Optimal Parallel Randomized Algorithms for 3-D Convex Hulls and Related Problems

John H. Reif* and Sandeep Sen*
Computer Science Department
Duke University
Durham, N.C. 27706

Abstract

We present further applications of random sampling techniques which have been used for deriving efficient parallel algorithms by Reif and Sen [27]. In this paper we present an optimal parallel randomized algorithm for computing intersection of half-spaces in three dimensions. Because of well-known reductions, our methods also yield equally efficient algorithms for fundamental problems like the convex hull in three dimensions, Voronoi diagram of point sites on a plane and Euclidean minimal spanning tree. Our algorithms run in time $T = O(\log n)$ for worst-case inputs and uses $P = O(n)$ processors in a CREW PRAM model where n is the input size. They are randomized in the sense that they use a total of only $O(\log^2 n)$ random bits and terminate in the claimed time bound with probability $1 - n^{-\alpha}$ for any fixed $\alpha > 0$. They are also optimal in $P \cdot T$ product since the sequential time bound for all these problems is $\Omega(n \log n)$. The best known deterministic parallel algorithms for 2-D Voronoi-diagram and 3-D Convex hull run in $O(\log^2 n)$ and $O(\log^2 n \log^* n)$ time respectively while using $O(n/\log n)$ and $O(n)$ processors respectively.

[0]*Research supported in part by Airforce Contract AFSOR-87-0386, Office of Naval Research Contract N00014-87-K-0310, National Science Foundation Contract CCR-8696134, DARPA/ARO Contract DAAL03-88-K-0185, DARPA/ISTO Contract N00014-88-K-0458.

A preliminary version of this paper appeared as 'Polling: A New randomized sampling technique for computational geometry' in the 21st Annual STOC, Seattle, 1989

1 Introduction

1.1 Background and previous work

Designing efficient parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. After some early work by Chow [6] in her thesis, Aggarwal et al. [1] developed some general techniques for designing efficient parallel algorithms for fundamental geometric problems. Most of the problems tackled in that paper had $\Theta(n \log n)$ sequential complexity and the authors presented parallel algorithms which used a linear number of processors and ran in $O(\log^k n)$ time (k being typically 2,3 or 4) in size of the input. Consequently, a majority of the algorithms were not optimal in $P \cdot T$ bounds. A number of the problems in the original list (in [1]) have now been successfully resolved as far as $O(\log n)$ time, n processors algorithms are concerned, mainly due to work by Atallah, Cole and Goodrich [2]. They extended the techniques used by Cole [12] for his parallel mergesort algorithm and used a date-structure called *plane-sweep tree* (first proposed by Aggarwal et al. [1]) to arrive at the optimal algorithms. Perhaps the two most important problems which have been eluding such efforts are the 2-D Voronoi diagram problem and the convex hull of points in 3-space. These are very fundamental problems in computational geometry and optimal algorithms for these problems would imply corresponding optimal solutions for a multitude of other problems.

A very general definition of Voronoi diagram given by Edelsbrunner [19] is as follows:

> Let S be a finite set of subsets of E^d and for each $s \in$ S let d_s be a mapping of E^d to positive real numbers; we call $d_s(p)$ the distance function of s. The set $\{p \in E^d$: $d_s(p) < d_t(p)$, t $\in S$ - {s}$\}$ is the Voronoi cell of s and the cell complex defined by the *Voronoi cells* of all subsets in S is called the *Voronoi diagram* of S.

In this paper, we confine ourselves to the case where S is a set of points in E^2 and the distance function is the L_2 metric. In mathematical literature, Voronoi diagrams appeared as early as in 1850 (due to Dirichlet) and again in 1907 due to Voronoi. Problems about packing and coverings of space by balls and other convex figures were among the first major applications of such diagrams. Shamos and Hoey [32] introduced Voronoi diagrams to computer science and and since then a considerable amount of research has been devoted for deriving efficient sequential algorithms for the 2-D Voronoi diagram problem ([20, 30, 10]). The Voronoi diagram is a very versatile tool for obtaining efficient solutions of some important proximity problems and is also a fundamental mathematical object in its own right. A large number of the problems can be solved in linear or $O(n \log n)$ time from the information contained in the Voronoi diagram that includes *all-points nearest neighbor, Euclidean minimal spanning tree, diameter, smallest enclosing circle* among others. In section 6.1, we make use of this property to obtain efficient parallel algorithms for some of these problems.

Since there are sequential algorithms for Voronoi-diagrams that run in time $\Theta(n \log n)$, it is a fundamental question if there is a parallel algorithm that runs in $O(\log n)$ time using n processors.

Aggarwal et al. have given a $O(\log^2 n)$ time, $O(n)$ processors algorithm which was recently improved to $n/\log n$ processors by Cole et at. [26] (or alternately $O(\log n \log\log n)$ time and $O(n \log^2 n)$ work). However, it appears that one would require very different techniques to eliminate the $O(\log n)$ factor. Cole and Goodrich [13] reiterated the difficulties posed by this problem, when they provided some more applications of their cascaded-merging technique but were unable to extend it to the Voronoi diagram problem. In this paper we settle this question by presenting a randomized algorithm for this problem that runs in $O(\log n)$ time and uses n processors in a shared memory model of parallel computation. The reader should note that the lower-bound of $\Omega(n \log n)$ also applies to the randomized algorithms by a reduction of sorting to (1-dimensional) Voronoi-diagrams. Levcopoulos, Katajainen and Lingas [24] presented an optimal expected time algorithm for Voronoi diagrams for randomly chosen set of input points; in contrast our algorithm makes no assumption about the input distribution and is optimal for the worst-case input.

Convex hulls in 3-D has a wide range of applications ranging from computer graphics to design automation to pattern recognition to operations research. Convex hulls in three dimensions can also be constructed sequentially in $\Theta(n \log n)$ time where as the best known deterministic parallel algorithm due to Dadoun and Kirkpatrick [14] runs in $O(\log^2 n \log^* n)$ time using n processors. In this paper we actually describe an optimal randomized parallel algorithm for constructing convex hulls in Euclidean 3-space. Due to a well-known reduction from 2-D Voronoi diagrams to 3-D convex hulls, we get an equally efficient algorithm for the first problem as an immediate corollary.

1.2 Random-sampling and Polling in computational geometry

Randomization has been successfully used in a wide number of applications (for example see [21, 29, 33]) and has recently been used to obtain efficient algorithms in computational geometry. Clarkson [8, 9, 10], Haussler and Welzl [22], and Mulmuley [25] used random sampling techniques to derive better upper-bounds for a large number of problems including the post-office problem, higher-order Voronoi diagrams, segment intersections, linear programming and higher-dimensional convex hulls. The general approach taken by these algorithms is as follows: a randomly chosen subset R of the input set S is used to partition the problem into smaller ones. Clarkson [10] proved that for a wide class of problems in computational geometry, the *expected* size of each subproblem is $O(|S|/|R|)$ and moreover the *expected* total size of the subproblems is $O(|S|)$. A random subset R which satisfies these conditions for fixed constant multiples is called a 'good' sample and is called 'bad' otherwise. Clarkson's results show that by using a straight-forward random sampling technique any randomly chosen subset is good with constant probability; implying that it can also be 'bad' with constant probability. Consequently, his methods yielded *expected* resource bounds but cannot be used to obtain high-likelihood bounds (i.e. bounds that hold with probability $1 - 1/n^{\alpha}$ for any $\alpha > 0$). This makes it very difficult to extend his methods in the context of parallel algorithms due to the recursive nature of the algorithms. In particular the *expected* bounds at each

recursive call are not strong enough to bound the resources used by the entire parallel algorithm due to the following reason. In a sequential algorithm, due to the linearity property of expectation (i.e. the expectation of the sum is the sum of expectations), it suffices to bound the expected time required by individual steps. The total expected time of the sequential algorithm is the sum of expected time of the individual steps. In contrast, consider the recursive parallel algorithm as a tree where a node corresponds to a procedure and the children of a node corresponds to the parallel recursive calls made by the procedure. The time required at each level of this tree is the *maximum* of the time required by any node of that level. There is no known method to bound the maximum of the expectations without using higher moments. The total time required by the parallel algorithm is the time when all the procedures corresponding to the leaf nodes are completed. Typically, in a parallel algorithm, the number of leaves in the corresponding process tree is at least n^{ϵ} $(0 < \epsilon < 1)$. Even if we succeed in bounding the expected time for completion of a leaf-node procedure, the expected bounds are too weak to bound the *maximum* of the time required by all such processes.

The above problem can be dealt with by developing a technique for choosing samples that are 'good' (as defined above) with high probability. By doing so we shall show that a leaf-node process terminates in a given time bound with probability 1- $1/n^{\alpha}$ for any $\alpha > 0$. In particular, for $\alpha > 1$, this implies that the failure probability for the entire algorithm is less than $1/n^{\alpha-1}$ (since there can be at most $O(n)$ leaf-level processes). We introduce a technique called *polling* to obtain a 'good' sample with high probability with relatively small overhead. Roughly speaking, we choose a number, $p(n)$, of random subsets (typically $p(n) = O(\log n)$) independently. We then determine which of these subsets is 'good' and we show with high probability one of them is 'good'. This scheme, though effective, is not very efficient since we have to repeat the procedure $p(n)$ times. However, we show that we can draw conclusions about the 'goodness' of a sample very accurately by using only a fraction (typically $1/(p(n))$ of the input which then makes the Polling scheme very efficient. This is actually very similar to the idea of polling a small fraction of the population to find out how the entire population would behave and hence the name. This turns out to be crucial in bounding the total running time of the parallel algorithm. In addition to the applications in obtaining the improved results in this paper in computational geometry, Polling appears to be a general tool for obtaining improved parallel randomized algorithms. A similar idea had been used previously by Dyer and Frieze [18] to improve the efficiency of a linear programming algorithm.

Note that the second property of a 'good' sample i.e. that of bounding the total size of the sub-problems is not an issue in 1-dimensional problems. In the parallel sorting algorithms of Reischuk[20] and Flashsort[19] the total size of the subproblems always equals the input size. This is another reason why the straight-forward random sampling techniques do not carry over to the recursive algorithms. Clarkson[8] circumvents this problem by limiting the number of recursive levels by a fixed constant. By using recursion over $s(n)$ steps the problem size could grow by a multiplicative factor of $2^{\Omega(s(n))}$ if the sum of the subproblems increases by only a constant factor over the parent-problem at every recursive call. This could seriously affect the efficiency of the

algorithms, especially when we are looking for optimal algorithms. We need additional arguments to bound the total size of the sub-problems at any level of recursive calls (independent of the level number).

1.3 Main results

The main result in this paper can be summarized as following:
Theorem: *There exists a randomized algorithm in the CREW PRAM model for constructing the intersection of n half-spaces in three dimensions that runs in $O(\log n)$ time for any input with probability $> 1 - 1/n^{\alpha}$ (for any fixed $\alpha > 0$) using n processors. Moreover, we can also limit the total number of random bits used by our algorithm to $O(\log^2 n)$.*
The above theorem immediately implies equally efficient algorithms for the following problems from well-known reductions

(i) Convex-hull of points in 3-dimensions
(ii) Voronoi-diagram of point sites in a plane
(iii) Euclidean Minimal spanning tree

The previously best-known algorithms for all these problems are sub-optimal by at least an $O(\log n)$ factor in time complexity For the last problem we require a **Priority** CRCW PRAM model. In this model, the highest priority (fixed in advance) processor among any group of contending processors succeeds in the event of a write-conflict.

We adopt a top-down approach in describing the algorithm. In the section 2 we list some of the preliminary results that will be used as low-level procedures in the algorithm and some probabilistic notations used to aid the analysis. In section 3, we sketch a very high level description of the algorithm that uses the straight-forward random sampling (without polling) and if implemented in a straightforward manner wouldn't be very efficient. In section 4, we give a formal description of polling and its probabilistic analysis. In section 5 we describe an efficient procedure for for carrying out the divide step of the algorithm. In section 6, we present probabilistic arguments for bounding the total time of the algorithm with high likelihood and bound the number of processors needed at any single step to complete the analysis.

2 Some preliminary results and overview

2.1 Model of computation and notations

Throughout this paper we will be using the CREW PRAM model which is the synchronous shared memory model of parallel computation in which processors may simultaneously read from a memory location but are not allowed to write concurrently. At each step, a processor is allowed to perform a real-arithmetic operation consistent with standard models used for sequential geometric algorithms.

Moreover, each processor has access to a random-number generator that returns in unit time a truly random number of $O(\log n)$ bits. However, see in section 6.3, where we limit the use of truly random bits.

The term *very high likelihood (probability)* is used in this paper to denote probability $> 1 - n^{-\alpha}$ for some $\alpha > 1$ where n is the input size. Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, we shall use $\tilde{\mathrm{O}}$ to represent complexity bounds of the randomized algorithms. We say that a randomized algorithm has resource bound $\tilde{O}(f(n))$ if there is a constant c such that the resource used by the algorithm is no more than $c\alpha f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$. (An equivalent definition will be bounding the resource by $\alpha \cdot f(n)$ with probability greater than 1 - $n^{-c\alpha}$ and in the rest of the paper they will be used in an interchangeable manner). Note that an algorithm whose *expected* resource bound is $O(f(n))$ does not have any better confidence interval beyond using Markov's inequality i.e. the probability that it exceeds the resource bound by a factor k is less than $1/k$. This implies that the failure probability does not diminish as rapidly as the high likelihood bounds. High-likelihood bounds are especially useful for parallel algorithms, where we need to bound the time complexity of all the processes. In contrast, the expected bounds as used by Clarkson[8] are difficult to use to bound the overall maximum time for all processes.

We will be using the term high-likelihood in a variety of situations throughout this paper that may look different from the canonical form given in the previous paragraph. We illustrate this with two lemmas which will be of use later.

Lemma 2.1 *The union of k events (k being any fixed integer), each of which succeeds with high probability, also succeeds with high probability.*

If the failure probability of event i is $< 1/n^{\alpha_i}$ the failure probability of the union of the events is less than $\sum_{i=1}^{k} n^{-\alpha_i} < k/n^{\alpha}$ where $\alpha = min(\alpha_1, .., \alpha_k)$. This is less than $n^{-(\alpha-\delta)}$ for any $\delta > 0$. Note that the above holds with appropriate change of constants even if k is a polynomial in n of some fixed degree.

In the remainder of this paper we shall often refer to the logical structure of the recursive parallel algorithm as a *process-tree* associated with the algorithm. The root of this tree is an instance of the original problem each internal node corresponds to a procedure and the children of a node corresponds to the parallel recursive calls made by the procedure.

Theorem 2.1 *Given a process-tree which has the property that a procedure at depth i from the root takes time T_i such that*

$$P[T_i \quad \geq \quad k\alpha \log n(\epsilon_0)^i] \leq \quad 2^{-(\epsilon_0)^i \alpha \log n}$$

then, all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time. Here i denotes the distance of the procedure from the root level, k, and α are constants greater than 0 and $0 < \epsilon_0 < 1$.

A rigorous proof of this assertion can be found in Reif and Sen [27] (a generalization of Reif and Valiant [29]). Intuitively, the time taken at a node which is at a distance i from the node is $O(\log n/2^i)$ with high probability.
For the rest of the paper, we assume that the success probability required by the algorithm is given, so that given n, we can fix α. From this, one can compute the required probability of success at every individual step of the algorithm even though we do not provide explicit formulae. Also, for convenience of notations, functions of n that may not be necessarily integral valued like $\log\log n$ or n^ϵ will actually denote the ceiling of such values i.e. $\lceil \log\log n \rceil$ and $\lceil n^\epsilon \rceil$. This does not affect the asymptotic bounds of the algorithm.

2.2 Useful results

In the remainder of the section we shall assume that the half-spaces are described as inequalities of the form $a_i x + b_i y + c_i z + d_i \geq 0$. We shall also use the terms 'half-space' and its bounding 'plane' interchangeably where it is clear from the context. The output is a list of vertices of the polyhedron $\mathcal{C}$ which is the intersection of the half-spaces. The vertices are defined by the 3-tuples of the three intersecting planes defining the vertex. We assume that the planes are in general position, that is every vertex is the intersection of exactly three planes. This assumption is for the convenience of analysis and not a real bottleneck for the algorithm. There are standard perturbation techniques which simulates the non-degeneracy condition. For example, changing the coefficients by a sufficiently small real value which is chosen randomly satisfies the property of non-degeneracy with probability approaching 1. The textbook [19] discusses symbolic perturbation techniques which are deterministic but more expensive. The edges of this polytope are those pairs of vertices which have two common planes in the tuples. A face is defined by all tuples which have one common plane. A tuple can be written in 6 ways (permutation of the 3 planes) and thus sorting them (all the 6 possible representation of the vertices) would enable us to obtain the faces and edges as the necessary adjacency structure of the polytope (which is a planar graph).

The following observation is useful for constructing the intersection of a random subset of half-spaces that is used to split up the problem evenly.

Lemma 2.2 *The intersection of a given set of n half-spaces can be computed in $O(\log n)$ time using n^4 processors in a CREW PRAM model.*

Proof: Assuming non-degeneracy (i.e. no 4 planes intersect at a common point), there are $O(n^3)$ candidate vertices for vertices of the convex hull (of the intersection). For each vertex, test whether it is a vertex of the convex polyhedron by checking if it satisfies all the equations defining the half-spaces. This can be done trivially in $O(\log n)$ time using $O(n)$ processors for each candidate point. Only the vertices would survive. Determine the faces of the convex polyhedron by identifying planes that contain 3 vertices of the intersection. The necessary adjacency structure can be constructed by an application of sorting. □

Lemma 2.3 *Given a set of n half-spaces, it is possible to compute their intersection in $O(\log^3 n)$ time using n processors in a CREW PRAM model.*

Proof: Follows immediately from Aggarwal et al. [1].

The previous result is used to stop the recursion at a level when the problem size is small (typically $O(\log^k n)$ for some integer k) and solve the problem directly. Note that any polylog-time algorithm using a linear number of processors would again suffice for our purpose. At this stage the problem size is so small that using a sub-optimal algorithm will not affect the asymptotic complexity of the algorithm.

2.3 The dual transform

The convex-hull problem has a very interesting dual problem, namely the intersection of half-spaces. This dual transformation $\mathcal{D}$ maps a point in E^d to a non-vertical hyperplane in E^d and vice-versa. Let $p = (\pi_1, \pi_2, ..., \pi_d)$ be a point in E^d. Then $\mathcal{D}(\mathrm{p})$ is the hyperplane $1 = \pi_1 x_1 + \pi_2 x_2 + ...\pi_d x_d$ and vice-versa such that a hyperplane h not containing the origin is mapped to a point p for which $\mathcal{D}(p) = h$.

The transform $\mathcal{D}$ is extended to sets of points (hyperplanes) in a natural way. Let $\mathcal{P}$ be a convex polytope with non-empty interior $int\mathcal{P}$ and assume that the origin O is contained in $\mathcal{P}$. Then $\mathcal{D}(\mathcal{P})$ is an infinite set of hyperplanes that avoid some convex region around O. The dual of $\mathcal{P}$ is defined as

$$\bar{\mathcal{P}} = closure(\bigcap_{h \in \mathcal{D}(\mathcal{P})} h^{pos})$$

where h^{pos} denotes the half-space containing the origin. The following observation can be verified [19]:

Lemma 2.4 *Point p belongs to the $int\mathcal{P}$, $boundary\mathcal{P}$ or $complement\mathcal{P}$ if and only if the hyperplane $\mathcal{D}(p)$ avoids $\bar{\mathcal{P}}$, avoids $int\bar{\mathcal{P}}$ but not $\bar{\mathcal{P}}$, or intersects $int\bar{\mathcal{P}}$ respectively.*

In other words, given a set of points S, the vertices of the convex hull are the dual transform of the facets of the intersection of the half-spaces $\mathcal{D}(S)$ (will be denoted by S^* in future). This property has been exploited very often so that the same algorithm can be used for both convex-hulls and intersection of half-spaces (if we know an interior point). In this paper, we actually derive an algorithm for constructing the intersection of half-spaces. Moreover, the dual transform has nice applications for searching (as used in section 5).

3 A naive random sampling algorithm and its shortcomings

Before we embark on a formal proof of the main theorem, let us give an informal description of the algorithm using the straight-forward random sampling strategy (as used by Clarkson [10]). We

intentionally leave out Polling from this preliminary discussion to illustrate the pitfalls of using naive sampling strategies for parallel algorithms. We shall assume for the time being that we know a point p^* in the intersection of the S^* and later show how to determine such a point efficiently. Using a random subset of S^*, we split the original problem *evenly* into smaller sized problems and then apply the algorithm recursively to each of the problems. By using a random subset of size n^ϵ, $(0 < \epsilon < 1)$ we split up the problem into sub-problems of expected size $n^{1-\epsilon}$. This results in a recurrence of the form $\bar{\mathrm{T}}(\mathrm{n}) = \bar{\mathrm{T}}(n^{1-\epsilon}) + f(n)$, where $f(n)$ is the time for dividing the problem. If $f(n) \leq \tilde{O}(\log n)$ (which requires the use of Polling that we describe in section 4), we have an algorithm whose expected running time is bounded by $O(\log n)$. We further need to show that the number of processors required at each step of the algorithm is $O(n)$.

Algorithm *Main*

Input: A set S^* of n half-spaces $H_1,\ H_2, \ldots H_n$.
Output: The output convex polyhedron $\mathcal{C}$ which is intersection of the n half-spaces.
(1) Choose a random subset $R \subset S$ of half-spaces such that $|R| = n^\epsilon$ (for some ϵ, $0 < \epsilon < 1$ that we shall determine during the course of analysis).
(2) Find the intersection of the half-spaces in R. Take a fixed plane and cut up each face of the polyhedron with (parallel) translates of this plane passing through the vertices. Thus each face is a trapezoid. Further, partition each trapezoid with a diagonal so that each face is triangular. For a face F_i consisting of vertices x_i, y_i, z_i consider the cone C_i formed by p^* as the apex and F_i as the base. Let C_R denote the number of cones. Note that $C_R \leq 2|R|$.
(3) For the remaining $S - R$ half-spaces find the intersection of the planes (bounding these half-spaces) with the cones. Note that a plane may intersect more than one cone. The intersection of the S half spaces is the union of the intersection of the half-spaces intersecting a cone (over all cones). That is, $\mathcal{C}$ is $\cup_{i=1}^{C_R} I_i$ where I_i is $\cap_j \{H^j\}$ restricted to C_i.
(4) If the number of planes intersecting a cone is more than a pre-determined threshold apply steps 1-3 recursively to this cone for the set of half-spaces (bounded by the planes) else solve the problem directly (using Lemma 2.4).

The algorithm outlined above that uses a straight-forward random-sampling in step (1) is only a skeleton of the actual algorithm and is not very efficient in its present form. One of the main problems is that in step (3), the total size of the sub-problems could exceed the size of the parent (calling) problem by a large factor at each recursive call. Note that bounding this increase at each recursive call by a constant factor does not suffice. This would imply that after O(loglogn) levels, we can only bound the the total size of the subproblems at this stage by $\mathrm{O}(\mathrm{nlog}^{O(1)}\, n)$. This is where this algorithm differs from some other recursive parallel algorithms like randomized parallel sorting algorithms of [29, 30] where the total size of the subproblems is always bounded by the

input size. We need more sophisticated methods for choosing random subset in step (1) to prevent this. We will show in section 4 how to solve this problem using Polling. Moreover, coming up with a fast procedure for detecting the intersection of the half-planes with the cones is by itself a non-trivial task. For the rest of the paper, we concentrate on individual steps and derive the necessary refinements to prove the main theorem.

4 Probabilistic lemmas

4.1 The need for Polling: an improved random sampling technique

A crucial part of the analysis rests on showing that a random subset R can be chosen efficiently in the first step of the algorithm that divides the the problem into almost equal sized sub-problems. In addition, we have to show that the total size of the sub-problems is same as the complexity of the original problem at every stage of the recursive calls. The following result follows from Clarkson (Clarkson [10], Corollary 4.3) for any random $R \subset S$ with $|R| = r$.

Lemma 4.1 *Let X_i denote the set of planes intersecting cone C_i (using the same terminology as in step 2 of the algorithm). Then the following conditions hold with with probability at least 1/2*

(i) $\sum_{i=1}^{C_R} |X_i| \leq k_{total}(n/r) \cdot E(C_R)$
and
(ii) $max_i|X_i| \leq k_{max}(n/r) \cdot \log r$
where k_{total} and k_{max} are constants and C_R is defined previously.

Any subset of the input half-spaces that satisfies the above conditions for some fixed constants is defined to be 'good' and 'bad' otherwise. A direct consequence of the lemma is that we can divide the problem into almost equal size sub-problems, such that the increase in the original problem size can be bounded by at most a constant multiplicative factor of k_{max}. Since our objective is to apply this recursively, we need a more sophisticated sampling algorithm to obtain a sample that is 'good' with high likelihood.

4.2 An informal description of Polling

The idea for choosing a 'good' sample is following. Since the above events would fail only with constant probability, the probability that the conditions would fail in $O(\log n)$ independent trials is less than $1/n^{\alpha}$ for some $\alpha > 0$. Therefore, if we choose independently $p(n)(= O(\log n))$ sets of samples, one of them is good with very high likelihood. However, to determine if a sample is 'good', we would have to carry out step (3) of the algorithm described in the previous section $O(\log n)$ times, each of which would require $O(\log n)$ time (such a method is described in section 5). Instead,

we try to estimate the the number of planes intersecting a cone C_i using only a fraction of the input planes. For example, we can choose $c_0 \cdot n/\log^d n$ half-spaces for some fixed integer $d > 2$ and a constant c_0 of the input planes randomly for the jth sample, R_j. The actual value of c_0 will be determined from the required success probability of the algorithm. Let X_i^j be the number of planes intersecting cone C_i corresponding to sample R_j, $1 \leq j \leq b \log n$ where b is fixed integer greater than 0. Let A_i^j be the number of planes intersecting C_i out of the $n/\log^d n$ randomly chosen input planes for the same sample. Clearly, A_i^j is a binomial random variable with parameters $c_0 \cdot n/\log^d n$ (total number of trials) and X_i^j/n (success probability in each trial). Assuming that X_i^j is greater than $\bar{c} \cdot \log^{d+1} n$, for some constant $\bar{c}$, we will apply Chernoff bounds (see appendix) to tightly bound the estimates within a constant multiplicative factor. Since we do it only for $1/\log^d n$ of the input planes, the total number of operations for the $O(\log n)$ random subsets is bounded by $O(n \log n)$ (as we show in the next section). Note that $X_i^j < \bar{c} \log^{d+1} n$, is an easy case since $n^{\epsilon} \cdot \bar{c} \log^{d+1} n$ for $\epsilon < 1$ is $o(n)$.

4.3 Probabilistic analysis of Polling

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

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

and

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

(for $c_0 > 1$). From the last two inequalities, X_i^j is bounded by $L^j = A_i^j \log^d n / c_0 c_2 \alpha$ from below, and $U^j =$ by $A_i^j \log^d n / c_1 \alpha$ from above. With appropriate changes in the constants, this condition holds with high likelihood (as defined in section 2.1) for all X_i^j simultaneously. We do the procedure (described in the next section) simultaneously for all the samples R_j and choose the sample R_{j_o} using the following simple test:

Algorithm *Polling*

(Let $N^j = \sum A_i^j$ and the let actual number of intersections be T^j. Let E^j be our estimate for the sample j and upper and lower bounds obtained from N^j are denoted by U^j and L^j respectively).

If $k_{total} n > U^j$ then accept sample R_j (since $k_{total} n \geq U^j \geq T^j$),

else if $k_{total} n \leq L^j$ then the sample is 'bad' (since $k_{total} n \leq L^j \leq T^j$),

else if $L^j \leq k_{total} n \leq U^j$, then accept the sample R_{j_o} for which E^{j_o} is minimum. Since both $k_{total} n$ and T^{j_o} lie in this interval this guarantees that $T^{j_o} \leq c_3 \cdot k_{total} n$ where $c_3 = U^j / L^j$ which is a constant.

Recall, that from our earlier discussion at least one of the samples would satisfy conditions 1 or 3 with very high likelihood. We summarize as following :

Lemma 4.2 (Polling lemma) *If we can choose a set of random splitters that expects to be 'good', then by using the* **polling** *algorithm, as described earlier we obtain a sample that is 'good' with high probability.*

The above procedure can actually be used in a more general situation where we need 'good' samples with very high likelihood from samples that only expect to be 'good'. Moreover, according to our previous discussion, the extra amount of overhead does not affect the asymptotic work done by the algorithm, because it uses only a fraction of the input to test the samples.

5 Finding intersections quickly

5.1 A locus-based approach for finding intersections

We now focus on a procedure to find the intersection of planes with each of the cones, C_i. Notice that a plane may intersect more than one cone which rules out detecting the intersections sequentially. That is, if a plane intersects n^δ cones ($\delta > 0$), we cannot afford to detect them one after the other since we are looking for an $O(\log n)$ time procedure. Note that in the sequential case, Clarkson and Shor's [11] randomized incremental constructions give optimal expected time bounds for computing the plane-cone intersections that cannot be applied in our case.

We shall use a *locus-based* approach to solve this problem. This approach involves considering each query as a higher-dimensional point and partitioning the underlying space into regions providing the same answer. Thus any query problem can be reduced to a point location problem given sufficient preprocessing time and space. In our case, we have to pre-process the convex polytope of the sampled half-spaces in such a way that given any plane, we should be able to report the list of cones that it intersects in $O(\log n)$ time using at most k processors where k is the number of intersections. We shall show that the pre-processing for a convex polytope of size n can be done in $O(\log n)$ parallel time using $\mathrm{O}(n^c)$ processors, where c is a fixed constant. Thus we can choose any sample of size less than $n^{1/c}$ since we have n processors. For our problem, the a value of c is will be worked out in the proof of Lemma 5.1.

Given a convex polytope in three-dimensions of size $O(n)$ along with an internal point which is the apex of the cones, there are only a polynomial (in n) number of combinatorially distinct possibilities of the way any given plane can intersect the cones. This can be seen from the following simple argument. Given any plane that intersects the polyhedron, we can perturb the plane without changing the cones it intersects so long as it remains within a fixed set of bounding vertices. Figure 1 illustrates the situation for a two-dimensional case. If we consider an equivalence relation where two lines are equivalent iff they intersect the same sets of cones, then the equivalence classes correspond

to the cells in the arrangement $\mathcal{A}(H)$ where H = $\{\mathcal{D}(p) : p$ is a vertex of the convex n-gon or internal point and $\mathcal{D}$ is a dual transform $\}$ (see [19] for more details). Given any query line l, the cones that it intersects is defined by the partition of $\mathcal{A}(H)$ that $\mathcal{D}(l)$ belongs to. This observation can be extended to hold for any dimension; in our case three. If we consider the partitions of the three-space induced by the intersections of the constraining half-spaces, these are equivalent classes with respect to the cones they intersect. Notice that even if this partitioning may not be minimal but it suffices for our purpose. All that remains to be done is pre-compute for each of these regions the cones that the corresponding planes would intersect so that for any query plane in the same equivalence class we can list off the intersecting planes by a table look-up.

5.2 A point-location algorithm

For the point-location problem, we use a pre-processing scheme due to Dobkin and Lipton [17] because of the ease in parallelization. The following is a fairly straight-forward extension of their method

Lemma 5.1 *Given a set of m planes in E^3, it can be pre-processed in O(logm) time using $O(m^7)$ processors, such that given an arbitrary query point, the unique cell containing that point can be reported in O(logm) time. The space required is $O(m^7)$.*

Proof: Find the pairwise intersections of the given set of planes (there are $O(m^2)$ of them). Project the resulting lines on a plane not normal to any of the lines. Find the pairwise intersections of the straight lines and consider their projection on the x-axis. There are $O(m^4)$ intervals induced by these. For each of these intervals, order the straight-lines in increasing ordinates by sorting. This can be done in $O(\log m)$ time using $O(m^2)$ processors for each of the $O(m^4)$ intervals. There are now $O(m^6)$ trapezoidal regions. For each of these, order the planes in increasing z-coordinates for binary-search by sorting which are totally ordered in these subdivisions. This can be done in $O(\log m)$ time using $O(m^7)$ processors. The cells induced by this pre-processing are homeomorphic to a 3-cube, so that given any query point it can be located in such a subdivision with 3 binary searches. □

For each of the cells in 3-space, we can precompute the cones that the corresponding plane intersects using $O(n^8)$ processors (by choosing a representative point in each cell and testing it against all the cones). Note that these subdivisions are finer than the minimal equivalence classes i.e. more than one subdivisions could have the same set of intersecting cones. We also store the number of intersecting cones for each of the subdivisions so that while listing the number of cones each query plane intersects we can do the processor allocation easily in $O(\log n)$ time using a prefix computation. By choosing less than $n^{1/8}$ samples, we can complete the entire preprocessing in the required time and processor bounds.

We summarize our conclusion in this section as follows

Lemma 5.2 *Step three of Algorithm Main uses $O(n)$ space and terminates in $O(\log n)$ time using n processors in a CREW PRAM model.*

6 Controlling the size of subproblems and processor allocation

From lemma 4, we know that the size of the problem can increase by a constant factor at each level and we wish to avoid this happen over O(loglogn) levels, which would increase the number of processors required by a polylog factor. For this we need to quickly identify the redundant planes that do not contribute to the output complexity and eliminate them from further recursive calls. This enables us to get a global bound on the total size of the subproblems (at any stage) which we shall show to be linear in the input plus the output size. More specifically, we allocate the processors recursively to the cones such that the number of processors is proportional to the number of output vertices in that cone, thereby bounding the number of processors to be $O(n)$. The details of the procedure are described below.

After we have found the planes intersecting a particular cone, we categorize them as following:

(a) planes that are completely occluded by another plane in the cone and hence these cannot be a part of the output in the cone,
(b) planes that are occluded because of more than one other plane in the cone i.e. there is no one plane that completely occludes them,
(c) planes that contribute to an edge without an end-point i.e. the end-points lie in some other cones,
(d) planes that do contribute to a vertex in the cone.

To eliminate planes of type (a), we use a variant of the *3-D* maxima algorithm. The *3-D* maxima problem is defined as:

> Given a set S of n points in a three-dimensional space, determine all points p in S such that no other point of S has x, y and z coordinates that simultaneously exceed the corresponding coordinates of p. In case there is such a point q, q is said to *dominate* p.

Since cones have a triangular base there are 3 edges that join it to the apex p^*. We sort the intersections of the planes with an edge in increasing distances from the apex. We repeat this for all the three edges. Call these three edges X, Y, Z and denote the intersection of a plane h_i as X_i , Y_i , Z_i and the ranks in the sorted list as r(X_i), r(Y_i) and r(Z_i).
Observation 1: If a plane A is occluded completely by another plane B iff it is dominated on its ranks of intersection on all the three edges by plane B.
This gives an effective strategy for eliminating planes of type (a) by identifying the complement of the set of the maximal elements, where we use the ranks of the intersection on the three edges

as the order relation. Using the algorithm of [2], we can do this in $O(\log n)$ time using a linear number of processors. For a two-dimensional illustration see Figure 2.

To identify planes of type (b) (c) and (d) we construct the intersection of the convex polytope $\mathcal{C}$ with each of the three faces of the cone. These are intersections of the faces with $\mathcal{C}$ that are 2-D convex polygonal chain. These will be referred to as *contours* for the following discussion. The *contours* can be computed in $O(\log n)$ time with n processors using any of the optimal 2-D convex hull algorithms. Note that these convex *contours* on the three faces are a part of the output and any plane that appears on this contour is a part of the final output. Consequently, a plane of type (b) cannot be a part of this contour. Unfortunately, there can be planes that are part of the output but are not part of any *contour*. Consider a plane that chops off a portion of the polytope within the cone. For the time being let us focus on only those planes that show up in the contours and consider the 3-D convex polyhedron formed only by these planes within a cone. We shall refer to such a 3-D polytope as a *skeletal-hull*. We shall use the term 'flattening' to imply that the vertices of the contour are projected along edges (intersection of two planes) they lie on, such that all of them become coplanar. Notice that there may be several such planes. We just choose one arbitrarily and these projected vertices defines a 'base' face.We now make following observation

Observation 2: Any plane that is not a part of the contour on any face can intersect at most one *skeletal hull*.

This follows from convexity. Notice that such planes are not necessarily a part of the output but we are not aiming for an output sensitive algorithm. The previous observation guarantees that if a plane is not a part of $\mathcal{C}$ it will not survive in more that one cone when the algorithm is called recursively in the cones. The planes that do not intersect the *skeletal-hull* cannot be a part of $\mathcal{C}$ within the cone.

A plane can be a part of the contour and not contribute to any vertex of the convex polytope $\mathcal{C}$, that is it only contributes a edge of the hull within the cone. In this case the edge intersects the cone in exactly two faces and the these vertices can be labeled by the two intersecting planes (which contributes to the output edge). Thus these planes can be identified quickly using sorting on the labels of the intersecting planes. We shall call these *free-edges*.

The objective of the above procedure is to eliminate some of the planes that do not contribute to the output and ensure that going into any recursive call, the sum of the subproblems is less than the output size. We define the output size of the 3-D convex polytope to be $3|V|$ where V is the number of vertices of the convex polytope. Since the surface of the convex polytope is a connected planar map, we can use Euler's equation to show that $|V| = 2|F| - 2$ where F is the set of faces in the polytope. Since $|F| \leq n$, $|V| \leq 2n - 2$. This calculation is done using the property that each vertex is of degree 3 (which follows from our assumption that the planes are in general position). Let the number of processors be $6n$. We distribute the processors among the subproblems depending on the output size. For a cone C_i, that does not contain any *free-edge*, the

output can be bound by the following:

Claim 6.1 *The output size of a cone is bounded by $3n_i + 6m_i - 6$, where n_i is the number of planes in the contour contributing at least one vertex and m_i is the number of planes of type (b) and (d).*

Proof: Let e_1 denote the number of edges of $\mathcal{C}$ that intersect the contour and contribute a vertex within the cone (the vertices of the contour are these edges). Let e_2 be the number of edges which lie within the cone (including both end-points). Let v be the number of vertices of $\mathcal{C}$ within the cone (these have degree 3), then $e_1 + 2e_2 = 3v$ or

$$e_1 + e_2 = \frac{3v + n_i}{2}$$

as $e_1 = n_i$. Consider the planar map of the polyhedron formed by n_i and m_i planes and a 'base' face by 'flattening' the contour. Refer to the explanations of the terms 'base' face and 'flattening' in the previous paragraphs. The number of edges on the contour equals the number of vertices on the contour. Then by applying Euler's formula $|V| = n_i + 2n_f - 2$ where n_f is the number of faces of of $\mathcal{C}$ that show up in the cone but are not a part of the contour. Since n_f (type d) can be bounded by m_i the claim is proved. □

Notice that if a cone contains d *free-edges*, then the above formula can be applied separately to each of the $d + 1$ partitions (induced by the free edges). The participant planes in each of these partitions are disjoint giving us the following:

Claim 6.2 *The output size of a cone is bounded by $3n_i + 6m_i - 6d$ where d is the number of free-edges.*

The processor allocation strategy is to simply allocate this number of processors to the sub-problem (in the cone). The total number of vertices over all the cones is bound by the maximum output size and by our allocation strategy, we are allocating processors proportional to the maximum output-size in each cone. Note that the actual output size may be less but we shall never have fewer processors than required and this maximum size can be actually achieved. Another way to look at the processor allocation strategy is that two processors are allocated to each edge of the output hull and we allocate those processors to the cones which contains (or potentially contain) vertices associated with that edge. The *free-edges* are not allocated any processors. Hence we have sufficient number of processors.

We shall now describe a procedure to construct the *skeletal-hull* within a cone and preprocess the *skeletal-hull* such that queries of the kind plane-polyhedra intersection detection can be answered quickly. The latter part can be done efficiently using a hierarchical polyhedra decomposition scheme due to Dobkin and Kirkpatrick [15]. The construction of the hierarchical representation can be done in $\tilde{O}(\log n)$ time using an algorithm of described in Reif and Sen [27] (also discovered independently by Dadoun and Kirkpatrick [14] but the analysis given in their paper is not sufficient for our

purposes). Given this representation, the plane-polyhedra intersection detection query can be answered in $O(\log n)$ sequential time ([16]).

We shall now discuss how to construct the *skeletal-hulls* quickly. Although the *skeletal-hulls* are themselves 3-D convex polytopes they have a much simpler structure (see Figure 3). More specifically, they have the following property: all faces are unbounded (i.e. they are part of the *contours*). This implies that, if we construct them recursively using the same algorithm, we do not have to worry about case (b) since all planes that are part of the output will show up in the *contours* and this holds for any level of the recursive call. From the analysis given in the next sub-section the *skeletal-hulls* can be constructed in $\tilde{\mathrm{O}}(\log n)$ time using a linear number of processors. The reader should convince himself that there is no circularity of arguments here. One way to look at the problem is the following : assuming that there are no planes which satisfies both cases (b) and (d) (i.e. all planes that are part of the output show up in the *contours*), the algorithm terminates in $\tilde{\mathrm{O}}(\log n)$ time using a linear number of processors. So after having constructed the *skeletal-hull* for the cone, the redundant planes are quickly eliminated using the procedure outlined in the previous paragraph. Subsequently, the algorithm is called recursively on the cone - this time to build the actual polytope $\mathcal{C}$ (as opposed to the *skeletal-hull*).

6.1 Final analysis

Consider the algorithm as a tree where each node corresponds to a procedure and the children of a node representing processes corresponding to the recursive calls made by the procedure. Then the running time of the algorithm corresponds to a worst-case sequence of nested procedure calls along any path in this tree from the root to a leaf node. Let us first analyze the cost of constructing the *skeletal hull* at any node. This process tree corresponding to this algorithm has the following property. If we choose the sample size to be $O(n^{1/8})$ a process at level i ($1< i <$ O(loglogn)) has size less than $\mathrm{O}(n^{(10/9)^{-i}})$ (from Lemma 4.1) and the process terminates in time $\mathrm{O}(\log\, n^{(9/10)^i}) = (9/10)^i\mathrm{O}(\mathrm{logn})$ with probability greater than 1 - $1/n^{(9/10)^i}$. From Theorem 2.1, any nested sequence of recursive calls exceeds time $c\gamma \log n$ with probability less than $1/n^{\gamma}$ for any $\gamma > 1$. It follows that all the leaf processes and hence the algorithm is completed within the same time with high likelihood. We may may thus conclude that the *skeletal hull* at any node with problem size n_i can be constructed in $\tilde{\mathrm{O}}(\log n_i)$ time.

For the overall algorithm, we add the time for detecting redundant half-planes. From the previous section this involves constructing a data structure for point location for detecting the plane-polyhedron intersections. Since this also takes $\tilde{\mathrm{O}}(\log n_i)$ time, from Lemma 2.1, the total time is also $\tilde{\mathrm{O}}(\log n_i)$ (with appropriate constants in the $\tilde{\mathrm{O}}$ notation). Another appeal to Theorem 2.1 results in the $\tilde{\mathrm{O}}(\log n)$ running time for the overall algorithm. The space used is $O(n)$ at step 3 of each recursive level giving a total bound of $O(n \log\log n)$ for all the $O(\log\log n)$ recursive levels of the algorithm. This proves the main result of the section. □

Corollary 6.1 *The following problems can be solved in $\tilde{O}(\log n)$ time using n processors in the CREW PRAM model*

(i) Convex hull of a set of points in 3-D
(ii) Voronoi diagram of point-sites in plane
(iii) All-points nearest neighbor
(iv) Euclidean minimal spanning tree

Proof: (i) follows immediately because of well-known reduction of convex hulls to intersection of half-spaces. To determine an internal point p^* in the intersection, we can determine an internal point of the convex hull and use it as the origin for the duality transform. The origin is known to be contained in the intersection of the half-spaces.
For (ii), given a set of n points in the plane we apply an inversive transformation given by Brown [4] to the input points to transform the problem into finding the convex hull of n points in 3-space.
(iii) can be obtained in $O(\log n)$ time from the Voronoi-diagram.
(iv) can be obtained by running a minimal-spanning tree algorithm on the edges of Delaunay triangulation which is the dual graph of the Voronoi diagram. This algorithm uses the stronger Priority CRCW model as in [3] □

7 Bounding random bits

7.1 Chebychev's inequality

The commonly used form of Chebychev's inequality has the form:

$$Prob[(|X - \mu| \geq t)] \leq \frac{\sigma^2}{t^2}$$

.

This simple fact was exploited by Chor and Goldreich [5] for their 2-point sampling theorem where they consider the following scenario. To determine if a property P holds for x (for example if x is prime), we often use a function $f(x, r)$ which satisfies the condition that if P holds for x then $f(x, r)$ is 1 with probability at least $1/2$ whereas if P does not hold for x then $f(x, r)$ is 0. Here r is a witness which is a random number in a certain range. This implies that if $f(x, r)$ is 1, then repeating the experiment for t *independent* random witnesses decreases the failure probability (of incorrectly categorizing x) to 2^{-t}. Instead of choosing t independent witnesses, if we choose t witnesses which are pairwise independent then we can analyze the probability of error as follows. Assume that in t trials, $f(x, r_i)$ is 0 for all the trials and $P(x)$ is true. Let $Y = \sum_i f(x, r_i)$. Then $E[Y] \geq t/2$ and the variance $\sigma_Y \leq \sqrt{t}/2$. Then $Prob[Y = 0] \leq Prob[|Y - E[Y]| \geq t/2$ which is less than $1/t$ from Chebychev's inequality.

The t *pseudorandom* numbers can be generated from two purely random numbers by using the following scheme $r_i = a + bi$ where a and b are random numbers. Thus instead of the confidence bound of $1/4$ (for two numbers), we can do much better i.e. $1/t$. Karloff and Raghavan [23] were able extend these techniques to show that Reischuk's sorting algorithm can be implemented in the same asymptotic bounds by using only $O(\log n)$ purely random bits (instead of the naive scheme requiring $O(\sqrt{n})$ random bits). Here we generalize their scheme to minimize the number of random bits used by the algorithms described in the previous chapter. For this we need to prove some preliminary results.

Definition: A family of random variables is called k-way independent if any subset of k variables are mutually independent.
Clearly a k-way independent family is l-way independent for any $l \leq k$.

Let p be a suitable prime number and choose k numbers a_i, $0 \leq i \leq k-1$ randomly from Z_p. Consider the numbers of the form $r_i = (\sum_{j=0}^{j=k-1} a_j \cdot i^j) \bmod p$ The following is a well known result [5] for this class of *pseudo-random* number generators.
Fact 1: The numbers r_i's are uniformly distributed in Z_p and are also k-way independent.
Fact 2: If X_i, $1 \leq i \leq n$ are n mutually independent random variables and ϕ_s are real-valued functions, then

$$E[\prod_{s=1}^{n} \phi_s(X_s)] = \prod_{s=1}^{n} E[\phi_s(X_s)]$$

Lemma 7.1 (Generalized Chebychev inequality)

$$Prob\{|X| \geq t\} \leq \frac{E(\phi(X))}{\phi(t)}$$

where ϕ is a positive monotonically increasing function.

Proof: Let $Prob[X = x_j] = f(x_j)$. Then

$$Prob\{|X| \geq t\} = \sum_{|x_j| \geq t} f(x_j)$$

$$\leq \sum \frac{\phi(|x_j|) f(x_j)}{\phi(t)}$$

$$\leq \frac{E(\phi(|X|))}{\phi(t)}$$

Lemma 7.2 *Let X be the sum of n $2k$-way independent and identical Bernoulli random variables X_i, $1 \leq i \leq n$ each of which has a success probability p. Then for a fixed k (chosen independently of n), $Prob\{|X - \mu| \geq \mu\} \leq O(\frac{1}{\mu^k})$ where $\mu = np$ and $p \leq O(n^{-\beta})$ for some $0 < \beta < 1$.*

Proof: Consider the generalized Chebychev's inequality

$$Prob\{|X| \geq t\} \leq \frac{E(\phi(X))}{\phi(t)}$$

. Using $\phi(t) = t^{2k}$ and substituting $X - E[X]$ for X and setting t to be equal to μ, we get

$$Prob\{|X - E[X]| \geq E[X]\} \leq \frac{E[(X - E[X])^{2k}]}{E^{2k}[X]}$$

Let us focus on the numerator - we shall show that it is O(μ^k) and the lemma follows. Since $E[X]$ = $\sum_{i=1}^{n} E[X_i]$, we can write $(X - E[X])^{2k}$ as $(\sum_{i=1}^{n} X_i - E[X_i])^{2k}$
In the multinomial expansion, all the terms containing $X_i - E[X_i]$ (for any i) as a factor vanish because of the $2k$-way independence property.
There are $\begin{pmatrix} n \\ c \end{pmatrix} \cdot \begin{pmatrix} 2k-1 \\ c-1 \end{pmatrix}$ terms which have c distinct non-unit product terms of the form $(X_j - E[X_j])^i$ such that $i > 0$ and $\sum i = 2k$. Also note that
$E[(X_j - E[X_j])^i] = (1-p)(-p)^i + p(1-p)^i$
We can factor out p^i so that we can write the coefficient of n^c as p^{2k}·f(p,c,k), where f is a function independent of n and can be absorbed in the big-O notation. From our observation about the first-order terms (which vanish), the maximum value of c is k. The numerator can be bound by the asymptotically dominating term O($n^k \cdot p^k$) = O(μ^k). Since the denominator is μ^{2k}, the lemma follows. □

Corollary 7.1

$$Prob\{|X - \mu| \geq a \cdot \mu\} \leq O(\frac{1}{a^k \cdot \mu^k})$$

where a is a constant between 0 and 1.

Proof: Substitute $t = a\mu$ in the previous lemma. □

Follows from the previous lemma by setting t to $a\mu$.

7.2 Rederiving probabilistic bounds with fewer random bits

In order to limit the number of random bits, we shall rederive some of the random-sampling bounds and the **Polling lemma** using the $2k$-way independent random variables. In particular we shall prove a slightly weaker bound than Lemma 4.1.

Lemma 7.3 *The probability that the maximum number of half-planes intersecting any cone exceeds $n^{1-\epsilon+\delta}$ is less than $n^{-k\delta+5}$. Here $0 < \delta < \epsilon$.*

Proof: By randomly choosing n^ϵ half-spaces, the expected number of half-spaces chosen in the sample in a sector that has more than $n^{1-\epsilon+\delta}$ half-planes is greater than n^δ. Thus the probability that none of the half-planes were chosen in the sample is less than $n^{-k\delta}$ from Lemma 5.2. Summing over the $O(n^5)$ possible cones (see Clarkson [10] for justification of this bound) gives us the required result.

Although this bound is somewhat weaker than Lemma 4.1, it still suffices to show that the process-tree (corresponding to the algorithm) has $O(\log\log n)$ depth (the constant is somewhat larger). Condition (i) of Lemma 4.1 is not affected since its proof in [10] uses only expectations. [1] For probabilistic analysis of **Polling** we make use of the fact that if the expected number of half-planes in a sector is larger than n^β then Lemma 5.2 directly yields high probability bounds for some $0 < \beta < 1$ and choosing a sufficiently large value for k. If the mean is less than n^β, then for small β, $n^{\epsilon+\beta} = o(n)$.

An identical argument can be carried out for the problems tackled in Reif and Sen [27], namely the trapezoidal decomposition. Notice that the Chernoff bounds that we were using earlier yielded much stronger bounds (of the order of 2^{-n^ϵ}) which was not required to prove our Polling lemma. The ramifications of using these bounds is that the constants associated with the running time for the same confidence bounds are much larger.

From here on we can directly use the scheme of Karloff and Raghavan. The random bits can be shared by the $O(n)$ paths. We need an extra $O(\log n)$ multiplicative factor of truly random bits for implementing Polling for which we need $O(\log n)$ independently chosen $2k$ random seeds of $O(\log n)$ bits each. Moreover the point-location algorithm can be shown to require only $O(\log n)$ bits (Sen [31]). We summarize as following:

Theorem 7.1 *The algorithm for constructing 3-D convex hulls runs in $\tilde{O}(\log n)$ time using n processors and $\tilde{O}(\log^2 n)$ purely random bits.*

8 Concluding Remarks

The randomized algorithms presented here the reinforces the optimism expressed in an earlier paper (Reif and Sen [27]) where we introduced randomization as an effective tool for developing parallel algorithms in computational geometry. Clarkson had demonstrated the usefulness of randomization for deriving improved expected time bounds for a large number of sequential algorithms. Although we draw from Clarkson's work, our results should be of independent interest because of many unique additional difficulties presented by the parallel environment and the techniques one needs to tackle them.

[1] Clarkson [7] pointed it out to the authors.

This paper describes the first $O(\log n)$ parallel time algorithm with optimal speed-up for 3-D convex hulls and related problems, however a number of questions are left unanswered. The most obvious problem is that of designing a deterministic algorithm with same bounds. It is possible that an optimal algorithm for 2-D Voronoi-diagrams may be easier to obtain than a similar algorithm for 3-D convex-hulls. Moreover, we use the CREW PRAM model for our algorithm that raises the question if the algorithm can be made to run without the feature of concurrent reads. A more theoretical issue is that of designing sub-logarithmic time parallel algorithms for all these problems with optimal speed-ups. Also can the probabilistic bounds be improved from $1 - 1/n^c$ for any c to say, $1 - 2^{-n}$?

An extremely important area of investigation in the field of parallel algorithms for computational geometry is development of efficient algorithms for fixed interconnection networks like hypercubes and butterfly networks. In spite of some elegant work done in the PRAM model, the currently best known results for almost all these fundamental problems except 2-D convex hulls remain sub-optimal. It appears very unlikely that the optimal algorithms would be deterministic since there are no known optimal deterministic sorting algorithms for these networks. This should encourage more research in the area of developing more sophisticated probabilistic methods for parallel computational geometry. Recently, Reif and Sen [28] were able to make some progress in this direction by presenting efficient algorithms for triangulation.

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.

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

[3] B. Awerbuch and Y. Shiloach. New connectivity and msf algorithms for ultracomputer and pram. *Proc. of the Int'l Conf. on Parallel Processing*, pages 175–179, 1983.

[4] K.Q. Brown. Voronoi diagram from convex hulls. *Informat. Process Lett.*, 9:223 – 228.

[5] B. Chor and O. Goldreich. On the power of two-point sampling. *Journal of Complexity*, 5:96–106, 1989.

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

[7] K. Clarkson. personal communication.

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

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

[10] 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.

[11] K.L. Clarkson and P. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized and incremental. *Proc. of the 4th ACM Symp. on Computational Geometry*, 1988.

[12] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770 – 785, 1988.

[13] R. Cole and M.T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. *Proc. of the 4th ACM Symp. on Computational Geometry*, pages 201 – 210, 1988.

[14] N. Dadoun and D.G. Kirkpatrick. Parallel processing for efficient subdivision search. *Proc. of the 3rd Annual ACM Symp on Comput. Geom*, pages 205 – 214, 1987.

[15] D. Dobkin and D. Kirkpatrick. A linear time algorithm for determining the separation of covex polyhedra. *Journal of Algorithms*, 6(3):381 – 392, 1985.

[16] D. Dobkin and D.G. Kirkpatrick. Determining the separation of preprocessed polyhedra - a unified approach. pages 400 –413, 1990.

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

[18] M.E. Dyer and A. Frieze. A randomized algorithm for fiixed-dimensional linear programming. *unpublished manuscript*, 1987.

[19] H. Edelsbrunner. *Algorithms in combinatorial geometry.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1987.

[20] S. Fortune. A sweepline algorthm for voronoi diagrams. *Algorithmica*, 2(2):153 – 174, 1987.

[21] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *Proc. of the 27th Annual IEEE Symp. on Foundations of Computer Science*, pages 492 – 501, 1986.

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

[23] H. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. *Proc. of the 20th Annual STOC*, 1988.

[24] C. Levcopoulos, J. Katajainen, and A. Lingas. An optimal expected-time parallel algorithm for voronoi diagrams. *Scandenavian conference on theoretical computer science*, 1988.

[25] K. Mulmuley. A fast planar partition algorithm 1. *Proc. of the 29th IEEE FOCS*, pages 580 – 589, 1988.

[26] M. Goodrich R. Cole and C. Dunlaing. Merging free trees in parallel for efficient voronoi diagram construction. *Proc. of the ICALP*, pages 432 – 445, 1990.

[27] J.H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Proc. of the 16th International conference on Parallel Processing*, 1987. A revised version is available as Duke University technical report CS-88-01.

[28] J.H. Reif and S. Sen. Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications. *Proc. of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, pages 327 – 337, 1990.

[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] M. Shamos and D. Hoey. Closest-point problems. *Proc. of the 7th ACM STOC*, pages 224 – 233.

[33] L.G. Valiant. A scheme for fast parallel communication. *SIAM J. on Computing*, 11:350 – 361, 1982.

A Appendix

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

A Bernoulli trial is an experiment with two possible outcomes viz. success and failure. The probability of success is p.

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

$$\text{Prob}(\text{X} \leq \text{x}) = \sum_{k=0}^{x} \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{n-k}$$

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

$$\text{Prob}(\text{X} \geq \text{m}) \leq (\tfrac{np}{m})^m e^{m-np} \quad (1)$$
$$\text{Prob}(\text{X} \leq \text{m}) \leq (\tfrac{np}{m})^m e^{-np+m} \quad (2)$$
$$\text{Prob}(\text{X} \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2) \quad (3)$$
$$\text{Prob}(\text{X} \geq (1+\epsilon)np) \leq exp(-\epsilon^2 np/3) \quad (4)$$

for all $0 < \epsilon < 1$. The last two bounds actually follow from the Chernoff bounds which (for a discrete distribution) can be stated as

$\text{Prob}[\text{A} \geq \text{x}] \leq \; z^{-x} G_A(z)$ where $G_A(z)$ is the probability generating function.

To minimize the bound we substitute $\text{z} = z_o$ that minimizes the right side expression.