95.508 Computational Geometry, Winter Term 1997:

Report on ”Polling: A Randomized Sampling Technique for Computational Geometry”

Silvia Götz

March 27, 1997

Abstract

This report explains the random sample technique of *Polling* using the example of intersection of half-planes in two dimensions. This technique, invented by J. H. Reif and S. Sen, finds a ”good” sample to partition the problem space by polling the input. As the recursive divide and conquer algorithm would still need more than $O(n)$ processors, a problem dependent bounding (called *Filtering*) is applied based on the partitioning of the ”good” sample.

By applying *Polling* (in combination with *Filtering*) a range of problems can be solved. For example, intersections of n half-spaces and convex hull of n points in two and three dimensions can be computed with *Polling* in $\tilde{O}(\log n)$ time on a $O(n)$ processor *CREW PRAM*.

Contents

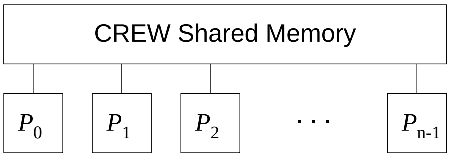


Figure 1: *CREW PRAM* with n processors

1 Introduction

Decreasing prices for hardware and the ability of communicating processors, opened a wide field in computer science. The area of parallel computing deals with the design and analysis of parallel algorithms, i.e. solving a problem by employing several processors in parallel. The running time of a parallel algorithm $T_{parallel}$ is the time between when the first processor starts and the last processor ends its computation. Let n be the number of processors. Then, the work $T_{parallel} \cdot n$ to solve the problem obeys, of course, the lower bounds in the sequential setting. This can easily be seen by the argument of simulation the parallel processors sequentially. If the work in the parallel algorithm equals the work for the sequential one, the algorithm is to be said *work-optimal.*

Parallel algorithms can be designed to run on a specific multicomputer or they are developed with respect to a model of parallel computation. In the latter you simplify the features of a real parallel machine, so that you can examine the potential parallelism in a problem more easily. Furthermore, machine independent algorithms can be applied more generally. On the other hand, when run on a real parallel machine, usually these algorithms do not perform well. Therefore, either adjustments of the algorithm are made, or a different algorithm is developed in respect to the features of the specific machine.

1.1 The Model Used

Perhaps the most common (and easiest) model for parallel computing is the *Parallel Random Access Machine*, *PRAM* for short. It consists of n processors (random access machines) which are assigned unique identifiers $p_0, p_1, \ldots, p_{n-1}$. The processors communicate through a shared memory, e.g. a processor writes a value into a memory location and in the next step another processor can read the information by accessing the same memory cell. For one read or write access to a memory location one time unit is charged. However, the concurrency of accessing the same memory cell can be restricted. The following different types of *PRAM* models show the possibilities.

1. EREW (exclusive read, exclusive write): At any time no more than one processor is allowed to access the same memory) location.

2. CREW (concurrent read, exclusive write): At any time no more than one processor is allowed to write to the same memory location, but more than one processor can read the same memory cell at the same time.

3. CRCW (concurrent read, exclusive write): More than one processor can access the same memory location at the same time for read and write. As a memory location can only have one value, there are different solutions for which value the cell will contain after the concurrent write.

The theoretically possible combination of exclusive read and concurrent write makes no sense, since writing is a more "powerful" and more "difficult" operation. The algorithms presented in this report will use the *CREW PRAM* model. The structure of this model is shown in figure 1.

Randomization can not only be applied to sequential algorithms, but also to parallel ones. In the PRAM model this requires the assumption, that each processor can get a random number of (bit)length $O(\log n)$ in unit time.

1.2 An Example of a Parallel Algorithm on the *CREW PRAM*

A simple example for a parallel algorithm is the *prefix sums* computation on a *CREW PRAM* with n processors. Let $x_0, \ldots, x_{n-1}$ be n real numbers. The problem is to compute the partial sums $\sum_{j=0}^{i} x_j$ for each $i \in \{0, \ldots, n-1\}$.

At the beginning, each processor p_i starts with value x_i. The processors are grouped into disjoint pairs of consecutive processors. The first processor of such a pair accesses the value of its left neighbor and adds the value to its own number. In general at each step k, 2^k consecutive processors $p_{i \cdot 2^k}, \cdots, p_{(i+1)2^k-1}, i = 0, \ldots$ form disjoint group. After this step the prefix sums of the values $x_{i \cdot 2^k}, \ldots, x_{(i+1)2^k-1}$ are computed. Thus, after $O(\log n)$ steps the prefix sums of all x-values are determined. Each step takes a constant number of operations, so that the total running time of the algorithm is $O(\log n)$. Figure 2 illustrates the computation of the prefix sums. Note the concurrent read, e.g. processors p_0 and p_3 access value 8 in step 2 at the same time. With exclusive read the processors would have to "stand in line" to access the value. This would lead to an increase in the running time.

The work of the algorithm (as defined earlier) is $O(n \log n)$ which is not work optimal, since the prefix sums can be computed sequentially in linear time. Small alterations on the above algorithm enables you to compute the prefix sums of n values on a *CREW PRAM* with $\frac{n}{\log n}$ processors in $O(\log n)$ time. (Grouping elements to a size of $\log n$ and compute their prefix sums sequentially [but in parallel for the groups] before applying the above algorithm leads to the result.)

1.3 Recursive Parallel Algorithms using Randomization

In a recursive parallel algorithm the original problem $P(n)$ of size n is partitioned into several subproblems $P_{sub}(n)$. The time between the first processor starts and the last processor ends, is determined by the longest possible path in the recursion tree (figure 3). Thus, the maximum time needed by a processor on *each level* has to be found in order to obtain the

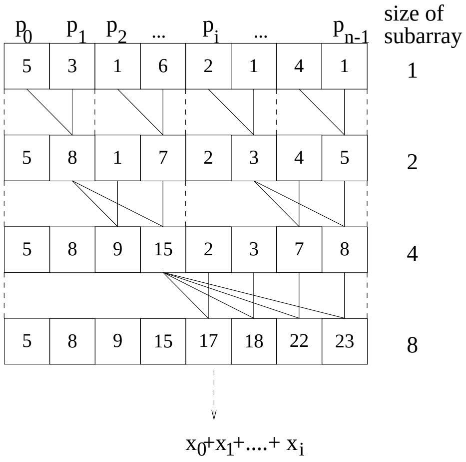


Figure 2: Prefix sums computation on a *CREW PRAM* with n processors

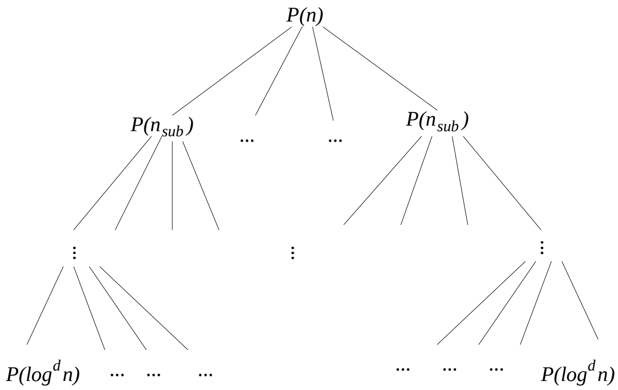


Figure 3: Recursive calls of a parallel algorithm illustrated as a recursion tree: the depth of the tree shows the number of recursive levels and the width depicts the number of parallel processes

running time of the parallel algorithm. The recursion usually goes down until the size of the subproblems is $O(log^k n)$, for some integer k. To these base cases a suboptimal algorithm is applied. This does not affect the total running time, since the problems are "small enough". In the deterministic setting the subproblems are of equal size and the maximum time needed for a subproblem can be bounded easily. In the randomized algorithms that will be presented later, we lose this property of equally sized subproblems, since the partitioning is dependent on some random numbers.

By applying random sampling a problem is divided into subproblems by independently choosing some random elements out of the input, called *samples*. The chosen r samples impose some properties (e.g. after sorting the samples $sample_i \leq$ input element $< sample_{i+1}$). That partitions the input into subsets of elements with the same properties. These subsets are referred to as *buckets*. Figure 4 depicts the partitioning through samples. Unfortunately, in the geometric setting we often do not have independent subproblems, i.e. an input element belongs to more than one bucket. In figure 4 the partitioning for some elements is shown, and element i falls into bucket $b_{i(0)}$ and $b_{i(1)}$. In these types of randomized recursive

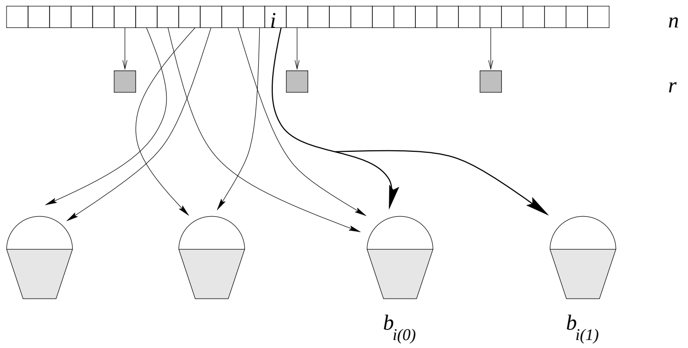


Figure 4: Partitioning of the input using samples results in subsets called *buckets*. In geometric settings partitioning can involve doubling of input elements.

algorithms (where doubling can occur), we do not only have to worry about the maximum size of each subproblem, but also about the sum of the subproblem sizes. Note, that we do not have the problem of doubling in a randomized sorting algorithm, as the operators $<$ and $\leq$ define a unique interval (bucket) for each input element.

The remainder of the report is organized as follows. Section 2 describes the technique of Polling using the example of intersection of half-planes in 2d. Intersection of half-spaces in $3d$ can also be computed efficiently with Polling. Section 3 gives some brief comments on this result. Finally, conclusions and open problems are stated in section 4.

2 Sampling Technique Polling and the Intersection of Half-Planes in Two Dimensions

In this section we will show how to compute the intersection of half-planes in two dimensions using the new sampling technique. Choosing one sample out of a set of $O(\log n)$ leads to a good sample w.h.p. In order to determine this sample the input is polled. To reduce the sum of the sizes of the subproblems further another technique called *Filtering* is applied.

Given a set S of n half-planes in two dimensions, the intersection of them is the part of $\mathcal{R}^2$ that is contained in each of the n half-planes. It can be easily seen that the intersection is convex and if it is bounded it is a convex polygon. The dual problem of the $2d$ intersections of half-planes is the *convex hull* of n points in two dimensions. The main result can be stated as follows:

Theorem 1 *Given a set S of n half-planes in $2d$. Then the intersection of S can be computed in time $\tilde{O}(\log n)$ on a* CREW PRAM.

Because of the duality be derive the following result for the convex hull.

Corollary 1 *Given a set S of n points in $2d$. Then the convex hull of S can be computed in time $\tilde{O}(\log n)$ on a* CREW PRAM.

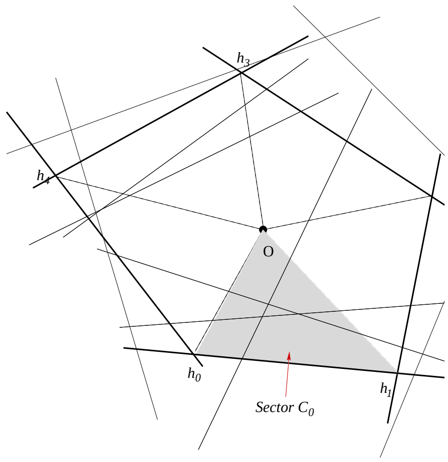


Figure 5: Dividing the problem space into sectors C_i by computing the intersection of the sample: The half-planes that intersect a sector C_i are the input for subproblem i. Doubling occurs if a half-plane intersects more than one sector.

Without loss of generality we can assume that the origin (or a known point) lies in the interior of the intersection. For the partition step we choose independently $r = \lfloor n^\epsilon \rfloor$, $0 < \epsilon < 1$, samples and construct the intersection of the sampled half-planes by a brute-force method. Then, after enumerating the vertices of the intersection in a cyclic order $h_0, h_1, \cdots$, we combine the vertices with the origin O and get the triangles O, h_i, h_{i+1}. These triangles are referred to as *sectors* C_i. Consider each sector C_i and determine the half-planes of the input Int_i that intersect the sector C_i. These subsets Int_i $(i = 0, \dots)$ are the input for the recursive call of each sector C_i. The output of the algorithm is the union of the intersections (represented by the vertices of the intersections) over all sectors. Note, that half-planes which intersect more than one sector are duplicated. This is necessary to consider the subproblems independently. Figure 5 shows the partitioning into sectors and algorithm 1 gives a more detailed description of the parallel algorithm.

Algorithm 1: IntersectHalfplane(S)
Input: Set S of n half-planes in $2d$ on a *CREW PRAM* with $O(n)$ processors.
Output: The vertices of the intersection of the half-planes of S.

1. Choose random subset *Sample* of $r = \lfloor n^\epsilon \rfloor$ half-planes, $0 < \epsilon < 1$;
2. Construct intersection of half-planes of *Sample*;
3. Number vertices of the intersection in a cyclic order: $h_0, h_1, \dots$;
4. Consider triangles O, h_i, h_{i+1} (sector C_i): determine half-planes Int_i intersecting sector C_i;
5. IF $|Int_i| > O(\log^k n)$ THEN
   call IntersectHalfplane(Int_i);
   ELSE determine intersections of Int_i with a polylogarithmic algorithm;
6. Output: Union of intersections over all sectors;

– end of algorithm –

2.1 Bounding the Problem Sizes

As mentioned earlier, to get the running time of an recursive parallel algorithm we need to determine the maximum size of a subproblem at each level. For the half-plane intersection we have to bound the maximum number of half-planes intersecting any sector. Furthermore, we have to take the width of the recursion tree into consideration. As we have a branching factor of $r = n^{\epsilon}$, $0 < \epsilon < 1$, which is dependent on the size of the parent problem n the width can become large, if there are many subproblems of size of the upper bound. Since in a parallel algorithm the width of the recursion tree represents the number of processors (note that one level of the tree is computed in parallel), we want to bound the width by $O(n)$ as only $O(n)$ processors are available. Moreover, we wish to achieve high probability results. Again, the objectives are:

1. bound maximum number of half-planes intersecting any sector w.h.p. AND
2. bound sum of the subproblem sizes w.h.p.

The following lemmata help a lot to solve the bounding problems. In fact, the first problem is solved by lemma 1, and we will see in theorem 2 that the bound of the maximum size of a subproblem $O(\frac{n}{r}\log r)$ is strong enough to get an overall running time of $O(\log n)$ with high probability. But this assumes that we do not "run out" of processors, i.e. we can bound the width appropriately. Lemma 2 gives us the desired bound but only as an expected value.

Lemma 1 *The probability that the maximum number of half-planes intersecting any sector exceeds $O(c \cdot \frac{n}{r}\log r)$ is less than n^{-c}.*

Lemma 2 *The expected value of the sum taken over all the sectors of all the half-planes intersecting a sector is $O(n)$.*

Let $\epsilon_0 > 1 - \epsilon$ be such that $O(n^{1-\epsilon}) = O(n^{\epsilon_0})$. Then the maximum problem size at level i is bounded by $O(n^{{\epsilon_0}^i})$. The following theorem states: If the algorithm needs $\tilde{O}(\log n_i)$ time for each stage, then it runs overall in time $\tilde{O}(\log n)$, whereas n_i is the maximum size of the subproblems in stage i.

Theorem 2 *If a procedure at depth i in the recursion takes time T_i such that*

$$Prob[T_i \geq kc\alpha \log(n^{\epsilon_0^i})] \leq n^{-c\alpha\epsilon_0^i}$$

then all the leaf-level procedures are finished in $\tilde{O}(\log n)$ time.

Although, the proof is not given here, the reader should note that the depth of the recursion is $O(\log\log n)$ since the branching factor is $\lfloor n^{\epsilon} \rfloor$ and we stop the recursion when the subproblem size is $O(\log^k n)$ for some integer k. (Recall that for constant branching factors the recursion depth is $O(\log n)$.)

Using lemma 1 and 2, applying Markov's inequality and choosing n big enough (so that $\frac{1}{k} + \frac{1}{n} < \frac{1}{2}$, $k > 2$ the factor in Markov's inequality) we get that both desired bounds hold with probability at least $\frac{1}{2}$. That is stated in the following lemma.

Lemma 3 *For some constants k_{total}, k_{max}, and large n, the following conditions hold with probability at least $\frac{1}{2}$:*

1. *The maximum number of half-planes intersecting any sector is less than $k_{max} \cdot \frac{n}{r} \log r \in O(n^{1-\epsilon} \log n)$*
2. *The sum (taken over all the sectors) of the number of half-planes intersecting a sector is less than $k_{total} n \in O(n)$.*

Definition 1 *A sample that leads to a partitioning of the input, so that 1. and 2. in lemma 3 are given, is called a "good" example. A sample which is not "good" is defined as "bad".*

2.2 Polling

We try to find a "good" example w.h.p., i.e., bound the sum of the subproblem sizes to $\tilde{O}(n)$, not loosing the first bound given by lemma 1.

Lemma 3 gives us an important property. Although it does not look like this, because the desired bounds are only achieved with probability at least $\frac{1}{2}$. But if we repeat the sampling independently $m \log n$ times, the probability that all samples are "bad" is less than $(\frac{1}{2})^{m \log n} = \frac{1}{n^m} = n^{-m}$. Therefore, by choosing $O(\log n)$ independent sample sets one of these sets is a "good" sample with high probability.

To obtain our two objectives, it is left to show how to detect this "good" sample. The easiest approach would be, to test the goodness of all of them, i.e. to determine the number of intersections in each sector and sum them up for each of the sample sets. But this is too time intensive. In fact, to determine if a sample is "good" requires $O(\log n)$ time with n processors. This leads to a total of $O(\log^2 n)$ time for testing all $O(\log n)$ samples, which is too much, since our objective is a $\tilde{O}(\log n)$ time algorithm.

By the way, to determine the sectors that a half-plane intersects, we apply Fibonacci search to the bimodal sequence ([3]) built of the distance from the vertices to the border of the half-plane and get the bounding sectors. The distance is negative on "one side" of the half-plane. This function is bimodal because of the convexity of the intersection of half-planes (in this case the sample half-planes). To compute the intersection of one half-plane with the sectors $O(\log |V|)$ time is needed, where V is the set of vertices of the intersection. Thus, we can compute the intersections of all n input half-planes with the sectors regarding *one* sample in parallel with n processors in $O(\log n)$ time.

The solution to the problem of determining a "good" sample, resembles a poll in real life. To determine if a candidate is good, it would be too much work to ask the whole population.

We just have to find a representative subset of the population and find out their favorite. For the problem of finding a "good" sample, it means that the goodness of a sample is determined by an estimation of intersections for each sector, whereas the estimation is done by taking only a fraction of the input half-planes. This idea gives the whole technique its name.

How large should this fraction be? It must be representative, i.e. large enough to make a valid decision w.h.p., but also small enough to be computed in $\tilde{O}(\log n)$ time. If we choose, for example, randomly $c_0 \cdot \frac{n}{\log^d n}$, $d > 2$ a fixed integer, $c_0 > 0$ a constant, (d and c_0 are determined by the desired success probability) of the input half-planes for *each* sample, it will lead w.h.p. to a "good" sample in respect to the whole input. For this fraction of the input only $O(\frac{n}{log^d n})$ processors are needed for one sample to compute in $O(\log n)$ time the intersections with the sectors. Thus, the intersections can be computed for all samples in parallel in $O(\log n)$ time with $O(n)$ processors.

Before we explain the details on how to choose a "good" sample, some notations are mentioned that will be used:
$Sample_j$: jth sample
Int_i^j: number of half-planes of the input intersecting sector C_i of sample $Sample_j$
$IntPoll_i^j$: number of half-planes of polled input intersecting sector C_i of $Sample_j$

It is easy to see, that $IntPoll_i^j$ is a binomial random variable with parameters $\frac{Int_i^j}{n}$ (the success probability) and $c_0 \cdot \frac{n}{\log^d n}$ (the number of trials). Since $IntPoll_i^j$ is a binomial random variable we can apply Chernoff bounds to bound the number of intersections of the polled input per sector for each sample for the case, that the number of intersections regarding the whole input is "large" (case 2). When the number of intersections for the whole input is "small" (case 1), we do not have to try to bound the number, since for the sum of the intersections (desired to be $O(n)$) the number of intersections is small enough.

Case 1: $Int_i^j \le c \cdot \log^{d+1} n$
$\Longrightarrow n^\epsilon \cdot c \log^{d+1} n = o(n)$
$\Longrightarrow$ sum of the subproblems $\le$ size of parent problem

Case 2: $Int_i^j > c \cdot \log^{d+1} n$
Chernoff bounds $\Longrightarrow \text{Prob}(IntPoll_i^j \le \alpha c_1 \frac{Int_i^j}{\log^d n}) \le n^{-\alpha}$ and

$$\text{Prob}(IntPoll_i^j \ge c_2 \alpha c_0 \cdot \frac{Int_i^j}{\log^d n}) < n^{-c_0\alpha} < n^{-\alpha} \text{ (for } c_0 > 1)$$

Using these bounds with appropriate constants one can show that with high probability Int_i^j are bounded for all i sectors simultaneously

- from below by $LowIntSec^j := IntPoll_i^j \cdot \frac{log^d n}{c_0 c_2 \alpha} \in \Omega(\log^d n \cdot IntPoll_i^j)$ and
- from above by $UpIntSec^j := IntPoll_i^j \cdot \frac{log^d n}{c_1 \alpha} \in O(\log^d n \cdot IntPoll_i^j)$

That means, that the number of intersections per sector for one sample regarding the whole input is w.h.p. bounded from above *and* below by a factor of $\log^d n$ times the number of intersections obtained from the polled input. To find a "good" sample, we determine the number of intersections per sector for each sample regarding the polled input and compute the upper and lower bound for the intersections of the whole input. Then, we compute a lower and upper bound for the sum of the intersections over all sectors which are $LowInt^j := n^\epsilon \cdot LowIntSec^j$ and $UpInt^j := n^\epsilon \cdot UpIntSec^j$ for each sample j. To determine for a sample whether it is good, we compare these bounds with the maximal allowed number for the sum of the intersections $O(n)$, or more precisely, $k_{total} \cdot n$. Recall that the definition of a "good" sample is based on the *exact* number of intersections for the whole input. Since we have only upper and lower bounds for this number we have to distinguish 3 cases:

- If the upper bound of the sum of intersections regarding the whole input is less than the allowed amount, we have found a "good" sample, since the actual value is below the upper bound and, therefore, below the allowed value.
- If the lower bound is greater than the allowed value, then this is a "bad" sample, as the actual value is larger than the lower bound and, therefore, larger than the allowed number.
- If the allowed amount lies in between the upper and lower bound, then the actual value differs only by a constant factor from the allowed value, since the actual value also lies between lower and upper bound and these bounds differ only in a constant factor. In this case the sample with the less intersections regarding the polled input is chosen.

Recall that there is a "good" sample (cases 1 and 3) with high probability. The algorithm to find a "good" sample is executed for all samples in parallel.

Algorithm 2: Finding a "good" sample

Input: Samples $Sample_1, ..., Sample_m$, $m = O(\log n)$.
Output: A "good" sample $Sample_{j_0}$.

/* case: clearly good */
1. IF $k_{total} \cdot n > UpInt^j$ THEN
accept sample $Sample^j$;

/* case: clearly bad */
2. IF $k_{total} \cdot n < LowInt^j$ THEN
sample is "bad";

/* case: choose the best */
3. IF $LowInt^j \leq k_{total} \cdot n \leq UpInt^j$ THEN
accept sample $Sample_{j_0}$ for which $\sum_i IntPoll_i^j$ is minimum;

– end of algorithm –

Lemma 4 (Polling Lemma) *Using the above algorithm a "good" sample is obtained with high probability.*

2.2.1 Bounding the Number of Processors

Finally, we found a "good" sample, so that not only the maximum number of intersections per sector is small enough, but also the sum of the sizes over all subproblems is linear in the size of the parent problem. Unfortunately, this is not small enough as the following consideration shows. The recursive depth of the algorithm is $O(\log\log n)$. In each level i we choose a "good" sample for each problem P on this level, which divides the problem so that

$$\text{sum of size of subproblems} = c_{i,P} \cdot \text{size of parent problem}$$

Let c_{min} be the minimum of all these constant factors $c_{i,P}$. Then, at *each* of the $O(\log\log n)$ levels, the sum of the subproblems increases by a factor of at least c_{min}. That leads over $O(\log logn)$ levels to an increase of $n \cdot (c_{min})^{\log\log n} = n \cdot \log^{\log c} n \in O(n \cdot \log^b n)$, b a constant.

To be able to run the algorithm we have the following opportunities:

- Increase the number of processors: That would reduce the quality of the algorithm, since we are looking for an algorithm that runs on an $O(n)$ processor *CREW PRAM* and not on one with $O(n \log^b n)$ processors.
- Simulate the missing $O(\log^b)$ processors: That increases the running time by a factor of $O(\frac{\log^b n}{n})$. Since we already need $\tilde{O}(\log n)$ time for each level, that means we exceed our objective time of $\tilde{O}(\log n)$ for each level.
- Bound the sum of subproblems more tightly, so that the sum of subproblems on each level equals $c \cdot n$, where c is a constant which is the same for each level and n is the original input size. This is the only opportunity that does not violate with our objective of needing $\tilde{O}(\log n)$ time for each level.

2.3 Bounding the Sum of the Subproblems More Tightly: ***Filtering***

After finding a "good" sample we have partitioned the problem into subproblems, so that the sum of the sizes of the subproblems exceeds the size of the parent problem by a constant factor. This is the input for the *Filtering* procedure. The output are subproblems, where the sum of the subproblems of *each level* is a fixed size, e.g., $c' \cdot (input + output) = c \cdot n$, c' and c constants. To achieve this output geometric properties of the specific problem are used.

Before we explain the details of how the *Filtering* works for the intersection of half-spaces in $2d$, we finish the overall consideration of the algorithm. The *Filtering* results in an algorithm that needs at each level $c \cdot n = O(n)$ processors. Combined with earlier statements we obtain an algorithm for the intersection problem that needs for each level $\tilde{O}(\log n)$ time and $O(n)$ processors, so that with theorem 1 we get a total time of $\tilde{O}(\log n)$ on an $O(n)$ processor *CREW PRAM*. A more detailed analysis of the running time is given in section 2.5.

We now describe the *Filtering* procedure for the intersection of half-planes in $2d$. We can observe that the output size for the problem is less or equal the input size, since in the maximal output each half-plane contributes to two vertices and each vertex results of an intersection of two half-planes. Therefore, we eliminate half-planes from the input for

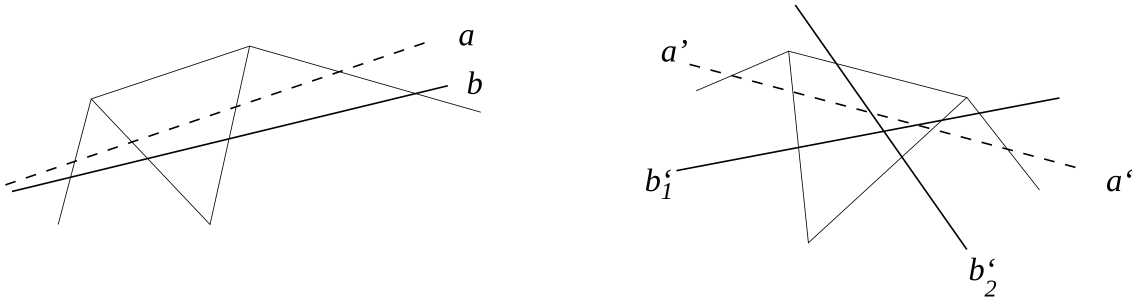


Figure 6: Case 1 (left): The border of the half-plane a is totally covered by another half-plane b. Case 2 (right): The border of the half-plane a' is covered by more than one half-plane, namely b'_1 and b'_2.

a sector that are not part of the output for this sector. To do so, we consider each sector and distinguish 4 cases: (The terms "visible" and "covered" consider a view from the origin, which was assumed to be in the interior of the intersection.)

Case 1: The border of a half-plane a in a sector C_i is totally covered by another half-plane b (figure 6,left). Then a can not be part of the output for this sector C_i. Hence, it is deleted from the input of the recursive call referring to this sector.

Case 2: The border of a half-plane a' is a sector C_i is covered by the action of two half-planes (figure 6, right). Again, a' can not be part of the output for this sector, but this case is more difficult to determine, so that half-plane a' will remain in the input for this sector.

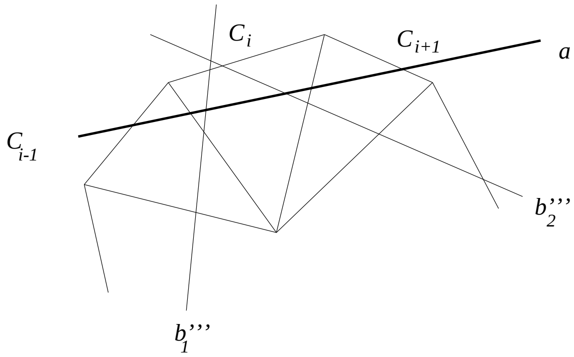
Case 3: A half-plane a''' is visible in at most one sector C_i (figure 7. This half-plane a''' will only be part of the input in the recursive call for sector C_i.

Case 4: A half-plane $a^{(iv)}$ is visible in more than one sector, namely in consecutive sectors $C_i, \ldots, C_j$ (figure 7. Then, $a^{(iv)}$ will be part of the input for the bounding sectors C_i and C_j. The subproblems for sectors C_{i+1} to C_{j-1} are solved and need no further recursive calls.

To determine case 1 we consider the intersections of half-planes with the line segments O, h_i and O, h_{i+1}, i.e. the "sides" of sector C_i. If the border of a half-plane a is totally covered by one other half-plane b in sector C_i, both intersections of b with the sides of sector C_i will appear before those of half-plane a, when we consider the intersections with the sides in sorted order (regarding increasing distance from origin 0).

For each half-plane a' case 2 can only be true in one sector, since in the other sectors a' is totally covered by one half-plane (case 1). The same applies for case 3. Cases 2 and 3 are treated in this stage equally, but in further partitions case 2 will resolve in case 1 and case 3 will resolve in case 4.

In case 4 half-plane $a^{(iv)}$ will contribute to vertices in the bounding sectors C_i and C_j. In the sectors between C_i and C_j there cannot be another intersection without violating that $a^{(iv)}$ is visible in the bounding sectors (this results from the convexity of the intersection).



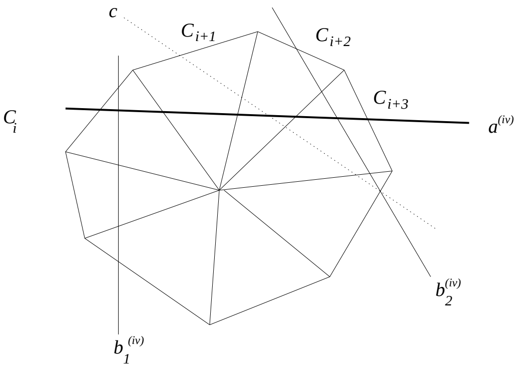


Figure 7: Case 3 (left): The half-plane a''' is only visible in sector C_i. In the other sectors the border of a''' is totally covered by half-plane b_1''' (to the left) and by b_2''' (to the right). Case 4 (right): The half-plane $a^{(iv)}$ is visible in sectors $C_i, \ldots, C_{i+3}$. It covers the border of other half-planes in the sectors C_{i+1} and C_{i+2} totally. If a half-plane c was visible in a sector between the bounding sectors, say C_{i+2}, it would cover $a^{(iv)}$ totally in one of the bounding sectors (here C_{i+3}).

Therefore, in the sectors C_{i+1} to C_{j-1} half-plane $a^{(iv)}$ covers totally all the other half-planes, which can be determined easily as in case 1.

All in all, at any stage there are at most 2 copies of a half-plane, and the sectors that get these copies can be determined in time $\tilde{O}(\log n)$ (since we only compute case 1).

The following processor allocation strategy assigns for each half-plane two processors, one for each vertex of intersection, that the half-plane might contributes to. It follows the above observations and guarantees that no reallocation of processors is necessary.

Case 1: no processors

Case 4: allocate 1 processor at each bounding sector

Unsolved cases (case 2 and case 3): allocate 2 processors

At each level at most $2n$ processors are needed.

2.4 General Strategy

To review the *Polling* technique, the general technique is listed below. For each problem that the technique is applied to, the earlier mentioned Lemma 3 and the *Filtering* have to be shown individually for each problem.

1. Select $O(\log n)$ samples, each of size $\lfloor n^\epsilon \rfloor$ for some $0 < \epsilon < 1$.
2. Select a "good" sample using *Polling*.
3. Divide the problem into subproblems using the "good" sample.
4. Use a *Filtering* scheme to bound the sum of the subproblems to a fixed size (e.g. output size+ input size). Purpose: bound the number of processors
5. IF the subproblem is small enough (usually $O(\log^k n)$, $k > 1$ an integer) THEN use a direct method to solve the subproblem

ELSE call algorithm recursively

2.5 Time Complexity

Considering the general strategy of the previous section, we review the time complexity of each step for the intersection of half-planes in $2d$.

With high probability:

1. $O(\log n)$ coin flips on each processor: $O(\log n)$ time
2. intersection of each sample (brute-force: compare all possible intersections with equations of half-planes): $O(\log n)$ time
   intersection of "polled" input ($\frac{n}{\log^d n}$) with sectors built by each sample: $O(\log n)$ time (Fibonacci search on bimodal sequence)
3. intersection of (whole) input with sectors built by "good" sample: $O(\log n)$ time (Fibonacci search on bimodal sequence)
4. $\tilde{O}(\log n)$ time
5. base case: $\tilde{O}(\log n)$ time

We obtain that the algorithm needs $\tilde{O}(\log n)$ time for each iteration. Theorem 1 leads to the overall time complexity of $\tilde{O}(\log n)$ with $2n$ processors on a CREW PRAM which is the statement in theorem 2.

3 Intersection of Half-Spaces in Three Dimensions

The *Polling* technique can also be applied to compute the intersection of half-spaces in three dimensions [1]. This is an interesting problem, since based on its solution several other problems can be solved:

- 3d convex hull (dual)
- Voronoi diagram
- all-points nearest neighbor
- Euclidean minimal spanning tree

The algorithm for the $3d$ half-space intersection leads to the same time and processor bounds as the $2d$ case. The following result can be stated:

Theorem 3 *The above problems can be solved in $\tilde{O}(\log n)$ time on a $O(n)$ processor CREW PRAM.*

The following section gives brief remarks on how to partition the problem in the $3d$ case.

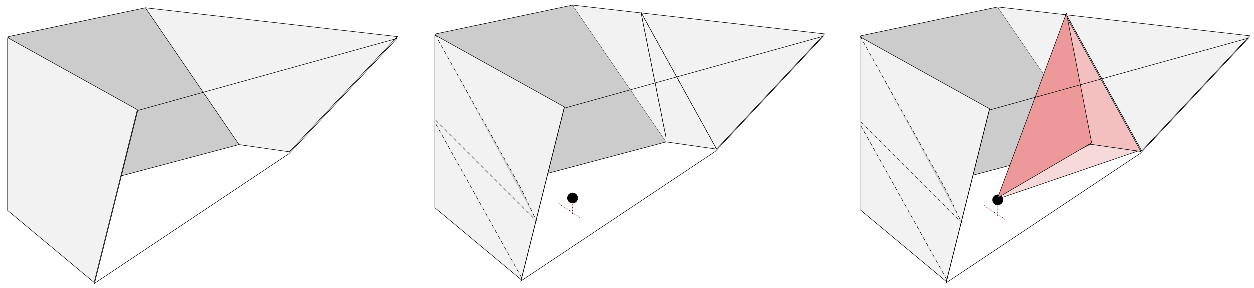


Figure 8: The division of the problem into sublems by cones.

3.1 Partitioning the Problem

For the $3d$ case lemma 3 can be proven, too. So that we proceed similarly as in the $2d$ case. We choose $O(\log n)$ samples, each of size $\lfloor n^{\epsilon} \rfloor$. Out of this set of samples, a "good" sample is chosen. A sector is equivalent to a *cone*in the $3d$ case. These cones are built as follows. First determine a point O that is in the interior of the intersection. The intersection of a sample is computed, which are the faces $F_0, \cdots$. Using a plane that is not parallel to any of the half-spaces in the sample, the faces are cut into trapezoids. These are divided into triangles. Number the vertices of the triangle and combine the vertices of a triangle with the interior point O (figure 8). The half-spaces that intersect a cone form a subproblem, which are bound more tightly by a filtering procedure. In general, the strategy is the same, but the individual steps, especially determining intersections with the cones, are more difficult.

4 Conclusions and Open Problems

The main ideas of the random sampling technique *Polling* for geometric problems can be summarized as:

- determine a "good" sample out of $O(\log n)$ samples by testing only a fraction of the input
- bound sum of subproblems on each level to a fixed (level independent) size by *Filtering*

Several problems can be solved applying this technique. The main results of [2] and [1] are: intersection of half-spaces in $2d$ and $3d$, convex hull in $2d$ and $3d$, and $2d$ Voronoi diagram can be solved in $\tilde{O}(\log n)$ time on a CREW PRAM with $O(n)$ processors.

Further research is recommended in applying the polling technique to other problems. It is open, if an equivalent algorithm for the *EREW PRAM* exists.

References

[1] J. H. Reif, S. Sen, "Polling: A New Randomized Sampling Technique For Computational Geometry", *SIAM Journal on Computing*, June 1992

[2] S. Rajasekaran, S. Sen, "Random Sampling Techniques and Parallel Algorithm Design", *Synthesis of Parallel Algorithms*, ed. J. H. Reif, 19 93

[3] B. Chazelle, D. Dobkin, "Intersection of Convex Objects in Two and Three Dimensions", *Journal of the ACM*, Vol. 34(1), 1987, pp. 1-27.