connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, p. 263, 1977.

[88] C. P. Kruskal, L. Rudolph, and M. Snir, "A complexity theory of efficient parallel algorithms," *Lecture Notes in CS:317, Proc. 15th ICALP*, 1988, pp. 333–346.

[89] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," in *Proc. 1985 IEEE Int. Conf. Parallel Processing* (St. Charles, IL), pp. 180–185.

[90] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. Ass. Comput. Mach.*, pp. 831–838, 1980.

[91] C. Levcopoulos, J. Katajainen, and A. Lingas, "An optimal expected time algorithm for Voronoi diagrams," in *Proc. 1st Scandinavian Workshop on Algorithm Theory*, 1988.

[92] D. T. Lee and F. P. Preparata, "Computational geometry—A survey," *IEEE Trans. Comput.*, vol. 33, pp. 872–1101, 1984.

[93] D. T. Lee, F. P. Preparata, C. S. Jeong, and A. L. Chow, "SIMD parallel convex hull algorithms," Tech. Rep. AC-91–02, Dept. Electrical Eng. and Computer Science, Northwestern Univ., 1991.

[94] F. T. Leighton, *An Introduction to Parallel Algorithms and*

Architectures. California: Morgan Kaufmann, 1991.

[95] J. M. Marberg and E. Gafni, "Sorting in constant number of row and column phases on a mesh," in *Proc. 24th Ann. Allerton Conf. Communication, Control, and Computing* (Monticello, IL), 1986, pp. 603–612.

[96] Z. Li and V. Milenkovic, "Constructing strongly convex hulls using exact or rounded arithmetic," in *Proc. Sixth Ann. ACM Symp. Comp. Geom.*, 1990, pp. 235–243. (To appear in *Algorithmica.*)

[97] R. Miller, "Convexity algorithms for digitized pictures on an Intel iPSC hypercube," *Supercomputer*, no. 31, VI-3, pp. 45–53.

[98] R. Miller and Q. F. Stout, *Parallel Algorithms for Regular Architectures.* Cambridge, MA: MIT Press, 1991.

[99] R. Miller and Q. F. Stout, "Efficient parallel convex hull algorithms," *IEEE Trans. Comput.* vol. 37, pp. 1605–1618, 1988.

[100] R. Miller and Q. F. Stout, "Mesh computer algorithms for computational geometry," *IEEE Trans. Comput.*, vol. 38, pp. 321–340, 1989.

[101] R. Miller and Q. F. Stout, "Geometric algorithms for digitized pictures on a mesh connected computer," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 7, pp. 216–228, 1985.

[102] G. L. Miller and J. H. Reif, "Parallel tree contraction and its applications," in *Proc. 26th ACM Symp. Foundations of Comp. Sci.*, 1985, pp. 478–489.

[103] S. Miyano, S. Shiraishi, and T. Shoudai, "A list of P-complete problems," Tech. Rep. RIFIS-TR-CS-17, 1989, Kyushu Univ., Japan.

[104] H. Mueller, "Sorting numbers using limited systolic coprocessors," *Inform. Process. Lett.*, vol. 24, pp. 351–354, 1987.

[105] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. Comput.*, vol, 30, pp. 101–106, 1981.

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

[107] J. O'Rourke, "Computational geometry," *Ann. Rev. Comp. Sci.*, vol. 3, pp. 389–411, 1988.

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

[109] W. Paul, U. Vishkin, and H. Wagener, "Parallel dictionaries on 2–3 trees," in *Proc. 10th Coll. Autom., Lang., and Prog. (ICALP), LNCS 154*, 1983, pp. 597–609.

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

[111] F. P. Preparata and R. Tamassia, "Fully dynamic techniques for point location and transitive closure in planar structures," in *Proc. 29th ACM Symp. Theory of Computing*, 1988, pp. 558–567.

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

[113] J. H. Reif and S. Sen, "Optimal randomized parallel algorithms for computational geometry," in *Proc. 1987 IEEE Int. Conf. Parallel Processing*, pp. 270–277. (To appear in *Algorithmica.*)

[114] J. H. Reif and S. Sen, "Polling: A new random sampling technique for computational geometry," in *Proc. 21st Ann. ACM Symp. Theory of Computing*, 1989, pp. 394–404.

[115] J. H. Reif and S. Sen, "Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications (preliminary version)," in *Proc. 2nd Ann. ACM Symp. Parallel Algorithms and Architectures* 1990, pp. 327–337.

[116] J. H. Reif and L. Valiant, "A logarithmic time sort for linear size networks," in *Proc. 15th ACM Symp. Theory of Comp.*, 1981.

[117] Y. Shiloach and U. Vishkin, "Finding the maximum, merging, and sorting in a parallel computation model," *J. Algorithms*, vol. 2, pp. 88–102, 1981.

[118] C. P. Schnorr and A. Shamir, "An optimal sorting algorithm for mesh connected computers," in *Proc. 18th ACM Symp. Theory on Computing*, 1986, pp. 255–261.

[119] I. Stojmenovic, manuscript, 1988.

[120] Q. F. Stout, "Constant-time geometry on PRAMs," in *Proc. 1988 Int. Conf. Parallel Computing*, vol. III, pp. 104–107.

[121] R. Tamassia and J. S. Vitter, "Parallel transitive closure and point location in planar structures," in *Proc. 1st Ann. ACM Symp. Parallel Algorithms and Architectures*, 1989, pp. 399–408.

[122] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *SIAM J. Comput.*, vol. 14, pp. 862–874, 1985.

[123] G. T. Toussaint, "Solving geometric problems with rotating calipers," in *Proc. IEEE MELECON '83* (Athens, Greece), May 1983.

[124] J.-J. Tsay, "Optimal medium-grained parallel algorithms for geometric problems," Tech. Rep. 942, Purdue CS Dept. , 1990.

[125] 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," Tech. Rep. 91–012, Purdue CS Dept , 1991. (To appear in *Proc. 2d Ann. ACM Symp. Parallel Algorithms and Architectures* (Hilton Head, SC), 1991.

[126] P. Vaidya, private communication.

[127] L. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, pp. 348–355, 1975.

[128] H. Wagener, "Optimally parallel algorithms for convex hull determination," manuscript, 1985.

[129] D. E. Willard and Y. C. Wee, "Quasi-valid range querying and its implications for nearest neighbor problems," in *Proc. 4th Ann. ACM Symp. Computational Geometry*, 1988, pp. 34–43.

[130] C. K. Yap, "Parallel triangulation of a polygon in two calls to the trapezoidal map," *Algorithmica*, vol. 3, pp. 279–288, 1988.

Mikhail J. Atallah (Senior Member, IEEE) received the B.E. degree in electrical engineering from the American University, Beirut, Lebanon, in 1975 and the M.S.E. and Ph.D. degrees in electrical engineering and computer science from Johns Hopkins University, Baltimore, MD, in 1980 and 1982, respectively.

In August 1982 he joined Purdue University, West Lafayette, IN, where he is currently Professor of Computer Science. In 1985 he received a Presidential Young Investigator award from the National Science Foundation. His research interests include the design and analysis of algorithms, parallel computation, and computational geometry.

Dr. Atallah is a member of the Association for Computing Machine and the Society for Industrial and Applied Mathematics. He serves on the editorial boards of the journals *Computational Geometry: Theory & Applications*; *Information Processing Letters*; *International Journal on Computational Geometry & Applications*; *Methods of Logic in Computer Science*; *Parallel Processing Letters*; and *SIAM Journal on Computing*. He is currently guest editor for a special issue of *Algorithmica* on computational geometry. In addition, he has served on conference program committees and state and federal panels.