MAX-PLANCK-INSTITUT FÜR INFORMATIK

Tight bounds for some problems in computational geometry: the complete sub-logarithmic parallel time range

Sandeep Sen

MPI–I–93–129 **July 1993**

INFORMATIK

Im Stadtwald
66123 Saarbrücken
Germany

Tight bounds for some problems in computational geometry: the complete sub-logarithmic parallel time range

Sandeep Sen

MPI-I-93-129 July 1993

Tight bounds for some problems in computational geometry : the complete sub-logarithmic parallel time range

(extended abstract)

Sandeep Sen *

July 9, 1993

Abstract

There are a number of fundamental problems in computational geometry for which work-optimal algorithms exist which have a parallel running time of $O(\log n)$ in the PRAM model. These include problems like two and three dimensional convex-hulls, trapezoidal decomposition, arrangement construction, dominance among others. Further improvements in running time to sub-logarithmic range were not considered likely because of their close relationship to sorting for which an $\Omega(\log n/\log\log n)$ is known to hold even with a polynomial number of processors. However, with recent progress in padded-sort algorithms, which circumvents the conventional lower-bounds, there arises a natural question about speeding up algorithms for the above-mentioned geometric problems (with appropriate modifications in the output specification). We present randomized parallel algorithms for some fundamental problems like convex-hulls and trapezoidal decomposition which execute in time $O(\log n/\log k)$ in an nk $(k > 1)$ processor CRCW PRAM. Our algorithms do not make any assumptions about the input distribution. Our work relies heavily on results on padded-sorting and some earlier results of Reif and Sen [28, 27]. We further prove a matching lower-bound for these problems in the bounded degree decision tree.

1 Introduction

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. One of the earliest work in this area is due to Chow [10], who developed algorithms for a number of fundamental problems which were deterministic and executed in inter-connection networks with polylogarithmic running time. A more general approach for deterministic PRAM algorithms was pioneered by Aggarwal et al. [1] who developed some new techniques for designing efficient parallel algorithms for fundamental geometric problems. A number of the most efficient deterministic PRAM algorithms are due to Atallah, Cole and Goodrich [3] who extended the techniques used by Cole [14] for his parallel mergesort algorithm. Their technique is called *Cascaded merging* and has been subsequently used (independently by Chandran [8]) for a number of other problems. Note that most of the geometric problems in the context of research in parallel algorithms have sequential time complexity of $\Omega(n\log n)$ and a typical performance that one aims to attain is $O(\log n)$ parallel time using an optimal number of processors.

In an independent development, Reif and Sen [28] were also able to derive $O(\log n)$ time optimal algorithms for point-location and trapezoidal decomposition which were randomized. Later in [27], they extended

*Present address : Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India. Work done when the author was visiting Max-Plank Institute for Informatik, Germany

their methods to give optimal algorithms for 3-D convex hulls (and hence 2-D Voronoi diagrams) on the *CREW* PRAM model. At the core of their algorithms were random sampling techniques which had also been introduced by Clarkson [11, 12, 13] and Haussler and Welzl [19]. In addition, a new resampling technique called **Polling** was used successfully to derive the parallel algorithms.

The randomized algorithms drew inspiration from the parallel sorting algorithm of Reischuk [30] and some of these algorithms were extended to the interconnection network model (without degradation of asymptotic complexity) in [26]. This can be viewed as being similar to the efforts of Reif and Valiant [29] who were able to adapt Reischuk's algorithm in the interconnection network model successfully although they had to resort to more sophisticated sampling techniques. Because of their close resemblance to the randomized sorting algorithms, the algorithms of [28, 27] appear to be more directly dependent on the present state-of-art of the complexity of randomized parallel sorting. With recent results in the area of *padded-sorting* (to be referred to as padsort in future), one is tempted to conjecture that these must have some consequences in the area of geometric problems. Of course, like padsort, the output specification of these problems have to be suitably modified to circumvent the lower bound of $\Omega(\log n/\log\log n))$ for input size n using polynomial number of processors (Beame and Hastad [5]). Roughly speaking, the problem of padsort involves ordering the input of size n into an output array of size $m \geq n$. When $m = n$, (or actually m is very close to n) the lower-bound of Beame and Hastad applies. This problem was first introduced by MacKenzie and Stout [22] and recently Hagerup and Raman [18] showed that one can padsort n elements with kn processors in 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, 4, 7] for the parallel-comparison tree model. These imply that the running time of any comparison based parallel algorithm for padsort is $\Omega(\log n/\log k)$ using kn processors.

To take advantage of the developments in padsort, we will modify the output specifications of the problems relevant to this paper. For example, for two-dimensional convex hulls we will relax the output to be an ordered sequence of the hull vertices which could be embedded in an array of slightly larger size. The previous lower-bound on padsort would imply a similar lower bound for this version of the convex hull problem. Even by relaxing the constraint of an ordered output, we prove a matching lower-bound for any reasonable output specification of the convex hull, namely identifying the hull vertices.

In this paper, we present 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 bound for three dimensional convex-hull holds for $k > \log n$. Since the algorithms resemble those in [28, 27], we will be somewhat terse in our description and focus more on portions that will be crucial for the analysis. We will encourage the reader to refer to the previous papers for more details of the individual algorithms for specific problems.

The rest of the paper is organized as follows. We begin by reviewing some of the consequences of padsort in a more formal setting. Then we illustrate the utility of padsort on a simple example where the results on padsort can be applied almost directly to obtain a fast algorithm. In section three we review a general randomized divide and conquer strategy which forms the backbone of our algorithms. In section four, we give details of the implementations of the general strategy for the individual problems. We conclude by proving a matching lower bound for some of these problems on the fixed-degree algebraic decision-tree model.

2 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 2.1 *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 $\lambda \leq 1/\log n$. 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 2.1 is to 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 consequence of Theorem 2.1, 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 2.1 *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 can design 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 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, $1 \leq i \leq k$.
3. To merge the subproblems, we compare the y coordinate of each element of the i-th subproblem with Y_j, $j > i$. For the surviving elements, (whose y coordinate is larger than Y_j's) we compute the maximum y-coordinate. This should be 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. We discuss only step 3. With k processors per element and concurrent read and writes, each element can find out if it survives in constant time. To find out which is the least (in terms of x coordinate) element that survives, we can use the result on finding the smallest index '1' element in a boolean array. This takes constant time using n processors (see Ja'Ja' [20], Ex 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 is almost an immediate consequence of the result of *padded sorting.*

Theorem 2.2 *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.*

Note that if we require our output to be the 'staircase' in a sorted order then this algorithm achieves optimal speed-up. However, we will establish the stronger notion of optimality which is independent of the ordering criterion in section 5.

Processor allocation is a common problem that one encounters in most parallel algorithms. In this context Hagerup [17] defines the problem of *interval allocation* as the following; Given n non-negative integers $x_1..., x_n$, allocate memory blocks of sizes $x_1.., x_n$ from a base segment of size $O(\sum_{j=1}^{n} x_j)$ such that the blocks don't overlap. Bast et al. [16] give a very fast algorithm for this problem which can be stated as

Lemma 2.2 *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 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 processors exceed $O(n\log n)$ 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 $1...s$. Then these can be sorted on their indices into an array of size $S(1+\lambda)$ from the previous theorem where S is the sum of the sizes of the subproblems. A processor indexed P is associated with the element in the cell numbered $\lceil P/S \rceil$. [1] 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 $s_i \cdot k/(1+\lambda)$. The processor advantage (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 $O(\log n/\log k)$. Hence the processor advantage at any depth of the recursion is no worse than $k/(1+\lambda)^{O(\log n/\log k)}$ which is still $\Omega(k)$. In our future discussions, we shall implicitly use this property for processor allocation.

3 Fast randomized divide-and-conquer

For a number of efficient algorithms in computational geometry, Reif and Sen [28, 27] had used a versatile approach which can be called 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 were half-planes) each of size $\lfloor n^\epsilon \rfloor$ for some $0 < \epsilon < 1$. Each such subset is used to partition the original problem into smaller sub-problems. A sample is 'good' if the maximum sub-problem size is less than $O(n^{1-\epsilon}\log n)$ and the sum of the sub-problem sizes is less than $\bar{c}n$ for some constant $\bar{c}$. From the probabilistic bounds proved in [13, 28], it is known that the first condition for a 'good' sample holds with high probability. From here it follows that the sum of the sub-problems 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 with probability about 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* [27] 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 sub-problems using the 'good' sample. The maximum size can be bound by $O(n^{1-\epsilon}\log n)$.
(4) Use a *Filtering* procedure to bound the sum of the sub-problem sizes by some fixed measure like the output size or input size. The reason for this being that the probabilistic bounds in step (1) bounds the sum of the sub-problems 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 geometry properties of a problem.
(5) If the size of a sub-problem is more than a threshold, then call the algorithm recursively else solve it using some direct method. At this stage the sub-problem sizes are so small (typically $O(\log^r n)$ for some constant r) that relatively inefficient methods work well.

The procedure used for dividing the sub-problems can often be reduced to point location in arrangements of hyperplanes, namely using a locus based approach. If one uses the Dobkin-Lipton method of searching,

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

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

In the locus based approach to partitioning the problem, each region in the arrangement gives rise to a set of elements which are labeled by the sub-problem they belong to. Each region in the arrangement is preprocessed, to determine its (unique) associated subproblems. Even though the processor complexity grows exponentially, for small (fixed) dimension, this approach can be used effectively. Note that even to 'read' the set of subproblems for a number of points, we have to solve a processor allocation problem; for this we shall use the results on interval allocation stated in the previous section. Assume that we have kept a count of the number of subproblems associated with each region. This can be done easily during the preprocessing stage by 'compressing' a bit vector. Now we run the algorithm for interval allocation on the counts associated with each point. This enables us to write the set of subproblems associated with each interval. We next sort them so that processor allocation can be done using the observation of the previous section. Note that although the total size of the intervals following the interval allocation algorithm can blow up by a constant factor, an application of padsort reduces that considerably (no more than the padding-factor).

Polling involves selecting $O(n/\log^2 n)$ input objects to test a sample instead of testing a sample with respect to the entire input set. Since there are $O(\log n)$ subsets, this saves the extra work we would have to do if we tested the 'goodness' of the sample on the entire input. The *Polling lemma* [27] 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. To each of the $O(\log n)$ sample that we apply polling, we use the locus-based approach described before to test the 'goodness' of the sample. We simply select that sample which gives us the smallest (estimated) blow-up of the problem size. We also make a note that although Polling appears crucial to obtaining optimal bounds when number of processors is about n, it is no longer so when processors exceed about $n2^{\log\log^2 n}$. That is because any sample (with high probability) does not blow up the size of the subproblems by a factor of $O(\log n)$. Since the depth of the recursion is bounded by $O(\log\log n)$, the cumulative blow-up is no more than the mentioned value. By observing that $O(\log n/\log k)$ is asymptotically the same as $O(\log n/\log(k/2^{\log\log^2 n})$ for k exceeding $O(n\log\log^2 n)$, we can dispense with polling for larger number of processors.

Perhaps the step that is most specific to a problem is the *Filtering* step where we have to use some geometric properties of the problem. While Polling controls the blow up by a constant factor at each recursive call, there could be blow up by a constant factor, say $\bar{c}$. Over j levels this could grow up to $\Omega(\bar{c}^j)$. 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. This step can only follow polling, as polling cuts down the problem size to $O(n)$ (instead of $O(n\log n)$). This step could be quite complicated for some problems (like the three dimensional convex hull). Again like our previous observation Filtering becomes redundant once the processor advantage exceeds $\Omega(\log n)$.

Hence as the processor advantage increases, that is, for larger values of k, our algorithms actually become simpler because we can dispense first with Filtering and subsequently Polling. This is contrary to the case $k \leq 1$ when algorithms become more complicated as processors increase and one tries to achieve optimal speed-up. The reason why this happens is because the speed-up is no longer linear in the number of processors.

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. 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.

$$T(n, nk) = T(\tfrac{n}{(nk)^{1/c}}, \tfrac{nk}{(nk)^{1/c}}) + a\log n/\log k$$

Here c and a are constants larger than 1. The reader can verify that the solution of this recurrence (with appropriate stopping criterion) is $O(\log n/\log k)$ by induction. A physical interpretation of this recurrence is that $T(n,m)$ represents parallel running time for input size n with m processors. When $m = nk$, the maximum subproblem size is no more than $\frac{n}{(nk)^{1/c}}$ with the processor advantage still k. Each recursive call (that is the divide step) takes no more than $O(\log n/\log k)$. 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). Our algorithms have a very similar property - 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 $\frac{n}{(nk)^{1/c}}$ (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.

Clearly, 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 [25]. 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 sub-problems 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

$$\text{Prob}[T_i \geq ac\alpha\epsilon^i \log n/\log k] \leq 2^{-\epsilon^i \log n c\alpha}$$

where a, c are constants and α a positive integer. Then extending the proof in [25], we obtain the following

Lemma 3.2 *If all the leaf nodes of the tree representing the algorithm terminate within T steps, then $Prob[T \geq \alpha \log n/\log k] \leq n^{-f\alpha}$. where f is a constant.*

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

4 Applications of k-way divide-and-conquer

In this section we apply the methods developed in the previous sections to obtaining 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 which are quite similar. The reader may refer to some previous work for further details of these.

4.1 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 (containing the origin) 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 $N - S$ half-planes, we determine how they intersect with $C(S)$. This is more easily done if we partition $C(S)$ into triangular sectors (see Figure 6) 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 [9] *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 of sectors suffice.

To apply Polling, one actually selects $O(\log n)$ random subsets and repeats the above procedure on a large fraction (about $O(n/\log^3 n)$) of the $N-S$ to select a 'good sample'. Once the sample is selected, the problem is partitioned using the procedure described in the previous section (locus-based approach). We describe below another alternative approach, that is the locus-based approach for problem partitioning. 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 arrangements of these lines in the dual space induce 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 3.2.

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 rank. 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 (see Figure 6). 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 number of processors is proportional to the output complexity within each of the subproblem 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 3.2 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 4.1 *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 $k \leq \log n$, the the output of the algorithm is exact, that is the output vertices appear in a compact sorted array. Moreover, we do not use padsort in the algorithm.

4.2 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 we 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 [27] for details) where we need to build a data structure for detecting intersections of half-planes with a convex polytope. For the range $1 < k \leq \log n$, this requires building this data-structure faster than $O(\log n)$ which is currently a bottle-neck. However, form our earlier remark, for $k \geq \log n$, we can dispense with Filtering and hence we can achieve the required speed-up.

Theorem 4.2 *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 $k \geq \log n$.*

As a consequence of the 'lifting' transformation, we obtain a similar bound for 2-D Voronoi diagram. Here the output is the list of the Voronoi vertices with their adjacency information.

4.3 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 [28] 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 [15]. This also simplifies the algorithm of Reif and Sen [28]. The Filtering step is simply compaction - the reader is referred to [28] for details. So we have the following result

Theorem 4.3 *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 [34], 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

Theorem 4.4 *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.*

5 Lower bounds

As mentioned earlier, some of our algorithms are optimal if we require the output to appear in a sorted order like the output vertices of the 2-D convex hull or the staircase of the maximas. In this section, we will further strengthen our results which will hold independent of such a rigid output specification. For example, we shall show that even identification of the convex-hull vertices require $\Omega(\log n/\log k)$ parallel time using kn processors which can be viewed as an extension of Yao's [33] observation in the sequential case. For this section, we shall use slightly modified versions (used previously in [21, 32]) of the convex-hull (dominance) problem where the issue is to determine if among a set of n points all the points belong to the convex-hull (maximas). Note that these versions are constant time reducible to the standard versions in a CRCW PRAM model with $p \geq n$ processors. The model of computation is the parallel analogue of *Bounded-degree decision tree model*(BDD Tree). At each node of this tree, each of the p processors computes the sign of a fixed degree polynomial and then the algorithm branches according to the *sign vector*, that is considering all the signs. The algorithm ends as we reach a leaf node which gives the answer. This is a stronger model than any CRCW PRAM model as it does not care about read-write conflicts and is not charged for branching decision time.

Our first lower-bound proof uses the approach by Boppana [7] who had earlier dramatically simplified the lower-bound proof of [2] on parallel-sorting. An useful consequence of our result is that the lower bound on sorting also extends to this model. We will first review Boppana's elegant proof technique which establishes a bound on the average-case complexity of parallel sorting and consequently any randomized algorithm for the worst-case.

Fact 1 *In a parallel comparison (BDD) tree of l leaves and maximum arity a, the average path-length is at least $\Omega(\log l/\log a)$.*

Given this fact (credited to Shannon), one needs a reasonably tight upperbound 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 and Bopanna used a result of Manber and Tompa [23] which bounds the number of *acyclic orientations* of an undirected graph to $(1 + 2m/n)^n$ where n and m represent 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 should 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 $\Omega(\log n / \log(p/n))$.

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. Maximising product of the cardinalities of the two acyclic orientations is an upper-bound on the arity and this is less than $(1 + p/n)^{2n}$. It is known that for the n-input dominance problem the number of leaves is $\Omega((n/2)^{(n/2)})$ ([21]).

Lemma 5.1 *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 further strengthen our results to hold in the BDD Tree model. Also note that comparison tree model is not a meaningful computing model for most problems in geometry like the convex hulls. For this, we will first prove a worst case bound along the lines of Ben-Or [6] and subsequently extend it to the average case. The additional complication presented 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 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 Broy [24]. to bound the number of such possibilities

Lemma 5.2 *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 compopents associated with such a node is $((O(bnp/d))^d$. In our context, the number of processors and (hence the polynomial signs computed at each stage) is bound 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 5.1 *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 ($k \geq 1$) 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 5.2

$$(ekn/n)^{hn} \cdot (ehkn/n)^n \geq |W|$$

where e is a constant that subsumes the degree of the polynomials. The first expression on the l.h.s. represnts maximum number of leaves and the second expression is the maximum number of connected components associated with a leaf at depth h. By simple manipulations and using $hn > h \log n$ we arrive at the required result. □

The above theorem immediately yields as corollary $\Omega(\log n / \log k)$ worst-case bound for a number of problems for which $|W|$ at least $(n/2)^{(n/2)}$. This includes sorting, dominance and the convex hull problems ([32, 21]).

To extend the above result to the average case we require a mild assumption about the algorithms. We shall restrict the parallel algorithms to be efficient, that is the worst-case time bound is polylogarithmic. This implies that the longest path in the parallel BDD tree is bounded by some $\log^f n$ for some constant f. This is not unreasonable as there exists deterministic algorithms with polylog running time using only n processors for all our problems. We can then bound the number of leaves of the BDD tree to be $\Omega(|W|/(enkL/n)^n)$ where L is the longest path to a leaf node. This yields a bound similar to the previous theorem.

Theorem 5.2 *Let $W \subset R^n$ be a set that has $|W|$ connected components. Then any parallel BDD tree algorithm which has a worst case polylog time complexity to decide membership in W using kn ($k \geq 1$) processors has average time complexity $\Omega(\log |W|/n \log k)$.*

Remark: This extra restriction on the worst case complexity is probably unnecessary - it is only to get around a nasty optimisation problem in the general lower bound proof.

Since $|W|$ is at least $(n/2)^{(n/2)}$. for the the two dimensional convex hull, the average running time at least $\Omega(\log l/n \log k)$ time. For sorting and the dominance problem, the same bounds hold. By a simple reduction of 2-D dominance to trapezoidal decomposition, we get a similar bound for the latter problem.

6 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 [31]. 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 $1 - 2^{n^\epsilon}$ for some $\epsilon > 0$ 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. We do not achieve optimal speed-up for 3-D convex hulls for processors in the range n to $n \log n$.

Regarding lower bounds, it will be interesting to extend these to the *algebraic model* which allows arithmetic computations ([6]). However, it appears that the presently known Milnor-Thom bounds are too weak for our purpose. Our algorihms 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.

Acknowledgement

The author wishes to thank S. Kapoor for suggesting use of Milnor-Thom like results for the lower bounds and P. Agarwal for pointing out the reference [23] and helpful comments.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Proc. of 25th Annual Symposium on Foundations of Computer Science*, pages 468 – 477, 1985. also appears in full version in Algorithmica, Vol. 3, No. 3, 1988, pp. 293-327.

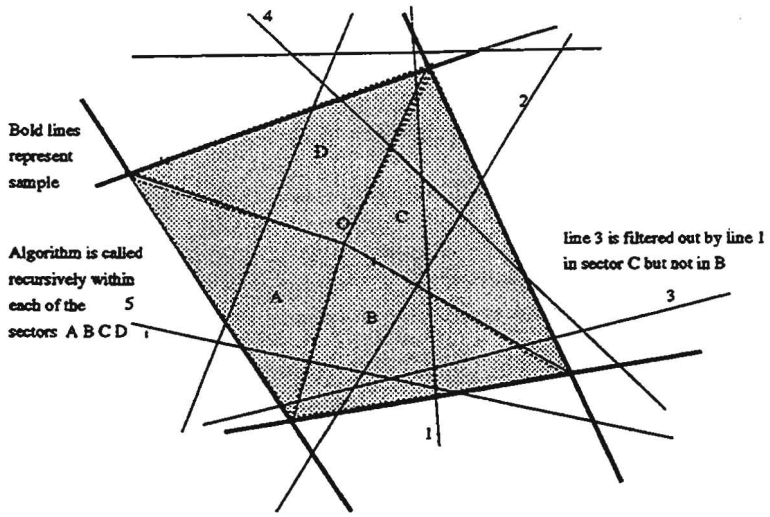

Figure 1: Illustration of the basic divide-and-conquer strategy for computing intersection of half-planes

[2] N. Alon and Y. Azar. The average complexity of deterministic and randomized parallel comparison-sorting algorithms. *SIAM Journal on Computing*, 17:1178–1192, 1988.

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

[4] Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM Journal on Computing*, 16:458–464, 1987.

[5] P. Beame and J. Hastad. Optimal bounds for decision problems on, crcw pram. *Proc. of the 19th Annual STOC*, pages 83 – 93, 1987.

[6] M. Ben-Or. Lower bounds for algebraic computation trees. *Proc. of the Fifteenth STOC*, pages 80–86, 1983.

[7] R.B. Boppana. The average-case parallel complexity of sorting. *Information Processing Letters*, 33:145–146, 1989.

[8] S. Chandran. *Merging in Parallel Computational Geometry.* PhD thesis, University of Maryland, 1989.

[9] B. Chazelle and D. Dobkin. Intersection of convex objects in two and three dimensions. *J.A.C.M.*, 34(1):1–27, 1987.

[10] A. Chow. *Parallel Algorithms for Geometric Problems.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[11] K.L. Clarkson. A probabilistic algorithm for the post-office problem. *Proc of the 17th Annual SIGACT Symposium*, pages 174 – 184, 1985.

[12] K.L. Clarkson. New applications of random sampling in computational geometry. *Discrete and Computational Geometry*, pages 195 – 222, 1987.

[13] K.L. Clarkson. Applications of random sampling in computational geometry ii. *Proc of the 4th Annual ACM Symp on Computational Geometry*, pages 1 – 11, 1988.

[14] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770 – 785, 1988.

[15] D. Dobkin and R.J. Lipton. Multidimensional searching problems. *SIAM J. on Computing*, 5:181 – 186, 1976.

[16] M. Dietzfelbinger H. Bast and T. Hagerup. A perfect parallel dictionary. *Proc. of the 17th Intl. Symp on Math, Foundations of Computer Science, LNCS 629*, pages 133– 141, 1992.

[17] T. Hagerup. The log star revolution. *Proc. of the 9th Annual STACS, LNCS 577*, pages 259 – 278, 1992.

[18] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose, parallel sorting. *Proc. of the 33rd Annual FOCS*, pages 628– 637, 1992.

[19] D. Haussler and E. Welzl. ε-nets and simplex range queries. *Discrete and Computational Geometry*, 2(2):127 – 152, 1987.

[20] Joseph JaJa: *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

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

[22] P. MacKenzie and Q. Stout. Ultra-fasttexpected time parallel algorithms. *Proc. of the 2nd SODA*, pages 414–423, 1991.

[23] U. Manber and M. Tompa. The effect of number of hamiltonian paths on the complexity of a vertex colouring problem. *SIAM J. COMPUT*, 13:109–115, 1984.

[24] R. Pollack and M.F. Broy. On the number of cells defined by a set of polynomials. *TR 618 Dept Computer Science NYU*, 1992.

[25] S. Rajasekaran and S. Sen. *Random sampling Techniques and parallel algorithm design*. J.H. Reif editor. Morgan, Kaufman Publishers, 1993.

[26] J.H. Reif and S. Sen. Randomized algorithms for binary search and load balancing o, fixed-connection networks with applications. *Proc. of the 2nd Annual SPAA*, pages 327 – 337, 1990. to appear in SIAM Journal on Comput.

[27] J.H. Reif and S. Sen. Optimal parallel randomized algorithms for 3-d convex hull, and related problems. *SIAM Journal on Computing*, 21:466 – 485, 1992.

[28] J.H. Reif and S. Sen. Optimal randomized parallel algorithms for, computational geometry. *Algorithmica*, 7:91 – 117, 1992.

[29] J.H. Reif and L.G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60 – 76, 1987.

[30] R. Reischuk. A fast probabilistic parallel sorting algorithm. *Proc. of the 22nd IEEE FOCS*, pages 212 – 219, 1981.

[31] S. Sen. *Random Sampling Techniques for Efficient Parallel, Algorithms in Computational Geometry*. PhD thesis, Duke University, 1989.

[32] J. Steele and A.C. Yao. Lower bounds for algebraic decision trees. *Journal of Algorithms*, pages 1–8, 1982.

[33] A.C. Yao. A lower bound for finding convex hulls. *Journal of the A.C.M.*, 28:780–787, 1981.

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