ELSEVIER

Theoretical Computer Science 188 (1997) 59–78

Theoretical Computer Science

Lower bounds for parallel algebraic decision trees, parallel complexity of convex hulls and related problems[1]

Sandeep Sen*

Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India

Received November 1994
Communicated by F.P. Preparata

Abstract

We present lower bounds on the number of rounds required to solve a decision problem in the parallel algebraic decision tree model. More specifically, we show that any parallel algorithm in the fixed degree algebraic decision tree model that answers membership queries in $W \subseteq R^n$ using p processors, requires $\Omega(\log |W|/n \log(p/n))$ rounds where $|W|$ is the number of connected components of W. This implies non-trivial lower bounds for parallel algorithms that use a super-linear number of processors, namely, that the speed-up obtainable in such cases is not proportional to the number of processors. We further prove a similar result for the average case complexity. We give applications of this result to various fundamental problems in computational geometry like convex-hull construction and trapezoidal decomposition and also present algorithms with matching upper bounds. The algorithms extend Reif and Sen's work in parallel computational geometry to the sublogarithmic time range based on recent progress in padded-sorting. A corollary of our result strengthens the known lower-bound of parallel sorting from the parallel comparison tree model to the more powerful bounded-degree decision tree.

1. Introduction

The primary objective of designing parallel algorithms is to obtain faster algorithms. Nonetheless, the pursuit of higher speed has to be weighted against the concerns of efficiency, namely, if we are getting our money's (processor's) worth. Ideally, one aims for speed-ups proportional to the number of processors, but in reality this often turns out to be an elusive goal. This is not because of any shortcoming of the algorithm designer; rather this bottleneck is a characteristic of the problem at hand. One of the celebrated results of this kind is the $\Omega(\log n/\log\log n)$ lower bound on the time

* E-mail: ssen@cse.iitd.ernet.in.

[1] Part of the work was done when the author was visiting Max-Plank Institüte für Informatik, Germany in summer of 1993. A preliminary version of this paper appeared in the Proc. 14th Conf. of the FST & TCS, Madras, India, 1994.

PII S0304-3975(96)00212-5

complexity for parallel summation of n bits using a polynomial number of processors in the CRCW PRAM model due to Beame and Hastad ([7]). This was a big setback for the algorithm designer who could not for example aim for an O(1) time algorithm for parallel sorting using even n^{10} processors (since sorting can be used to compress the n bits). Moreover, because sorting is such a fundamental problem, it implied a similar bound for a number of other problems as well.

However, in the parallel comparison tree model (to be referred subsequently as PCT), the scenario was less grim in light of the results due to Alon and Azar [2] and Azar and Vishkin [6]. For example, their results implied that sorting can be done in O(1) time with $n^{3/2}$ comparisons and moreover one cannot do much better. Since PCT model accounts for all comparisons, one has to accept this as an inherent limitation for any comparison-based parallel sorting algorithm (even in the PRAM model). The actual work of circumventing the lower bound of Beame and Hastad in the PRAM model required an alternative (more relaxed) notion of sorting called *padded-sorting*, first defined by MacKenzie and Stout [20] and subsequently refined in the work of Raman and Hagerup [17]. Roughly speaking, the problem of padsort involves ordering the input of size n into an output array of size $m \geqslant n$. When $m = n$, or when m is very close to n the lower-bound of Beame and Hastad applies. Hagerup and Raman [17] showed that one can padsort n elements with kn processors in expected time $O(\log n/\log k)$ in a CRCW PRAM as long as $m > n + n/\log n$. Actually, they give a trade-off between m/n and the number of processors. These bounds are asymptotically tight owing to the lower-bound results in [2, 6, 9].

Although the lower-bound results of Beame and Hastad had wide-ranging ramifications, its implications were not clear in the case of several interesting problems, for example, in constructing the convex hull of a point set. It is well-known that a point lies on the convex hull iff it is not contained in any of the triangles defined by a three-tuple of input points. In a CRCW model, one can easily identify the boundary points in O(1) time using $O(n^4)$ processors. However, one cannot hope to compress the vertices in an ordered array in view of the lower bound. Yao [29] had proved that in the sequential context, the identification of the convex hull vertices was no easier than sorting, but our simple argument shows that such is not the case with parallel algorithms. Moreover, such a naive approach also does not shed much light on the actual complexity of this problem.

Our main result implies an $\Omega(\log n/\log k)$ time bound for any deterministic or randomized algorithm for identifying the hull vertices using kn processors, where $k \geqslant 1$ in the parallel algebraic decision tree model. Subsequently, we also establish its tightness by presenting a randomized algorithm with a matching upper bound. Both our lower and upper bounds are more general and apply to several other problems as well. Since the algorithms are based on the approach of Reif and Sen [24, 23], we shall only outline the basic strategy and highlight the portions that require further refinement. Some of the previous techniques have to be carefully fine-tuned to obtain the required bounds.

2. Lower bounds

The model of computation is the parallel analogue of the *bounded-degree decision tree model* (BDD Tree). At each node of this tree, each of the p processors compares the value of a fixed degree polynomial with 0. Accordingly each processor gets a a *sign* $\in \{0, +, -\}$ depending on the result of the comparison being $\{=, >, <\}$ respectively. Subsequently, the algorithm branches according to the *sign vector*, that is by considering the signs of all the processors. The algorithm terminates when we reach a leaf node containing the final answer. If the polynomials are restricted to be of the form $x_i - x_j \leqslant 0$, then it is the parallel comparison tree (PCT) model. While there is no cost for branching (that includes processor allocation and read–write conflicts), it does not have the arithmetic instruction set of the PRAM model. So, strictly speaking it is incomparable with the PRAM model. Notably, the known geometric algorithms in the PRAM model do not exploit this extra power so lower bounds in the BDD Tree are often regarded as binding in the PRAM model also.

Our first lower-bound proof is for the Dominance problem on the PCT model that follows an approach of Boppana [9], who had considerably simplified the lower-bound proof of [2] for parallel sorting. We will first review Boppana's elegant proof technique which establishes a bound on the average-case complexity of parallel sorting and consequently the expected time bound of any randomized algorithm for the worst-case input.

Fact 2.1. *In a parallel comparison (BDD) tree of l leaves and maximum arity a, the average path-length is at least* $\Omega(\log l / \log a)$.

The *arity* of a tree is the maximum number of children at any node. Given this fact (credited to Shannon), one needs a reasonably tight upper bound on the arity of the parallel comparison (BDD) tree model and a lower bound on the number of leaves to establish a lower bound of any parallel algorithm. The number of leaves is related to the number of connected components in the solution space in R^n where n is the dimension of the solution space (which is often the input size). The arity of this tree is the number of distinct outcomes of computations performed by $p > 1$ processors. For sorting, this tree has $n!$ leaves. Bopanna used a result of Manber and Tompa [21] that upper bounds the number of *acyclic orientations* of an undirected graph by $(1 + 2m/n)^n$ where n and m represents the number of vertices and edges, respectively. Sorting can be viewed as assigning directions to the edges of a complete graph on n vertices and taking the transitive-closure after every round of comparisons. Obviously, the graph will remain acyclic at every stage because of the total ordering. The arity can be bounded by $(1 + 2p/n)^n$ as each of the p processors can be viewed as assigning direction to at most one edge – the result of a single comparison. This immediately implies the required bound of $\Omega(\log n / \log(p/n))$.

A point (x_i, y_i) is *dominated* by another point (x_j, y_j) iff $x_i \leqslant x_j$ and $y_i \leqslant y_j$. Given a set of points in plane, the 2-D Dominance problem is to identify all those input points

that are not dominated by any other input point. If we stick to the parallel-comparison model for the 2-D dominance problem, we can prove a similar lower bound as a corollary. Indeed, all known algorithms for the maxima problem use only comparisons to arrive at the solution and hence our assumption is not unjustified. Since the x and the y coordinates are independent, the only useful comparisons are between the x and y coordinates separately. Hence, at each stage, we have two independent acyclic orientations corresponding to each of the coordinate axes. Maximizing product of the cardinalities of the two acyclic orientations is an upper bound on the arity which is less than $(1 + p/n)^{2n}$. It is known that for the n-input dominance problem the number of leaves is $\Omega((n/2)^{(n/2)})$ [19]. For $p = kn$ we obtain the following result.

Lemma 2.2. *In a parallel comparison-tree model, any algorithm that identifies the maximal points among a set of n points in a plane require* $\Omega(\log n/\log k)$ *time using* kn *processors.*

We now shift our attention to the BDD tree model. Note that comparison tree model is not a meaningful computing model for most problems in geometry like the convex hulls, that inherently involve polynomials of degree greater than one. For this, we will first prove a *worst-case* bound along the lines of Ben-Or [8] and subsequently extend it to the average case. The additional complication present in a BDD tree model is that each leaf node may be associated with several connected components of the solution set W. Even if we know $|W|$ (the number of connected components of W), we still need a lower bound on the number of leaves. Ben-Or tackles this by bounding the number of connected components associated with a leaf using results of Milnor and Thom. His result shows that even under these conditions the *worst-case* sequential lower bound is still about $\Omega(\log|W|)$.

If the parallel BDD algorithm uses p processors then the signs of p polynomials can be computed simultaneously. Each test yields a sign and we branch according to the collective possibilities of all the tests. We shall use the following result on the number of connected components induced by m fixed degree polynomial inequalities due to Pollack and Roy [25] to bound the number of such possibilities.

Lemma 2.3. *The number of connected components of all nonempty realizations of sign conditions of m polynomials in d variables, each of degree at most b is bounded by* $((O(bm/d))^d$.

This gives us a bound on the arity of the parallel BDD tree model as well as the number of connected components associated with a leaf node at depth h. The number of polynomials defining the space in a leaf-node at depth h is hp and hence the number of connected components associated with such a node is $((O(bhp/d))^d$. In our context, the number of processors and (hence the polynomial signs computed at each stage) is bounded by kn and d is the dimension of the solution space which is approximately the size of the input. This gives us the following theorem.

Theorem 2.4. *Let $W \subset R^n$ be a set that has $|W|$ connected components. Then any parallel BDD tree algorithm that decides membership in W using kn $(k \geqslant 1)$ processors has time complexity $\Omega(\log |W|/n \log k)$.*

Proof. If h is the length of the longest path in the tree then from Lemma 2.3

$$(ekn/n)^{hn}(ehkn/n)^n \geqslant |W|,$$

where e is a constant that subsumes the degree of the polynomials. The first expression on the left-hand side represents the maximum number of leaves and the second expression is the maximum number of connected components associated with a leaf at depth h. By rearranging terms we obtain

$$h \log(ek) + \log(ehk) \geqslant \log |W|/n.$$

Using $h \log(ek) > \log(ehk)$ for $h > 1$,

$$2h \log(ek) \geqslant \log |W|/n,$$

from which we arrive at the required result. □

The above theorem immediately yields as corollary the $\Omega(\log n/\log k)$ worst-case bound for a number of problems for which $|W|$ has at least $(n/2)^{(n/2)}$. This holds for a slightly modified version (used previously in [19, 27]) of the convex-hull identification problem where the objective is to determine if among a set of n points all the points belong to the convex hull. Note that this version is constant time reducible to the standard version in a CRCW PRAM model with $p \geqslant n$ processors. The same holds true for dominance problem as well [19, 27].

Corollary 2.1. *Any algorithm in the parallel BDD tree model that constructs the convex hull of n points using kn processors requires $\Omega(\log n/\log k)$ rounds, $k \geqslant 1$. The same bound also holds for computing the set of maximal points in a set of n points in the plane.*

Clearly, a similar bound also holds for sorting n real numbers which strengthens the earlier lower bound for the PCT model [2, 6, 9].

Corollary 2.2. *Any algorithm in the parallel BDD tree model requires $\Omega(\log n/\log k)$ steps to sort n numbers using kn processors, $k \geqslant 1$.*

To extend the above result to the average case we require a technical lemma that generalizes the following property of balanced trees – the balanced tree on N leaves, achieves the minimum average height among all trees with N leaves. We extend this property of balanced trees to the case where a leaf node at depth l has a weight l^d

associated with it where d is an integer significantly less than the arity of the tree. The total weight of a tree is the sum of the weights of its leaf-nodes and we will denote this weight function by F_d. The previous property can be viewed as a case where the leaves have unit weight or $d = 0$.

Lemma 2.5. *The tree that attains the minimum average height among all trees with total weight at least w for weight function F_d (defined above) and sufficiently large arity (dependent on d) is a balanced tree.*

Proof. See the Appendix.

In the context of the parallel algebraic decision trees, arity a is $(ek)^n$ and $d = n$. We can then bound the number of leaves of the BDD tree by $\Omega(|W|/(enkL/n)^n)$ where L is the average height. This yields a bound on the average complexity similar to the previous theorem. Using Yao's [28] observation on the connection between average-case complexity and the worst-case complexity of randomized algorithms, we obtain the following result.

Theorem 2.6. *Let $W \subset R^n$ be a set that has $|W|$ connected components. Then any parallel BDD tree algorithm for deciding membership in W using kn ($k \geqslant 1$) processors has average time complexity $\Omega(\log |W|/n \log k)$. Consequently, this also lower bounds the expected running time of any randomized algorithm for the worst-case input.*

Since $|W|$ is at least $(n/2)^{(n/2)}$ for the two-dimensional convex hull problem, the average running time is at least $\Omega(\log n/\log k)$ time. For sorting and the dominance problem, the same bounds hold. By a simple reduction of 2-D dominance to trapezoidal decomposition, we obtain a similar bound for the latter problem. (2-D dominance problem can be solved by constructing the vertical visibility map of the horizontal line segments whose right end-points are the input points and the vertical visibility map is obtained directly from trapezoidal decomposition.)

The *algebraic computation model* is more powerful than the BDD model as it also allows arithmetic operations. This is as powerful as the PRAM model without the indirect addressing. We can view this as a tree where some of the nodes involve arithmetic operations and the others are branching nodes. Ideally, we would like to extend our previous results to the algebraic model. The main difficulty arises from rapid growth in the degree of the polynomials involved. For example, after t rounds (depth t in the computation tree), the degree could be as high as 2^t by repeated squaring. Ben-Or tackled this problem using auxiliary variables, that is by trading degree with dimension of the underlying space. This does not work in the parallel model because a large number of variables can be introduced in every round (equal to the number of processors). Consequently, we obtain a weaker result for the parallel algebraic computation tree by setting the degree of the polynomials to 2^t at depth t and rederiving the bounds.

Theorem 2.7. *Let $W \subset R^n$ be a set that has $|W|$ connected components. Then any parallel algebraic computation tree algorithm for deciding membership in W using kn $(k \geqslant 1)$ processors has time complexity $\Omega(\sqrt{\log|W|/n + \log^2 k} - \log k)$.*

Proof. Using the same notations as in the proof of Theorem 2.4 we obtain the following:

$$\prod_{t=0}^{h} (2^t k)^n \sum_{j=0}^{h} (2^t k)^n \geqslant |W|$$

$$\Rightarrow k^{hn+1}[2^{h(h+1)n/2}][2^{hn+1}/2^n - 1] \geqslant |W|$$

$$\Rightarrow k^{hn-1}[2^{(h+1)(h+2)n/2}] \geqslant |W|$$

$$\Rightarrow (hn+1)\log k + h^2 n \geqslant \log|W|$$

using $h^2 \geqslant \frac{1}{2}(h+1)(h+2)$. Solving the degree-two equation (in h) and using $\log k \ll \log|W|$ yields the required bound. □

Remark. For $2^{\sqrt{\log|W|/n}} = o(k)$, this bound matches that of the parallel BDD model using the fact that $(1+y)^{1/2} > 1 + y/2 - O(y^2)$ for $y < 1$. Here $y = \log|W|/n \log^2 k$. For $|W| = \Omega(n^n)$, this implies that for $2^{\sqrt{\log n}} = o(k)$ the running time is $\Omega(\log n/\log k)$.

3. Parallel algorithms in computational geometry

Designing efficient parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. There have been two distinct approaches to this area of research, namely the deterministic methods and algorithms that use random sampling. A general approach for deterministic PRAM algorithms was pioneered by Aggarwal et al. [1], who developed new techniques for designing efficient parallel algorithms for fundamental geometric problems. A number of the most efficient deterministic PRAM algorithms are due to Atallah et al. [3], who extended the techniques used by Cole [12] for his parallel mergesort algorithm. Their technique is called *Cascaded merging* and has been subsequently used for a number of other problems. An informative account of many basic techniques employed for solving some fundamental problems appears in [4]. In an independent development, Reif and Sen [24] designed $O(\log n)$ expected time optimal speed-up algorithms for point-location and trapezoidal decomposition and subsequently ([23]) extended their methods to develop equally efficient algorithms for 3-D convex hulls on the CREW PRAM model. At the core of their algorithms were random sampling techniques and an efficient resampling technique called *Polling*.

Note that a majority of the geometric problems in the context of research in parallel algorithms have a sequential time complexity of $\Omega(n \log n)$ and a typical performance

that one aims, for is $O(\log n)$ parallel time using an optimal number of processors. The issue of speeding up these algorithms significantly was ignored largely because of the $\Omega(\log n/\log\log n)$ lower bound for parallel sorting in the CRCW model. Problems like the convex hull and 2-D dominance are known to have close relationship with sorting.

The recent results in the area of *padded-sorting* (to be referred to as padsort in future), inspired us to investigate its consequences in the area of geometric problems. To take advantage of the developments in padsort, we will modify the output specifications of the problems relevant to this paper. For example, in the case of two-dimensional convex hulls, we will allow the output to be an ordered sequence of the hull vertices which could be embedded in an array of slightly larger size.

Our basic strategy also implies equally efficient algorithms for the following problems – two- and three-dimensional convex hulls and trapezoidal decomposition which achieve a running time of $O(\log n/\log k)$ with kn processors in a CRCW PRAM. These in turn imply similar algorithms for two-dimensional Voronoi diagrams and triangulation of simple polygon. The algorithms extend the randomized divide-and-conquer techniques used in [24, 23]. We will be somewhat terse in our description of the algorithms and we will focus more on the technical portions that will be crucial for the analysis.

The rest of the paper is organized as follows. We begin by reviewing some consequences of padsort in a formal setting and present a direct application of padsort in developing an optimal parallel algorithm for the 2-D dominance problem. Following this, we review a general randomized divide and conquer strategy which forms the basis of our algorithms. Next, we present applications of the general strategy to individual problems starting with 2-D convex hulls.

Notation. We will use the notation $\tilde{O}$ in place of O to denote that the upper bound holds in the expected sense and moreover the probability of exceeding the expected value by a constant factor c decreases as n^{-c} where n is the input size. We will also refer to such bounds as high-probability bounds.

4. Padded sorting and parallel algorithms

A crucial factor in the performance of the padsort algorithm is the size of the output array m or more specifically the ratio m/n. If $m = (1+\lambda)n$ then λ is called the *padding factor*. A slightly weaker version of the main result of Hagerup and Raman can be stated as

Theorem 4.1. *Given n elements from an ordered universe, these can be padded-sort with kn* CRCW *processors in $\tilde{O}(\log n/\log k)$ time with a padding-factor $\lambda \leqslant 1/\log n$. Moreover, between any $\log n$ consecutive input keys, there is no more than one empty cell in the output array.*

A nice consequence of Theorem 4.1 is ordered searching. The output of the padsort algorithm makes it almost directly applicable to search for predecessor of a given key value. One simply probes the elements like a normal binary search except that when an empty cell is probed, we make an extra probe in the adjoining cell. By the consequence of Theorem 4.1, two adjacent cells cannot be empty. Alternatively, one may simply fill up the empty cells with the contents of the previous cell, and perform a usual binary search. The same holds true for any k-ary search. In summary

Lemma 4.2. *The output of the padded-sorting algorithm can be used for performing k-ary search on an n-element ordered array in* $O(\log n/\log k)$ *steps.*

Equipped with the above results, we obtain a fast parallel algorithm for finding the dominating set in plane from a set of n input points.

Algorithm Dominance

(0) Sort the given set of points in increasing order with respect to x coordinate.
(1) If the problem size is larger than a certain threshold, partition the problem into k (nearly) equal subproblems based on the x-coordinates and call steps 1–3 recursively. Else solve directly and also compute the maximum y-coordinate and then return.
(2) Let the maximum of the y coordinate in each of the intervals and denote them as Y_i, $1 \leqslant i \leqslant k$. To merge the subproblems, we compare the y coordinate of each element of the i-th subproblem with Y_j, $j > i$.
(3) For the surviving elements, (whose y coordinate is larger than Y_j's) we compute the maximum y-coordinate. (This is the element which has the least x-coordinate among the survivors.)

The analysis of this algorithm is quite straightforward. Each of the steps 1–3 can be performed in $O(1)$ time using kn processors. The partitioning in step 1 is trivially done and the maximum of a constant number of elements can be computed in $O(1)$ steps. Using k processors per element and concurrent read and writes, each element can find out if it survives in constant time in step 2. To find out which is the least element that survives in terms of x coordinate, we can use the result on finding the smallest index '1' element in a boolean array. This takes constant time using n processors (see JaJ'a [18, Example 2.13]). The recurrence for steps 2 and 3 can be written as

$$T(n) = T(n/k) + O(1),$$

which is $O(\log n/\log k)$. Note that only the first step is randomized so that the following theorem is almost an immediate consequence of the result of *padded sorting*. An alternate approach would have been to compute prefix-maximum of y-coordinates after sorting (step 0). We used this example to introduce the simplest form of k-way divide-and-conquer.

Theorem 4.3. *The dominating set of n points in a plane can be computed in* $\tilde{O}(\log n/\log k)$ *using* kn CRCW *processors and this is optimal.*

Processor allocation is a common problem that one encounters in most parallel algorithms. In this context, Hagerup [16] defines the problem of *interval allocation* as the following: Given n non-negative integers $x_1,\ldots,x_n$, allocate memory blocks of sizes $x_1,\ldots,x_n$ from a base segment of size $O(\sum_{j=1}^{n} x_j)$ such that the blocks do not overlap. Bast et al. [14] give a very fast algorithm for this problem which can be stated as

Lemma 4.4. *The interval allocation problem of size n can be solved in* $\tilde{O}(k)$ *time using* $n\log^{(k)} n$ CRCW PRAM *processors.*

We shall use this result for processor allocation in the context of our parallel algorithms especially as a substitute for exact prefix sums whenever we have to compute it faster than $O(\log n/\log\log n)$. Note that, in such cases the number of processors exceed $n\log n$ so that there is no problem in applying the previous lemma.

A common scenario for our algorithms is the following. Suppose s is the number of subproblems ($s < n$) and each of the input elements for the subproblems has been tagged with an index in $1,\ldots,s$. Then these can be sorted on their indices into an array of size $S(1+\lambda)$ from Theorem 4.1 where S is the sum of the sizes of the subproblems. A processor indexed P is associated with the element in the cell numbered $\lceil P/S\rceil$.[2] In most cases, $S=n$, so that if we have kn processors, then the number of processors allocated to a subproblem i of size s_i is at least $s_i k/(1+\lambda)$. The *processor advantage* which is defined to be the ratio of the number of processors to the subproblem size, is not as good as it was initially, namely it is $k/(1+\lambda)$ instead of k. However, for our purposes it will make little difference because of the property that the number of recursive levels in our algorithm will be bounded by $O(\log\log n)$. Hence, the processor advantage at any depth of the recursion is no worse than $k/(1+\lambda)^{O(\log\log n)}$ which is still $\Omega(k)$ for $\lambda \leqslant 1/\log n$. In our future discussions, we shall implicitly use this property for processor allocation.

5. Ultra-fast randomized divide-and-conquer

For a number of efficient algorithms in computational geometry, Reif and Sen [24, 23] had used a versatile approach based on randomized divide-and-conquer. We shall recapitulate the main general steps of their strategy for the problems under consideration

(1) Select $O(\log n)$ subsets of random objects (in case of 2-D hulls these are halfplanes) each of size $\lfloor n^{\varepsilon}\rfloor$ for some $0 < \varepsilon < 1$. Each such subset is used to partition the original problem into smaller subproblems. A sample is 'good' if the maximum subproblem size is less than $O(n^{1-\varepsilon}\log n)$ and the sum of the subproblem sizes is less

[2] We will avoid using the ceiling and floor functions when it is clear from the context.

than $\bar{c}n$ for some constant $\bar{c}$. From the probabilistic bounds proved in [11,24], it is known that the first condition for a 'good' sample holds with high probability. From here it follows that the sum of the subproblems is no more than $\tilde{O}(n \log n)$. However, the second condition which bounds the blow up in the size of the subproblems by a constant factor is known to hold with probability about $\frac{1}{2}$.

(2) Select a sample that is 'good' with high probability using *Polling*. At least one of the $\log n$ samples in the previous case is 'good' with high probability. *Polling* [23] is a sampling technique which allows us to choose a 'good' sample efficiently. This high probability bound is crucial to bound the running time of the algorithms by $O(\log n)$.

(3) Divide the original problem into smaller subproblems using the 'good' sample. The maximum size can be bound by $O(n^{1-\varepsilon} \log n)$.

(4) Use a *Filtering* procedure to bound the sum of the subproblem sizes by some fixed measure like the output size or input size. The reason for this being that the probabilistic bound in step (1) bounds the sum of the subproblems by $\bar{c}n$. If this increase by a multiplicative constant continues over each recursive stage, after i stages, the input size will have increased by a factor of $2^{\Omega(i)}$. If i is large (that is larger than a constant), then the parallel algorithm becomes somewhat inefficient affecting the processor time product bound. This *filtering* procedure is problem dependent and uses the specific geometric properties of a problem.

(5) If the size of a subproblem is more than a threshold, then call the algorithm recursively else solve it using some direct method. At this stage the subproblem sizes are so small – typically $O(\log^r n)$ for some constant r, that relatively inefficient methods work well.

The procedure used for partitioning the problem to a set of subproblems can often be reduced to point location in arrangements of hyperplanes, namely, using a locus-based approach. In the locus-based approach to partitioning the problem, each region of the arrangement is labeled with a set of subproblems it spawns (see [23]). In the Dobkin–Lipton method of searching, then this reduces to searching in ordered lists and the preprocessing reduces to sorting (padsort suffices). The following result is a corollary of the the above observations.

Lemma 5.1. *Given h hyperplanes in d dimensions, a data structure for point location can be constructed in $\tilde{O}(d \log n / \log k)$ time using kn^{2^d-1} processors. This data structure can be used to do point location in $O(d \log n / \log m)$ steps using m processors for each point.*

Even though the processor complexity grows exponentially, for small (fixed) dimension, this approach can be used effectively.

Polling is a randomized technique for selecting a 'good' sample from the $O(\log n)$ subsets. The *Polling lemma* [23] guarantees that with high probability we can choose a good sample using this method. Since the test for 'goodness' is carried out independently for each of the sample, this part of the algorithm is inherently parallelizable

even on the networks. We select that sample which gives us the smallest (estimated) blow-up in the problem size.

While Polling controls the blow-up within a constant factor, say α (the second condition for a 'good' sample), over j levels this could grow up to $\Omega(\alpha^j)$. For any non-constant j this could be significant. Hence, we need to further control the blow-up (to unity) which is achieved during this step which is done in the Filtering phase. Note that Filtering becomes redundant once the processor advantage exceeds $\log^{\Omega(1)} n$ from the simple observation that the depth of recursion is bounded by $\log\log n$.

In the remaining section, we shall look closely at a recurrence relation whose solution will be the crux of our analysis of the algorithms that follow in the next section. This is a generalization of the recurrence used by Reif and Sen for the special case, $k = 1$. We shall assume that the number of processors is kn with $k > \log^{\Omega(1)} n$. For k less than this we will outline suitable modifications. Let $T(n, m)$ represent the parallel running time for input size n with m processors:

$$T(n, nk) = T\left(\frac{n}{(nk)^{1/c}}, \frac{nk}{(nk)^{1/c}}\right) + a \log n / \log k.$$

Here c and a are constants larger than 1. The recurrence arises from the following property of our algorithms – when $m = nk$, the maximum subproblem size is no more than $n/(nk)^{1/c}$ with the processor advantage still k. Each recursive call, that is the divide step, takes no more than $O(\log n / \log k)$ time. The constant c is such, that given n^c processors, one can solve the problem in constant time. For example, in the maxima problem, c is no more than 2 since one can determine, using n processors per point, if it is a maximal point.

The reader can verify that the solution of this recurrence with appropriate stopping criterion is $O(\log n / \log k)$ by induction.

In our algorithms, we sample roughly $(nk)^{1/c}$ input elements which we use to partition the problem. From our earlier discussion the maximum subproblem size is no more than $n/(nk)^{1/c}$ (actually we are ignoring a logarithmic factor which can be adjusted by choosing slightly larger sample) with high likelihood. Moreover, we shall show how to achieve the partitioning including Polling and Filtering in $\tilde{O}(\log n / \log k)$ steps.

Technically, we cannot use a deterministic solution of this recurrence directly for our purposes as our bounds are probabilistic. So we use a technique which is a simple extension of the solution outlined in [22]. View the algorithm as a tree whose root represents the given problem (of size n) and an internal node as a subproblem. The children of a node represents the subproblems obtained by partitioning the node (by random sampling) and the leaves represent problems which can be solved directly without resorting to recursive calls.

Denote the time taken at a node at depth i from the root by T_i. It can be shown that T_i satisfies the following inequality:

$$\Pr[T_i \geqslant ac\alpha\varepsilon^i \log n / \log k] \leqslant 2^{-\varepsilon^i \log nc\alpha},$$

where a, c are constants and α a positive integer. Then extending the proof in [22], we obtain the following.

Lemma 5.2. *If all the leaf nodes of the tree representing the algorithm terminate within T steps, then* $\Pr[T \geqslant \alpha \log n / \log k] \leqslant n^{-f\alpha}$, *where f is a constant.*

See the appendix for a proof. In other words, the algorithm terminates in $O(\log n / \log k)$ time with very high likelihood.

6. Optimal speed-up sublogarithmic algorithms

In this section, we apply the methods developed in the previous sections to obtain very fast algorithms for a number of problems in computational geometry. We shall discuss only one of them, namely the two-dimensional convex hull more extensively and omit the details for the other problems.

6.1. Two-dimensional convex-hulls

Given a set N of n points in two dimensions we would like to compute the convex hull of these points. For convenience, we shall assume that we are solving the dual problem, that is, computing the intersection of half-planes in two dimensions which are represented by linear inequalities. We will use $C(N)$ to denote the intersection of the N half-planes.

Following the general strategy discussed in the previous section, we choose a sample S of half-planes and construct their intersection. For example, if we sample $O((nk)^{1/4} \log n)$ $(= s)$ half-planes then we can compute all the $O(s^2)$ pairwise intersections using s^2 processors. Then check which of them lie within the intersection using s processors per point in $O(1)$ time. Hence, with $O(s^3)$ or nk processors, we can determine the vertices of the intersection. Sorting these points gives a standard representation of the convex region $(C(S))$. By using padsort this can be done in $\tilde{O}(\log n / \log k)$ steps.

For the remaining set of $N - S$ half-planes, we determine how they intersect with $C(S)$. This is more easily done if we partition $C(S)$ into triangular sectors and then determine where the lines defining the half-planes intersect the sectors. Note that each half-plane could intersect more than one sector (in fact an arbitrary number of sectors). Denote by N_i the half-planes intersecting sector i. As a consequence of the random sampling lemmas, for all i, $N_i = \tilde{O}(n/(nk)^{1/4})$. To determine which sectors a half-plane intersects, we can use Chazelle and Dobkin's [10] *Fibonacci Search* which is easily modified to a k-ary search. It actually yields the intersection points of a line (defining the half-plane) and the convex region $C(S)$. From here one can easily determine the set of sectors the half-plane intersects. For polling, the number (cardinality) of sectors suffices.

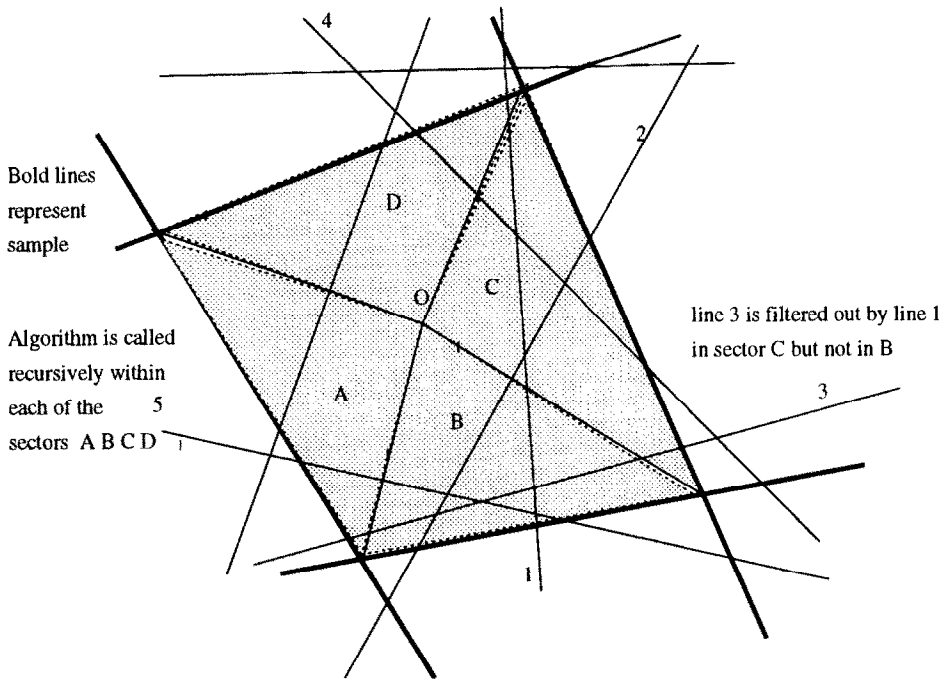

Fig. 1. Illustration of the basic divide-and-conquer strategy for computing intersection of half-planes.

To apply Polling, one actually selects $O(\log n)$ random subsets and repeat the above procedure on a large fraction (about $O(n/\log^3 n)$) of the $N - S$ half-planes to select a 'good sample'. Once the sample is selected, the problem is partitioned using the the locus-based approach mentioned previously. This is a more general method which is applicable to other problems unlike the *Fibonacci search*. Consider the duals of the vertices of the $C(S)$. The arrangement of these lines in the dual space induces a partitioning such that a (dual of) point in a fixed region intersects the same set of sectors of $C(S)$. Hence, the locus-based approach of the previous section is applicable directly in dimension two. This affects the size of the sample we choose initially as there is a big blow up in the number of processors required for preprocessing in Dobkin–Lipton algorithm. Hence, we will choose $s = O((nk)^{1/6})$ but that will still allow application of Lemma 5.2.

Next we will apply the filtering to further control $\sum_i N_i$ which is now $\tilde{O}(n)$. Recall that when $k > \log n$ we can actually skip this phase. After this step, we are left with at most one copy of a half-plane that does not show up in $C(N)$, that is a total of $2n$. The filtering step works as follows. For each sector i, one computes the intersections of the half-planes in N_i with the radial boundaries of the sector. Let $L(N_i)$ and $R(N_i)$ represent these intersections and let $\bar{L}(N_i)$ ($\bar{R}(N_i)$) represent the ranks of the sorted sequence in the radial direction (distance from origin). So each half-plane is now associated with a tuple – the left and right ranks. We now

determine the maximal half-planes in each sector using the algorithm of Section 2. Clearly, the half-planes that are not maximal would not form a part of the output inside the sector and we can discard these. We attach one processor to each half-plane that contributes to one vertex in a sector and two processors otherwise. The former condition is determined easily by checking if it is visible in exactly one of the (radial) boundaries. During further recursive calls, this processor allocation strategy ensures that the number of processors is proportional to the output complexity within each of the subproblems and we have sufficient processors. Following filtering we call the algorithm recursively within each sector.

For analyzing the algorithm we see that each of the phases can be carried out in time $\tilde{O}(\log n/\log k)$ and hence Lemma 5.2 can be applied to yield a running time of $\tilde{O}(\log n/\log k)$. Moreover, the final convex hull is obtained as a sorted sequence of vertices in an array which could have some empty cells like the padded-sort algorithm.

Theorem 6.1. *The convex hull of n points in a plane can be computed in* $\tilde{O}(\log n/\log k)$ *steps in a kn processors* CRCW PRAM. *The output of this algorithm is an ordered set of the hull vertices in an array of slightly larger size.*

Remark. For the case when $k \leqslant \log n$, the output of the algorithm is exact, that is the output vertices appear in a compact sorted array.

6.2. 3-D Convex hulls and 2-D Voronoi diagrams

An almost identical approach works for computing the convex hull of points in three-dimensions – where the vertices of the convex hull is produced as the output. We actually compute the intersection of half-spaces in three dimensions once we know a point in the (non-empty) interior. We do encounter some difficulty in the Filtering step (see [23] for details) where we need to build a data structure for detecting intersections of half-planes with a convex polytope. Instead, we use a simpler scheme due to Amato et al. [5]. The Filtering step is essentially locating the half-spaces that are *pinned* to a pyramid (see [5] for details). The scheme involves detecting the closest vertex of a convex chain from a line that can be done in $O(\log n/\log k)$ steps using the k-ary search. We use compaction to discard the redundant half-spaces within a pyramid and processor allocation is done using the observations of Reif and Sen [23]. Skipping further details, we state the following result.

Theorem 6.2. *The convex-hull of n points in three dimensions can be constructed in* $\tilde{O}(\log n/\log k)$ *steps by kn* CRCW PRAM *processors for* $k \geqslant 1$.

Note that the output of the algorithm produces a list of vertices in an array of slightly larger size. To compute the adjacency information (the edges), we require an application of integer sorting. The sorting is done on the list of 3-tuples (the

planes intersecting at that point) and we have all the six permutations corresponding to a tuple. If the adjacent tuple have two planes in common, they define an edge.

As a consequence of the 'lifting' transformation, it implies an identical bound for 2-D Voronoi diagram. Here the output is the list of the Voronoi vertices with their adjacency information. A Voronoi diagram without an associated point-location data structure is hardly of any use. We do address this issue in the next subsection, where we show how to build a point-location data structure in the same bounds.

6.3. Trapezoidal decomposition and triangulation

The problem of trapezoidal decomposition is a version of the vertical visibility problem. Given n non-intersecting (except at end-points) segments, one has to determine for each end-point, which segment lies immediately above it, that is find the first segment intersected by a upward vertical ray. Reif and Sen [24] describe an algorithm which has the same basic structure as the previous algorithms. The modification we require is in the first step – that is, for building the data structure for point location in a trapezoidal map of s randomly chosen segments. We substitute the *Cascaded Merging* technique of [3] (which requires a fractional cascading data structure) by the simpler point-location data structure of Dobkin and Lipton [13]. This also simplifies the algorithm of Reif and Sen [24]. The Filtering step is simply compaction – the reader is referred to [24] for details. So we have the following result.

Theorem 6.3. *The trapezoidal decomposition of n non-intersecting segments can be constructed in $\tilde{O}(\log n/\log k)$ steps using kn* CRCW PRAM *processors.*

Combining this with a result of Yap [30], where he reduces the triangulation of a simple polygon to two calls of trapezoidal decomposition (one vertical and one horizontal) we obtain the following corollary.

Corollary 6.1. *The triangulation edges of a simple polygon on n vertices can be determined in $\tilde{O}(\log n/\log k)$ steps using kn* CRCW PRAM *processors.*

Moreover, we can use a similar approach for building a point-location data structure for a planar subdivision. Actually, a by-product of the trapezoidal decomposition is a data structure that supports vertical ray shooting and hence one can do point location (see [3, 24]) using this scheme.

Corollary 6.2. *Given a planar subdivision of size n, a data structure for point location can be constructed in $\tilde{O}(\log n/\log k)$ steps using kn* CRCW PRAM *processors. This data structure supports point location by z* CREW *processors in $O(\log n/\log z)$ steps, where z is an integer greater than* 1.

7. Conclusion

We have presented a unified approach to speeding up various algorithms in computational geometry. Our method relies heavily on the results on *padded sorting* and exploit the generic randomized divide-and-conquer techniques of [26]. In addition, we have demonstrated that these are the best possible in a fairly strong sense, namely average speed-up. Our algorithms can be made somewhat stronger by making the running time hold with probability $1 - 2^{n^{\varepsilon}}$ for some $\varepsilon > 0$ instead of the standard high probability bounds derived in the paper.

This paper leaves open various directions for further research, the most significant being matching deterministic algorithms.

Regarding lower bounds, it will be interesting to extend these to the *algebraic computation model* where we could obtain matching lower bounds for a number of processors exceeding a certain threshhold. We suspect that these are not tight but it is not clear if the techniques used in this paper can be extended further. Our algorithms do not match the lower bounds for small output instances for which one may be able to obtain better speed-up, namely $O(\log h/\log k)$ where h is the output size.

The super-linear processor algorithms in this papers have found applications in obtaining faster output-sensitive parallel algorithms for convex hulls [15].

Acknowledgements

The author wishes to thank S. Kapoor for suggesting the use of Milnor–Thom-like results for the lower bounds and P. Agarwal for pointing out Ref. [25] and other helpful comments.

Appendix

Proof of Lemma 2.5. In the proof, we will use the following approach. We will start with a balanced tree $\mathscr{T}$ of height h for the given weight w. Then we shall show that any unbalanced tree of the same weight must have a larger average height.

An unbalanced tree T of the same weight will have leaf nodes at depth less than h and greater than h, where depth of a node is its distance from the root. For a leaf-node A at depth i ($i < h$), the corresponding subtree at A in $\mathscr{T}$ has a^{h-i} leaf-nodes at depth h, where a is the arity of the tree. If T has to match the weight of $\mathscr{T}$, then this 'weight-loss' has to be compensated for by leaves at depth greater than h. This is the way we view it, i.e., the leaves in T at depths less than h have the corresponding subtree (in $\mathscr{T}$) missing whose weights are made up of leaves at depths greater than h. More precisely, for leaf A at depth i, we have to make up for a weight $a^{h-i} \cdot h^d - i^d$.

So we can group with leaf A leaves at depth $h_1, h_2, \ldots, h_l$ that will attain this weight. Notice that we can do this groupings for all leaves at depth less than h. However, some of these groups could be overlapping because the same node N (at depth greater than h) may be compensating fractionally for more than one group. In the remainder of this proof, we shall show that the average height of each group exceeds h which will imply the lemma.

Let there be c_i leaves at height h_i ($> h$) compensating for A at height i. At most two c_i could be fractional – the first and the last. For weight compensation $c_1 h_1^d + c_2 h_2^d + \cdots + c_l h_l^d \geqslant a^{h-i} h^d - i^d$ where $h < h_1 < h_2$. We want to show that $\sum_i c_i h_i + i / \sum_i c_i + 1 > h$ or by rearranging, $\sum_i c_i (h_i - h) > h - i$. It suffices to show that $\max\{c_i\} > h - i$. Let c_m be the maximum of c_i. Clearly, $c_m \geqslant a^{h-i} h^d - i^d / h_1^d + h_2^d \ldots h_l^d$. If $l > 2h - i + 1$, then clearly the average height of this group exceeds h. So we will only consider $l \leqslant 2h - i + 1$:

$$c_m \geqslant \frac{a^{h-i} h^d - i^d}{(h - i + 1)(2h - i + 1)^d}. \tag{A.1}$$

Since $i^d / a^{h-i} < i^d$, inequality (A.1) can be written as

$$c_m \geqslant \frac{a^{h-i}(h^d - i^d)}{(h - i + 1)(2h - i + 1)^d}. \tag{A.2}$$

By minimizing $h^d - i^d$ and maximizing $(2h - i + 1)^d$ (use $i = h - 1$ and $i = 1$, respectively),

$$c_m \geqslant \frac{a^{h-i} d (h-1)^{d-1}}{(h - i + 1)(2h)^d} \geqslant \frac{a^{h-i}}{(h - i + 1)\, 3^d}$$

when $h < d$. For sufficiently large arity a, namely $a \approx (ek)^n$ where $e, k > 4$, $c_m > h - i$. □

Proof of Lemma 5.2. Setting $t_i = a(\varepsilon)^i \alpha (c - c_0) \log n / \log k$, where c_0 is some constant, we obtain

$$\Pr[T_i \geqslant a\alpha c(\varepsilon)^i \log n / \log k + t_i] \leqslant 2^{-(\varepsilon)^i c \alpha \log n} \leqslant 2^{-t_i \log k / a}.$$

If T is the total time for this worst-case chain of nested calls and $m = 1/(1 - \varepsilon)$, the probability that it takes more than $ma\alpha c_0 \log n / \log k + t$, is less than the sum of the probability of events where $\sum_i t_i = t$. Below, we calculate the probability that $\sum_i t_i = t$. $\prod_{\sum t_i = t} 2^{-t_i \log k / a} \leqslant \sum 2^{-t \log k / a}$ over $t^{O(\log(\log n / \log k))}$ tuples. Thus, $\Pr[T > am\alpha c_0 \log n / \log k + t] < 2^{-t \log k / a + O(\log t \log(\log n / \log k))}$. Using $t \geqslant am\alpha(c - c_0) \log n / \log k$, for large values of n and $m > 1$, we can rewrite the above expression as

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

For $c > 4c_0$, i.e. $c - c_0 > 3/4c$, we have the following required bound:

$$\Pr[T > \alpha \log n/\log k] \leqslant 2^{-(3/4)c\alpha \log n} \leqslant n^{-c_1\alpha}$$

assuming that a, m and c are larger than 1. □

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, Parallel computational geometry, in: *Proc. 25th Ann. Symp. on Foundations of Computer Science* (1985) 468–477. Also appears in full version in *Algorithmica* **3** (1988) 293–327.

[2] N. Alon and Y. Azar, The average complexity of deterministic and randomized parallel comparison-sorting algorithms, *SIAM J. Comput.* **17** (1988) 1178–1192.

[3] M.J. Atallah, R. Cole and M.T. Goodrich, Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Comput.* **18** (1989) 499–532.

[4] M.J. Atallah and M. Goodrich, in: J.H. Reif, ed., *Deterministic Parallel Computational Geometry* (Morgan Kaufman, Los Altos, CA, 1993).

[5] N.M. Amato, M.T. Goodrich and E.A. Ramos, Parallel algorithms for higher-dimensional convex hulls, in: *Proc. 35th Ann. FOCS* (1994) 683–694.

[6] Y. Azar and U. Vishkin, Tight comparison bounds on the complexity of parallel sorting, *SIAM J. Comput.* **16** (1987) 458–464.

[7] P. Beame and J. Hastad, Optimal bounds for decision problems on CRCW PRAMS, in: *Proc. 19th ACM STOC* (1987) 83–93.

[8] M. Ben-Or, Lower bounds for algebraic computation trees, in: *Proc. 15th STOC* (1983) 80–86.

[9] R.B. Boppana, The average-case parallel complexity of sorting, *Inform. Process. Lett.* **33** (1989) 145–146.

[10] B. Chazelle and D. Dobkin, Intersection of convex objects in two and three dimensions, *J. ACM* **34** (1987) 1–27.

[11] K.L. Clarkson, Applications of random sampling in computational geometry-II, in: *Proc. 4th Ann. ACM Symp. on Computational Geometry* (1988) 1–11.

[12] R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988) 770–785.

[13] D. Dobkin and R.J. Lipton, Multidimensional searching problems, *SIAM J. Comput.* **5** (1976) 181–186.

[14] M. Dietzfelbinger, H. Bast and T. Hagerup, A perfect parallel dictionary, in: *Proc. 17th Internat. Symp. on Math. Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 629 (Springer, Berlin, 1992) 133–141.

[15] N. Gupta and S. Sen, Optimal and output sensitive algorithms for constructing planar hulls in parallel, *Comput. Geom. Theory Appl.*, to appear.

[16] T. Hagerup, The log star revolution, in: *Proc. 9th Ann. STACS*, Lecture Notes in Computer Science, Vol. 577, (Springer, Berlin, 1992) 259–278.

[17] T. Hagerup and R. Raman, Waste makes haste: tight bounds for loose, parallel sorting, in: *Proc. 33rd Ann. FOCS* (1992) 628–637.

[18] J. JaJa, *An Introduction to Parallel Algorithms* (Addison-Wesley, Reading, MA, 1992).

[19] S. Kapoor and P. Ramanan, Lower bounds for maximal and convex layer problems, *Algorithmica* (1989) 447–459.

[20] P. MacKenzie and Q. Stout, Ultra-fast expected time parallel algorithms, in: *Proc. 2nd SODA* (1991) 414–423.

[21] U. Manber and M. Tompa, The effect of number of hamiltonian paths on the complexity of a vertex coloring problem, *SIAM J. Comput.* **13** (1984) 109–115.

[22] S. Rajasekaran and S. Sen, in: J.H. Reif, ed., *Random Sampling Techniques and Parallel Algorithm Design* (Morgan Kaufman, Los Altos, CA 1993).

[23] J.H. Reif and S. Sen, Optimal parallel randomized algorithms for 3-d convex hull, and related problems, *SIAM J. Comput.* **21** (1992) 466–485.

[24] J.H. Reif and S. Sen, Optimal randomized parallel algorithms for, computational geometry, *Algorithmica* **7** (1992) 91–117.
[25] M.F. Roy and R. Pollack, On the number of cells defined by a set of polynomials, *Comptes Rendus* **316** (1992) 573–577.
[26] S. Sen, *Random sampling techniques for efficient parallel, algorithms in computational geometry*, Ph.D. Thesis, Duke University, 1989.
[27] J. Steele and A.C. Yao, Lower bounds for algebraic decision trees, *J. Algorithms* (1982) 1–8.
[28] A.C. Yao, Probabilistic computation – towards a unified measure of complexity, in: *Proc. 18th Foundations of Computer Science* (1977) 222–227.
[29] A.C. Yao, A lower bound for finding convex hulls, *J. ACM* **28** (1981) 780–787.
[30] C.K. Yap, Parallel triangulation of a polygon in two calls to the, trapezoidal map, *Algorithmica* **3** (1988) 279–288.