In-Place Techniques for Parallel Convex Hull Algorithms
(Preliminary Version)

Mujtaba R. Ghouse* and Michael T. Goodrich†

Dept. of Computer Science
Johns Hopkins University, Baltimore, MD 21218-2686

Abstract

We present a number of efficient parallel algorithms for constructing 2- and 3-dimensional convex hulls on a randomized CRCW PRAM. Specifically, we show how to build the convex hull of n pre-sorted points in the plane almost surely in $O(1)$ time using $O(n \log n)$ processors, or, alternately, almost surely in $O(\log^* n)$ time using an optimal number of processors. We also show how to find the convex hull of n unsorted points in $\Re^2$ (resp., $\Re^3$) in $O(\log n)$ time using $O(n \log h)$ work (resp., $O(\log^2 n)$ time using $O(\min\{n \log^2 h, n \log n\})$ work), with very high probability, where h is the number of edges in the convex hull (h is $O(n)$, but can be as small as $O(1)$). Our algorithms for unsorted input depend on the use of new *in-place* procedures, that is, procedures that are defined on a subset of elements in the input and that work without re-ordering the input. For the pre-sorted case we also exploit a technique that allows one to modify an algorithm that assumes it is given points so that it can be used on hulls; we call such algorithms *point-hull invariant.*

*This research supported in part by NSF and DARPA under Grant CCR-8908092. Email: ghouse@cs.jhu.edu.

†This research supported in part by the National Science Foundation under Grant CCR-9003299, and by NSF and DARPA under Grant CCR-8908092. Email: goodrich@cs.jhu.edu.

1 Introduction

The problem of constructing the convex hull of a set of n points is perhaps the most studied problem in computational geometry (see [3, 4, 7, 10, 18, 19, 26, 31]). The problem is generally defined as that of constructing the boundary of the smallest convex set containing all n points. The 2-dimensional convex hull can be constructed sequentially in $O(n)$ time [27], assuming the input points are pre-sorted (say, by increasing x-coordinates), which of course can be done in $O(n \log n)$ time [27]. The 3-dimensional convex hull can also be constructed in $O(n \log n)$ time [27]. These sequential algorithms are in fact optimal, since the 2-dimensional convex hull problem has an $\Omega(n \log n)$ lower bound [34]. Nevertheless, Kirkpatrick and Seidel [21] show that one can beat this lower bound in some cases, in that they give a 2-dimensional convex hull algorithm that runs in $O(n \log h)$ time, where h is the size of the output. Strictly speaking, their method does not contradict the lower bound arguments, as these arguments assume that the output size is linear. Similarly, for the 3-dimensional case, Edelsbrunner and Shi [16] show that one can beat the lower bound in some cases, by giving an algorithm that runs in time $O(n \log^2 h)$. In both the 2- and 3-dimensional case, h is $O(n)$, but can be as small as $O(1)$. Algorithms such as those of Kirkpatrick and Seidel [21], and Edelsbrunner and Shi [16], are said to be *output-size sensitive.*

Convex hull construction has also been well-studied in parallel. For the 2-dimensional convex hull problem, Chow [12] achieved $O(\log^2 n)$ time with n processors, and optimality was achieved by the deterministic, $O(\log n)$ time, n processor algorithm of Atallah and Goodrich [5], Aggarwal *et al* [1] and the simpler (deterministic) technique of Atallah and Goodrich [6]. All these results were on the CREW PRAM, but Miller and Stout [25] achieved the same result on the EREW

PRAM. For the 3-dimensional problem, both Chow [12] and Aggarwal *et al* [1] achieved $O(\log^3 n)$ time, with n processors, and Dadoun and Kirkpatrick [15] achieved $O(\log^2 n \log^* n)$ time with n processors. Reif and Sen [30] give a randomized algorithm that achieves $O(\log n)$ expected time with very high probability [1], using n processors. For the case in which the input is presorted, Berkman *et al* [8] achieved $O(\log \log n)$ time with optimal work. Stout [33] gives a randomized algorithm that achieves constant expected time with n processors for convex hull assuming a uniform input distribution. However, as Stout observes, given such a uniform distribution, the expected number of points on the convex hull is $O(\log n)$, which allows "brute-force" constant-time techniques to be used that would not be constant-time otherwise.

None of these previous algorithms achieve the sequential output-sensitive work bounds described above, however. Nevertheless, using standard parallel techniques, one can implement the algorithm of Kirkpatrick and Seidel [21] in $O(\log^2 n)$ time, using $O(n \log h)$ work. An analogous implementation of the method of Edelsbrunner and Shi would run in $O(\log^3 n)$ time, using $O(n \log^2 h)$ work. It is not clear how one can improve these times without the introduction of new techniques, however.

In this paper we present efficient parallel algorithms for convex hull construction in 2- and 3-dimensions. We address both the pre-sorted and unsorted cases. For the pre-sorted case we give an algorithm that runs almost surely[2] in $O(1)$ time using $O(n \log n)$ processors. We then show how to modify our algorithm so that, even though it assumes the input is a set of points, it can be implemented to run on a set of upper hulls[3]. We call such algorithms *point-hull invariant*, and show how such an algorithm can be used to derive a solution to the convex hull problem running almost surely in $O(\log^* n)$ time with an optimal number of processors. Our analysis does not depend on any assumptions about the input distribution.

For the unsorted case, we give efficient output-sensitive parallel methods for both the 2- and 3-dimensional convex hull problems. Our algorithms attain work bounds that are lower than previous bounds, and match those attained sequentially by Kirkpatrick and Seidel [21] for the 2-dimensional problem, and by Edelsbrunner and Shi [16] for the 3-dimensional problem. Specificallly, we show how to solve the 2-dimensional convex hull problem in $O(\log n)$ time using $O(n \log h)$ work with very high probability, and how to solve the 3-dimensional convex hull problem in $O(\log^2 n)$ time using $O(\min\{n \log^2 h, n \log n\})$ work, again, with very high probability.

All of our methods depend upon the use of new *in-place* techniques, whereby we mean methods that are defined on a subset S' of elements in the input and work without re-ordering the input. Intuitively, we have a virtual processor "standing by" each element in S', and this virtual processor does all the work necessary because of the inclusion of this element in S'. The significance of these techniques is that they allow one to perform each level in a parallel divide-and-conquer scheme very fast (in our case, $O(1)$ time with very high probability) without ever needing to explicitly perform the "divide" step. Instead, one simply divides the subproblems logically, and by keeping a virtual processor with each element e, we can associate e with the correct subproblem. This contrasts with most previous parallel techniques, which require that the elements in a subproblem belong to a contiguous portion of some array.

Our methods are also based on various adaptations of a method of Alon and Megiddo [2] for performing linear programming in fixed dimensions almost surely in $O(1)$ time using $O(n)$ processors in a randomized CRCW PRAM model. The general approach of our methods is based on the approaches of [21] and [16]: namely, to use linear programming to "probe" the convex hull, finding a facet about which we may then split the problem and recurse. By adapting Alon and Megiddo's method to be both point-hull invariant and in-place we allow for such probes to be repeated recursively.

In addition, as we use algorithms on subproblems that have confidence bounds dependent on subproblem size, in order to achieve confidence bounds dependent on the total problem size we employ a technique we call *failure sweeping*. Intuitively, we "sweep" those subproblems that have not been solved within the desired time into a limited space m, then solve them all with a number of processors that is super-linear with respect to m.

In Section 2 we present our method for the presorted case, which in turn motivates our approach for the unsorted case, which we describe in Section 4. Before we describe our methods for the unsorted case, however, we first present a number of general in-place techniques

[1] *Very high probability* - the probability that this will not occur $\leq n^{-c}$, where $c \geq 1$ is a constant.

[2] *Almost surely* - the probability that this will not occur $\leq c^{-n^d}$, where $c > 1$, $d > 0$.

[3] An *upper hull* is a convex chain monotone in the x-direction that "curves to the right" as one traverses it by increasing x-coordinates.

in Section 3. We follow our algorithm description with a discussion of processor allocation issues in Section 5, and conclude in Section 6.

2 Convex Hull Algorithms for Presorted Input

Given n presorted points in the plane (assume, wlog, that they are sorted in increasing order of x-coordinates) in an array, we find their upper hull such that every point in the array has a pointer to the hull edge that it is beneath (or on). Thus, one edge may occur in this list many times, as it will be stored by every point below it.

2.1 Preliminaries

First, we review some elegant results due, respectively, to Ragde, to Alon and Megiddo, and to Chernoff, that we use in our algorithm:

Lemma 2.1 (The Approximate Compaction Lemma (Ragde [28])): *Given an array of size n containing at most k non-zero elements, one can determine whether $k < n^{\frac{1}{4}}$ and if so, one can compress these k elements into an area of size k^4, all in constant time on a CRCW PRAM (deterministically) with n processors.*

The next lemma deals with the linear programming problem: given m constraints (half-spaces) in $\Re^d$, and some particular linear (objective) function, finding the point in $\Re^d$ that maximizes the objective function, subject to the constraints.

Lemma 2.2 (Alon and Megiddo [2]): *Given n constraints in $\Re^d$, linear programming can be performed in constant time with n processors on a CRCW PRAM, with the probability of taking more than some constant amount of time given by $2^{-cn^{\frac{1}{3}}}$, where $c > 1$ is a constant.*

Lemma 2.3 (Tail Estimation (Chernoff [11], and, in this form, Raghavan [29])): *Let $X_1, X_2, \ldots, X_n \in \{0,1\}$ be n bernoulli trials: independent trials with probability p_i that $X_i = 1$, where $p_i \in (0,1)$. Define $X = \Sigma_{i=1}^{n} X_i$, $\mu = \Sigma_{i=1}^{n} p_i$. Then, for all $\delta > 0$,*

$$Prob(X > (1+\delta)\mu) < \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu},$$

and, for all δ such that $0 < \delta \leq 1$,

$$Prob(X < (1-\delta)\mu) < \left[\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right]^{\mu}.$$

We also make use of the following observations:

Observation 2.1 (Eppstein and Galil [17]) *The first non-zero element of an array of size n can be found in constant time on a CRCW PRAM, with n processors.*

Observation 2.2 (Brute Force Linear Programming) *It is possible to solve linear programming in d dimensions in constant time, with n^{d+1} processors.*

Proof sketch: We find the intersection of all d-tuples of constraints, then for each such tuple, check whether its intersection, which is a candidate solution, is violated by any other constraint in the subproblem. ∎

Observation 2.3 (Brute Force Convex Hull) *It is possible to find the upper hull of n points in the plane in constant time with n^3 processors.*

Definition *The* bridge *is the upper hull edge that intersects the vertical line through one specified point.*

Kirkpatrick and Seidel [21] observed the following:

Observation 2.4 (Linear Programming for Bridge Finding) *The problem of finding a bridge can be reduced to, and hence solved by, linear programming.*

The final result we review is that of constant-time convex hull determination using an inefficient number of processors.

Lemma 2.4 *For any integer $k \geq 1$, one can find the upper hull of n points in the plane in time $O(k)$, using $n^{1+\frac{1}{k}}$ processors, deterministically, on a CRCW PRAM.*

Proof: This result is part of the "folklore" of parallel computational geometry. For completeness, however, we include the details in the final version. ∎

2.2 A More Efficient Algorithm

Having reviewed the above results, we now present an $n \log n$ processor, constant time, algorithm. Suppose we are given n input points in the plane, such that they are sorted in increasing order of x-coordinate. We consider a complete binary tree T, built "on top" of these points, and note that finding the bridge for each median point in the subtree of every node v in T would result in finding all the hull edges.

Given $n \log n$ processors in a randomized CRCW PRAM, we can, for each v in T, simultaneously solve the associated linear programming problem (see Observation 2.4), using Alon and Megiddo's algorithm (see Lemma 2.2).

Having done this, we then assigned $\log n$ processors to each node v, and check whether any of v's ancestors store a bridge that covers (i.e. is the bridge for) the solution to v, in constant time (this amounts to an OR). Finally, assigning $\log n$ processors to each point (leaf)

p, we find the lowest ancestor of p that is not covered by another edge: this will be the edge above p. (See Observation 2.1.)

Thus, it might seem that, as Alon and Megiddo's algorithm takes constant time and has confidence bounds that are exponential in the size of the input, we have achieved our result. This is not the case, however, as Alon and Megiddo's algorithm is, in general, not used on the entire input but on subproblems, some of which are very small. Since the confidence bounds are only exponential in m, the size of these subproblems, they are not exponential in n. (For example, if $m = O(\log n)$, then the confidence bounds are only polynomial in n, and for smaller subproblems they are not even polynomial in n.)

We overcome this lack of confidence by using Lemma 2.4 for any problems containing fewer than $\log^3 n$ points, and by using a technique that we call *failure sweeping* for problems of size m such that $\log^3 n \leq m \leq n^{\frac{1}{4}}$.

Before we describe our failure sweeping technique, however, let us analyze the processor bounds needed so far. For small problems, that is to say, problems of size $m < \log^3 n$ we use an $m^{\frac{4}{3}}$ processor, constant time, algorithm (see lemma 2.4, with $k = 3$). So the number of processors required for each level of height $h < 3 \log\log n$ in the tree, $(m = 2^h)$ is $m^{\frac{4}{3}} n/m = nm^{\frac{1}{3}}$. As the highest level that uses brute force has $m < \log^3 n$, the total number of processors required is $< n \log n(1 + \frac{1}{2^{\frac{1}{3}}} + \frac{1}{4^{\frac{1}{3}}} + \ldots)$ which is $O(n \log n)$. Note that all these small problems are solved deterministically in constant time. All other problems are solved using Alon and Megiddo's algorithm, with a linear number of processors in constant time, with probability of failure $\leq 2^{-cm^{\frac{1}{3}}}$, where $c > 1$ (see Lemma 2.2). Thus, all these other problems require n processors per level, and hence $O(n \log n)$ processors in total.

2.3 Improving Confidence by Failure Sweeping

In this subsection, we describe *failure sweeping*: a technique for improving the confidence bounds of an iterative or recursive randomized algorithm.

Consider a randomized algorithm with expected running time $t(n)$ when run on input of size n, and which has a failure probability $p(n)$ so the probability of taking longer than $ct(n)$ for some constant c, is $p(n)$. Also, assume that there is a "brute-force" technique for solving this problem with a super-linear, but polynomial, number of processors. (For the sake of concreteness, let us say we have an n^3 processor "brute-force" method.) If this algorithm is run on $\frac{n}{m}$ subproblems, each of size $m = \Omega(p^{-1}(n))$ (such that $p(p^{-1}(n)) = n$), then failure sweeping can improve the failure probability from $p(m)$ to $p(n)$, and thus improve the confidence from $1 - p(m)$ to $1 - p(n)$.

This is achieved as follows: the algorithm is run for $ct(m)$ steps on the subproblems, so the expected number of *failures* is $\frac{n}{m} p(m) \leq 1$, where a *failure* is a subproblem that has not yet been solved. The failures are then compacted into an area of size $n^{\frac{1}{r}}$, where $r \geq 4$, using Ragde's algorithm (see lemma 2.1). Then, we assign $n^{1-\frac{1}{r}}$ processors to each failure, and use a "brute-force" method to solve them.

In our problem, we use Alon and Megiddo's algorithm on all problems of size $\geq \log^3 n$, and perform failure sweeping on each level of the binary tree that is of height h such that $3 \log\log n \leq h \leq \frac{1}{4} \log n$. From Lemma 2.2, we have $p(n) = 2^{-cn^{\frac{1}{3}}}$, and from Observation 2.2, we have an appropriate brute force technique, so we can perform failure sweeping.

Now, we use Ragde's compaction algorithm, from Lemma 2.1, which will attempt to compact the failures into a space of size $n^{\frac{1}{4}}$. Then, we use the brute force algorithm of Observation 2.2, with $n^{\frac{3}{4}}$ processors assigned to each failure. (This is a sufficient number of processors for all problems of size $m \leq n^{\frac{1}{4}}$ as the brute force technique requires m^3 processors.)

Analysis

After spending some constant α amount of time on Alon and Megiddo's technique, at each level the expected number of failures is $\mu = \frac{n}{m} 2^{-cm^{\frac{1}{3}}}$, but as $m \geq \log^3 n$, we have $\mu \leq \frac{1}{n^{c-1}}$, so $\mu \leq 1$.

Therefore, by the Chernoff bounds (Lemma 2.3), the number of failures is greater than $n^{\frac{1}{16}}$ with probability

$f \leq \left(\frac{e^{n^{\frac{1}{16}}-1}}{(n^{\frac{1}{16}})^{n^{\frac{1}{16}}}} \right)$, which implies

$f \leq \frac{1}{e} \left(\frac{n^{\frac{1}{16}}}{e} \right)^{-n^{\frac{1}{16}}}$,

$f \leq \frac{1}{e} 2^{-n^{\frac{1}{16}}(\frac{1}{16}\log n - \log e)}$.

So the number of failures is less than $n^{\frac{1}{16}}$ with probability

$1 - f \geq 1 - \frac{1}{e} 2^{-n^{\frac{1}{16}}}$

for all n greater than a constant, g, which is exponentially small in n.

We then use Ragde's approximate compaction method, which will succeed if there are not more than $n^{\frac{1}{16}}$ failures, which is true with probability $1 - f$.

This gives us the following lemma:

Lemma 2.5 *The 2d convex hull problem can be solved in constant time with $O(n \log n)$ processors, with probability $\geq 1 - 2^{-n^{\frac{1}{16}}}$, for all n larger than a constant, g.*

Over the next three subsections, we show how to use this result to derive an $O(\log^* n)$ time, optimal algorithm.

2.4 Making Alon and Megiddo's Method Point-Hull Invariant

Here, we define the property of *point-hull invariance* and we show that our application of Alon and Megiddo's linear programming algorithm [2] is point-hull invariant.

The algorithm of Alon and Megiddo [2] for linear programming in any fixed dimension comprises two parts: repeatedly choosing a subset of the constraints, and finding the solution to this subset of constraints, as a linear programming problem. The initial subset is chosen at random from all the constraints, and later choices are made at random from those that violate the currently known solution. These can be found by considering the current solution (a point s), for every constraint (a line c). The constraint c violates s iff s is not *above* c.

To find the solution to the subproblem, one uses brute force (see observation 2.2). The subset is chosen to be so small that this can be done in constant time with n processors.

The only primitive operations required are the following:

- choosing a sample at random;
- intersecting two lines; and
- determining whether a point is above or below a line.

Since we wish to apply their method to objects that are themselves upper hulls, we note that these operations on points and lines have corresponding operations on upper hulls. In particular,

- finding the x (or y) coordinate of a point, or determining on which side of a line a point lies, corresponds to finding the intersection of a line with a upper hull;
- finding the line defined by two points corresponds to finding the common tangent of two upper hulls; and

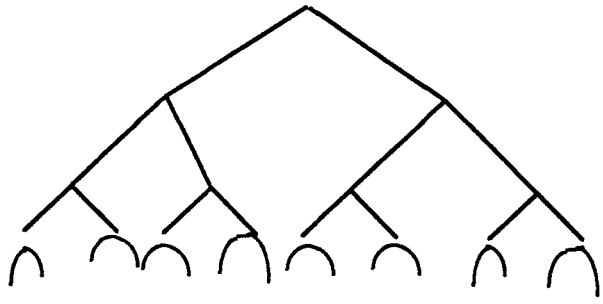

Figure 1: The Use of Point-Hull Invariance.

- finding the intersection of two lines corresponds to finding the intersection of two hulls (assuming, of course, that one knows there can be only one intersection).

Any algorithm on n points that would require only these operations to function with n upper hulls as input instead, is defined as *point-hull invariant.* See Figure 1. Atallah and Goodrich [6] show that each of these upper hull problems can be solved in constant time with $O(n^{\frac{1}{b}})$ processors, where b is a constant and n is the total number of edges in the upper hull.

This gives us the following observation:

Observation 2.5 Both the brute force algorithm and Alon and Megiddo's technique (of Observation 2.2 and Lemma 2.2, respectively) are point-hull invariant. In addition, the constant time upper hull algorithm (of Lemma 2.5) is also point-hull invariant.

As a result, given m upper hulls, each containing q points, we can run the constant time convex hull algorithm of lemma 2.5, using a modification of Alon and Megiddo's algorithm such that in place of constant time calls to the trivial operations on points, we call the constant time operations on upper hulls due to Atallah and Goodrich [6]. This gives us the following lemma:

Lemma 2.6 The algorithm obtained from the method of lemma 2.5, by replacing all calls to the point-primitives by calls to the primitive hull operations of Atallah and Goodrich, will run in $O(1)$ time with the same probability as given in section 2.1, namely $1 - f$ where $f = (2^{-\Omega(n^e)})$, but now requires $mq^{\frac{1}{b}}$ processors, where b, e are constants.

2.5 An Almost-Optimal Algorithm

Here, we show how to find the convex hull of a set of pre-sorted points in the plane in $O(\log^* n)$ time with

n processors on a CRCW PRAM. (In the next subsection, we show how to reduce the number of processors to $\frac{n}{\log^* n}$, making the algorithm optimal.)

Our method uses the constant-time point-hull invariant algorithm of Lemma 2.6 as a subroutine.

Our algorithm is as follows:

1. We split the input of n points into $\frac{n}{\lceil \log^b n \rceil}$ contiguous groups of size $\lceil \log^b n \rceil$ points each, where b is a constant that will be set in the analysis. Given that t is the expected amount of time required to solve these subproblems recursively, we spend time ct, where c is a constant, to solve the problem recursively for each of the subproblems in parallel. Any subproblem that has not been solved by this time, we label a *failure.*

2. We then perform failure sweeping on the subproblems. As with the constant time algorithm, we use Ragde's approximate compaction algorithm to sweep the *failures* into a space of size $n^{\frac{1}{4}}$, then solve them using the brute force technique of Observation 2.3.

3. We assign $O(\lceil \log n \rceil)$ processors to each group of size $\lceil \log^b n \rceil$, and use the constant-time algorithm on hulls of Lemma 2.6, to find the upper hull with these upper hulls as input (acting like points).

It should be clear that the above algorithm uses only $O(n)$ processors. We also have the following:

Lemma 2.7 (Confidence Bounds) *The above algorithm runs in time $O(\log^* n)$, and the probability of failure is $p(m) \leq 2^{-m^{\frac{1}{4}}}$, for a problem of size m.*

Proof: Omitted in this preliminary version. ∎

2.6 An Optimal Algorithm

Here, we show how to reduce the number of processors required in the "almost-optimal" algorithm above, by the use of two-level arrays, and halting the recursion early. This allows our algorithm to run optimally in $O(\log^* n)$ time with $\frac{n}{\log^* n}$ processors. We give the details in the full version of the paper.

This gives us the following theorem:

Theorem 2 *One can find the convex hull of n presorted points in the plane in time $O(\log^* n)$ time with probability $\geq 1 - 2^{-n^{\frac{1}{b}}}$ (where $b > 16$ is a constant), using $\frac{n}{\log^* n}$ processors, on a randomized CRCW PRAM.*

In the next section we present the techniques that we will eventually use in our convex hull algorithms for unsorted inputs.

3 In-Place Techniques

Four basic techniques used in our algorithms for the general (unsorted input) case are the *random vote*, the *random sample*, *in-place approximate compaction* and *in-place bridge-finding.* These are all in-place techniques: they do not require any reordering of the data, and none of them require that the points be at contiguous array locations. They achieve this in-place property through the use of $o(n)$ work space.

3.1 Random Samples and Random Votes

Given m points, each with an associated processor, the random sample procedure is to choose a sample of size $\Theta(k)$, where k is $o(m)$. This sample is to be uniformly random (that is to say, every point must have an equal probability of being included in the sample) and is to be stored in a work space of size $16k$. The random vote procedure is to pick one of the points at random, such that every point has an equal probability of being chosen.

The random sample procedure is as follows:

1. Each processor decides whether it will attempt a write, with probability $2k/m$.

2. Each processor that has decided to write chooses a random location in the work space, and attempts to write its **id** to that location if it is unoccupied.

3. Every processor that performed a successful write then checks whether any other processors attempted to write to this location. This can be done by having the unsuccessful processors re-attempt their write.

4. Each processor that wrote to a location that did not suffer a collision, writes the coordinates of its point into that location. Each processor that did suffer a collision repeats steps 2-4 for a total of up to d attempts, where d is a constant.

Note that all the above steps can be performed in constant time with m processors, on the CRCW PRAM.

By the Chernoff bound [11] (lemma 2.3), fewer than $4k$ processors will attempt a write, with probability

$$\geq 1 - \left(\frac{4}{e}\right)^{2k},$$

and more than k processors, with probability

$$\geq 1 - \left(\frac{e}{2}\right)^{-k}.$$

Thus, given that m' processors attempt to write, with $k \leq m' \leq 4k$, the probability of such a processor suffering a collision is $\leq \frac{1}{4}$. So the number of processors that do suffer a collision is $\leq \frac{k}{2}$ with probability $\leq \left(\frac{e}{2}\right)^{-k}$ (from the Chernoff bound).

Thus, m'', the number of processors that write and do not suffer a collision, is $\geq \frac{k}{2}$, with probability $\geq 1 - \left(\frac{e}{2}\right)^{-k}$. This gives us the following lemma:

Lemma 3.1 *An in-place random sample of size $\Theta(k)$, from an array of size n, can be found in constant time, with n processors on a randomized CRCW PRAM, using work space of size $\Theta(k)$. It is uniformly random with probability $\geq 1 - 2\left(\frac{e}{2}\right)^{-k}$.*

In order to perform a random vote, we take a random sample, and then pick any one element in it, using any method that does not favor some points over others. For example, as the location written to is uniformly random, the first location in the work space that has been written to could have been written to by any point with equal probability, and can be found in constant time (see observation 2.1). This gives us the following:

Corollary 3.1 *An in-place random vote, choosing one out of n elements in an array, can be performed in constant time, with n processors on a randomized CRCW PRAM. It uses $\Theta(k)$ work space where it is uniformly random with probability $\geq 1 - 2\left(\frac{e}{2}\right)^{-k}$.*

3.2 In-Place Approximate Compaction

Lemma 2.1 describes the result of an elegant approximate compaction technique due to Ragde [28]. Unfortunately, this technique is not in itself in-place, so here we present an in-place approximate compaction technique that uses Ragde's method as a subroutine. We achieve the following result:

Lemma 3.2 *Given an array of size m, containing at most k non-zero elements, one can determine whether $k < m^\epsilon$, and if so, one can perform an in-place approximate compaction of these elements into an area of size k^4, all deterministically, using $\max\{k, m^{4\epsilon+\delta}\}$ processors on a CRCW PRAM, with workspace of size $m^{4\epsilon+\delta}$, where $\delta < 1$ and $\epsilon < \frac{1-\delta}{4}$ are constants.*

Proof We split the array into $m^{4\epsilon+\delta}$ groups of size $m^{1-(4\epsilon+\delta)}$, and for every non-zero element in group i, we write a 1 into the ith location of an array of bits, of size $m^{4\epsilon+\delta}$. Note that there can be at most $\min\{m^{4\epsilon+\delta}, k\}$ non-zero bits in this array. We then perform approximate compaction (see Lemma 2.1) on this second array, to compress the bits into an area of size $m^{4\epsilon}$. If $k > m^\epsilon$, then this will be detected (see Lemma 2.1). We then split each original group into m^δ groups of size $m^{1-4\epsilon-2\delta}$, and repeat the procedure, ignoring all of the original groups that were found to contain no non-zero elements.

We can iterate this process at most $\frac{1}{\delta}$ times, after which we will have compressed the non-zero elements into an area of size $m^{4\epsilon}$, or determined that $k \geq m^\epsilon$. ∎

3.3 In-Place Bridge Finding

Recall the bridge-finding problem: Given m unsorted points in d dimensions, find the *bridge*, the convex hull facet that intersects a vertical line passing through one specified point (which we call the *splitter*). We address a slightly more general version of the problem, namely, that of finding the bridge for each of q point sets (each with its own splitter), in an array of n points, such that the points corresponding to any one point-set cannot be assumed to be contiguous.

Alon and Megiddo's algorithm [2] solves this problem, but it is not an in-place method. In our algorithm for the unsorted input problem we must deal with many unrelated problems scattered through the input such that the processors of any one problem cannot be assumed to be contiguous, while Alon and Megiddo's algorithm assumes contiguous input for one problem. In this section we show how to achieve almost surely constant time m-constraint linear programming using m processors in-place. Our technique takes a similar approach to that of Alon and Megiddo's algorithm, but has the advantage of being simpler to implement, although it achieves the same time, work and confidence bounds.

We solve the bridge-finding problem by considering the associated linear programming problem (see Observation 2.4). This is solved by repeatedly picking and (deterministically) solving a *base problem*, until the solution has been found. The base problem consists of $\Theta(k)$ constraints, where k is sufficiently small that there are enough processors available to solve the base problem by brute force (see observation 2.2). The work space used for the base problem is $16k$.

In more detail, the 2d problem is solved as below, with $p \geq m$ processors and $k = p^{1/3}$ (the 3d case differs only in that the size of the base problem, k, is $p^{1/4}$, not $p^{1/3}$):

1. Apply the random sample procedure to find a base

problem of size $\Theta(k)$.

2. Solve the base problem deterministically in constant time.

3. After solving the base problem, check for every point whether it violates the solution just found. All constraints that violate the solution in this manner are candidates to be in the next base problem, and are said to be *survivors.*

   At iteration j, having solved $j-1$ base problems, each survivor decides whether to attempt to write into the base problem with probability $p_j = \min\{1, 2kp_{j-1}\}$ in the 2d case, and $p_j = \min\{1, 2kp_{j-1}\}$ in the 3d case. Then the base problem is chosen as above. (From above, $p_1 = \frac{2k}{p}$, and $p_j = 1$ for $j > 4$.)

4. Repeat steps 1-3 for a constant β number of iterations (where β will be set in the analysis). Then perform in-place approximate compaction (see Lemma 3.2) on the survivors, compressing them all into the base problem. (If there are too many to be so compressed, then repeat steps 1-3 once more.) The solution to this problem will be the final solution. Note that if at any earlier point there are no survivors, then *the solution to the last base problem is the convex hull facet sought.*

This completes our procedure. Let us, therefore, analyze this procedure.

3.4 Analysis of the In-Place Bridge-Finding Algorithm

Lemma 4.1 *With probability* $\geq 1 - e^{-\Omega(k^r)}$, *where* $1 > r > 0$ *is a constant, the number of survivors will be reduced to* $\leq k^{\frac{1}{5}}$ *within a constant number of iterations.*
Proof The proof follows the same lines as the analysis in Alon and Megiddo [2], and will be given in the full version of the paper.
Lemma 4.2 *With probability* $\geq 1 - e^{-\Omega(k^r)}$, *where* $1 > r > 0$ *is a constant, the convex hull facet (edge) through the splitting point can be found in a constant number of iterations, of solving base problems.*
Proof From Lemma 4.1, after a constant, β, number of iterations, the number of survivors will be less than $k^{1/5}$, almost surely (with probability $\geq 1 - e^{-\Omega(k^r)}$). Once this is the case, we perform in-place approximate compaction, which then compresses the survivors into an area of size k (see Lemma 3.2), which is the space for the base problem. We then solve the base problem, and as it contained all the survivors, the solution to the base problem will be the solution to the entire problem. This can only fail if the number of survivors $> k^{\frac{1}{4}}$, which from Lemma 4.1, is true with probability $e^{-\Omega(k^r)}$. ∎

4 Convex Hull Algorithms for Unsorted Input

4.1 The 2d Algorithm

Suppose we are given an array of n points in the plane, in no particular order. We wish to find the upper convex hull of these points such that each point has a pointer to the edge above it. (Thus, there may be many points referring to any one edge.) We output the hull edges in a binary tree, built "on top" of the unsorted input points.

The algorithm is similar in structure to randomized quicksort, picking a point at random, uniformly, from the input, then splitting the input about that point and recursing. However, there is no compaction performed, and the convex hull facet above the splitting point is found before recursion. In this sense, the algorithm uses the "marriage-before-conquest" paradigm of Kirkpatrick and Seidel [21].

Initially, every point has a processor assigned to it, and every processor is *active*, i.e., it is assigned operations. If a point is found to be below a convex hull edge, then its associated processor ceases to be active: it ceases to perform any operations associated with the point, and is regarded as *dead.*

The algorithm consists of a number of phases, each of which consists of $\frac{\log n}{32}$ iterations. At the end of each phase, we perform compaction of the subproblems remaining, then reassign the workspace, and reset a counter for the level of recursion.

In phase q, at the i^{th} level of recursion, there are $\leq n^{\frac{1}{32}} 2^i$ subproblems, with average size $s_i = n/(n^{\frac{1}{32}} 2^i)$. (See step 3 below.) The size of the base problems solved is $16k$ where $k = s_i^{1/3}$. For each problem j, $1 \leq j \leq n^{\frac{1}{32}} 2^i$, we do the following:

1. Apply the random vote procedure to choose a splitting point p (see corollary 3.1). We then attempt to find the convex hull edge above p, by applying the in-place bridge-finding procedure (see lemma 4.2). See figure 2. If the bridge-finding procedure for problem j has not naturally terminated after α

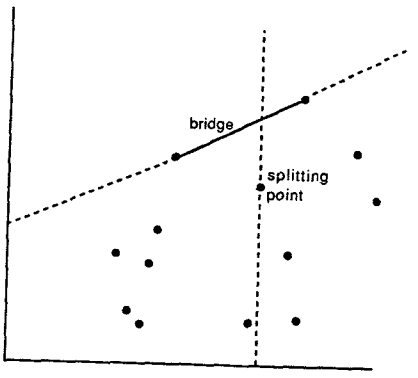

Figure 2: 2D convex hull by bridge-finding

steps (α will be set in the analysis), then we terminate the procedure and say that subproblem j has *failed.*

2. In this step, we use failure sweeping to compact those problems that failed in the previous step, so that each problem can be assigned $n^{\frac{3}{4}}$ processors. We assign $k = n^{\frac{1}{4}}$ space, and $p = n^{\frac{3}{4}}$ processors, to each problem, and use the in-place bridge-finding procedure to find the bridge.

3. If $i \geq \frac{\log n}{32}$, then the algorithm has already taken $O(\log n)$ time, so use parallel prefix sum to compact the remaining points and find the number of subproblems remaining, m. Add m to the number of hull edges found thus far to get l, which is a lower bound on h. If $l \geq n^{\frac{1}{32}}$ (on the first such check it won't be), then solve the problem using any $O(\log n)$ time, n processor algorithm, e.g. the algorithm of Atallah and Goodrich [6]. Otherwise, we reassign the work space among the remaining problems (so each problem recieves $\frac{n}{l} \geq n^{\frac{31}{32}}$), and continue, each subproblem resetting i to 1, incrementing q, and resetting its problem number j to its rank in the compacted list of subproblems.

4. Change the problem number of each point that is active such that a problem number j is changed to $2j-1$, if it is to the left, and to $2j$, if it's to the right, of the solution to problem j. If a point is under the solution edge (a hull edge), then it is given a pointer to that edge, after which it does nothing and it is regarded as a *dead* point. Once this has been done for all points, we halve the work space for each of the problems that has just been solved, and assign the halves to its two "child' problems. Then, we recurse on the subproblems.

The algorithm then continues at recursion level $i+1$, with the above actions, until all points are dead (this can be tested by a concurrent write of all active processors to one location), or until we switch to using the method of Atallah and Goodrich [6].

This completes our description. Let us, therefore, analyze our method.

4.2 Analysis of the 2d algorithm

We first observe that the total number of subproblems active at any step in our algorithm is at most $l2^i \leq n^{\frac{1}{16}}$. In addition, we have the following:

Lemma 5.1 *At the i^{th} level of recursion, with probability $\geq 1 - 2^{-2i}$, each subproblem is of size $< (15/16)^i n$.*

Proof: The likelihood that at least 15/16 of the points are on the same side of the splitting point is 2/16. Thus, after i such splits, the probability that subproblem j is of size $\geq (15/16)^i n$ is $\leq 2^{-3i}$. So the probability p that not all subproblems are of size less than $(15/16)^i n$) is

$$p = 2^i/8^i = 2^{-2i}.$$

■

This implies that the parallel time for our method is $O(\log n)$, with very high probability ($\geq 1 - n^b$, where b is a constant). Let us, then, analyze our space requirements. It is easy to see that the work space available for any active subproblem is $\geq \frac{n^{\frac{31}{32}}}{2^i} \geq n^{\frac{15}{16}}$.

Therefore, the total space needed is $O(n)$.

Lemma 5.2 *At each level of recursion, with probability $\geq 1 - e^{-\Omega(n^f)}$, where $1 > f > 0$ is a constant, every bridge is found in constant time.*

Proof: Lemma 4.2 states that after a constant number of steps, which we call α, the in-place bridge-finding procedure will have found the bridge with probability $\geq 1 - e^{-\Omega(k^r)}$, and in this case, $k = \left(\frac{n}{2^i}\right)^{\frac{1}{3}}$. So, after α steps, we terminate any failures, and use Ragde's approximate compaction algorithm. The total number of problems at each recursion level is always $< n^{\frac{1}{16}}$, so Ragde's approximate compaction algorithm [28] can be used to compress the problem ids into an area of size $n^{\frac{1}{4}}$, giving $p = n^{\frac{3}{4}}$ processors for each problem. Given in-place bridge-building with a sample size of at most $k = n^{\frac{1}{4}}$, and $p = n^{\frac{3}{4}}$, lemma 4.2 gives the result. ■

Lemma 5.3 *The work required to find the 2d convex hull is $O(n \log h)$.*

Proof: Given that $W(n,h)$ is the work required to find the convex hull of n points, with h edges, using the

above algorithm, we can show the result in a manner similar to the analysis in Seidel [32]. We give the details in the full version of the paper.

Theorem 5 *With probability $1 - n^{-b}$, where b is a constant, the 2d convex hull can be found in $O(\log n)$ time with $O(n \log h)$ work.*

Proof: This is a consequence of the above lemmas, and Lemma 7. (Lemma 7 is the observation of Matias and Vishkin [24] on processor allocation discussed in section 5.) ∎

4.3 The 3d Algorithm

Suppose we are given an array of n points in $\Re^3$, in no particular order. We wish to find the upper convex hull of these points such that each point knows the face above it. (Thus, there may be many points referring to any one face.)

Our algorithm is similar in structure to quicksort, and to the sequential 3d convex algorithm of Edelsbrunner and Shi [16]. Unlike Edelsbrunner and Shi, however, we do not split the input into subproblems about the "ham-sandwich cut", but about a point chosen at random, uniformly, from the input.

The algorithm consists of a number of phases, each of which consists of $O(\log n)$ iterations. At the end of each phase, we perform compaction of the subproblems remaining, reassign workspace, and reset a counter for the level of recursion. In the qth phase, at the i^{th} level of recursion, there are at most $n^{\frac{1}{32}}4^i$ subproblems, with average size $s_i = n/(n^{\frac{1}{32}}4^i)$ (see step 4 below). The size of the base problems solved is $O(k)$ where $k = s_i^{1/4}$. For each problem j, $1 \leq j \leq n^{\frac{1}{32}}4^i$:

1. We choose a splitting point by the random vote procedure, (see corollary 3.1) and find the convex hull face that it is under, using the 3d in-place bridge-finding technique (see lemma 4.2) (which, in this case, finds a triangular face of the upper hull). Note that if the in-place bridge finding technique for problem j does not naturally terminate after α steps (α will be set in the analysis) , then we terminate the procedure and say that subproblem j has *failed.*

2. In this step we use failure sweeping to compact those problems that have not yet been completed, so that each problem can be assigned $n^{\frac{3}{4}}$ processors. Using $16k = 16n^{\frac{1}{4}}$ as the space for the random sample, we use the in-place bridge-finding procedure to find the bridge.

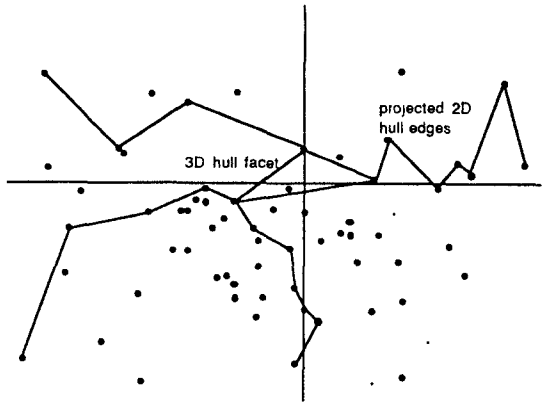

Figure 3: Division of the point set

3. We divide each problem into 4 subproblems as follows:

   We project all the points onto both the xz and yz planes through the splitting point, projecting them along directions parallel to that of the convex hull face just found. Then, we use our 2d algorithm (unsorted case) to find the 2d hulls of the projected points in these planes: the 2d hull edges will also be edges of the 3d hull [16] and due to the nature of our 2d algorithm, each point will store its position relative to these hulls. The hulls necessarily divide each problem into 4 (possibly unequal) parts, defined by the intersection of the 2 hulls. The depth of recursion taken to find the 2d convex hulls is added to the total *depth* taken, i.e., the depth of recursion in the 3d algorithm, plus the depth of recursion in previous calls to the 2d algorithm. If the 2d algorithm must resort to using the method of Atallah and Goodrich [6], then the total work required must be $O(n \log n)$, so use the algorithm of Reif and Sen [30], to find the 3d convex hull in $O(\log n)$ time using n processors with very high probability.

4. If $i \geq \frac{1}{64}\log n$ or the total depth (as defined in step 3) $\geq \log n$, then perform parallel prefix sum, to compact and take and a count of the problems left, and the number of edges found thus far, the sum of these two giving l. If $l \geq n^{\frac{1}{32}}$, then find the convex hull using the algorithm of Reif and Sen [30], in $O(\log n)$ time using n processors with very high probability. Otherwise, redivide the workspace among the remaining problems, reset i to 1, increment q, and continue. See figure 3.

5. For all points, change their problem number from j to $4j-3, 4j-2, 4j-1$ or $4j$, depending on the quadrant of the solution it finds itself in. If a point

is under the solution facet, then it is given a pointer to the solution, which is the hull facet above it. After this, the point does nothing, and is regarded as a *dead* point. Once this has been performed for all points, recurse on the subproblems. The algorithm then continues at recursion level $i+1$, with the above actions, until all points are dead, or the method of Reif and Sen [30] must be used.

4.4 Analysis of the 3d algorithm

We first observe that the total number of subproblems active at any step in our algorithm is $\leq l2^{2i} \leq n^{\frac{1}{16}}$. In addition, we have the following:

Lemma 6.1 *At the i^{th} level of recursion, with probability $\geq 1-2^{-4i}$ each subproblem is of size $< (15/16)^i n$.*

Proof: Consider the quadrants that are defined in the xy-plane, by the xz- and yz-planes through the splitter. Now if one quadrant contains $\geq \frac{15}{16}$ of the points, then both quadrants with a different x-coordinate range must contain at most $\frac{1}{16}$ of the the points, (which occurs with probability $\frac{2}{16}$) and the same must be true of the two quadrants that have a different y-coordinate range. So the probabiity of this happening for any one splitter is $\frac{4}{256} = \frac{1}{64}$.

Thus, after i such splits, the probability that subproblem j is of size $\geq (15/16)^i n$ is $\leq 2^{-6i}$. So the probability p that all subproblems are of size less than $(15/16)^i n)$ is $\geq 1-2^{-4i}$. ∎

This implies that the depth of recursion for our method is $O(\log n)$, and thus our method takes $O(\log^2 n)$ parallel time (by Theorem 5), Let us, then, analyze our space requirements. It is easy to see that the work space available for any active subproblem is $O(n^{\frac{15}{16}})$. Therefore, the total space needed is $O(n)$.

In the full version of the paper, we prove a series of lemmas analogous to lemmas 5.2-5.3, which lead to the following theorem

Theorem 6 With probability $1-n^{-b}$, where b is a constant, the 3d convex hull can be found in $O(\log^2 n)$ time with $O(min\{n\log^2 h, n\log n\})$ work, on a randomized CRCW PRAM.

Proof: The proof is given in the full version of the paper. ∎

5 Processor Allocation

As described above, our algorithms assume that some number of processors (n, if the input is unsorted) are available. If the number of processors p is less than this, then the work must be divided such that each processor will be assigned tasks so as to preserve the total amount of work w, and achieve time $O(\frac{w}{p}+t)$ or as close to these bounds as possible. Here, t is the time taken by the algorithm if the number of real processors = the number of virtual processors. If $p \leq n/\log n$, then the simplistic method used for the algorithms for presorted input will suffice. Otherwise, we need to use an observation of Matias and Vishkin [24].

Lemma 7 Given an algorithm with a work bound w, and time bound t, that requires at least n processors, assuming that w is known and that p processors are available, the algorithm can be simulated with p processors in time $T = t + \frac{w}{p} + t_c \log t$ with work $W = pt + w + pt_c \log t$.

Proof: given by Matias and Vishkin [24].

6 Conclusion

We have shown how to use randomization, generalized linear programming, and a number of in-place techniques to achieve fast work bounds that are very efficient for 2 and 3d convex hulls. It would be interesting to see how these results generalize to higher dimensions.

7 Acknowledgements

We would like to thank Omer Berkman for discussions that led to the developement of the $O(\log^* n)$ time algorithm for pre-sorted input 2d convex hull computation.

References

[1] Aggarwal, A., Chazelle, B., Guibas, L., Ó'Dunlaing,C., and Yap, C., "Parallel Computational Geometry," *Proc. 26th IEEE FOCS* (1985), pp.468-477.

[2] N. Alon and N. Megiddo, "Parallel Linear Programming in Fixed Dimension Almost Surely in Constant Time", *Proc. 31st IEEE FOCS Symposium* (1990), pp. 574-582.

[3] Andrew, A.M., "Another efficient algorithm for convex hulls in two dimensions," *Info. Proc. Lett.* **9**, (1979) pp.216-219.

[4] Avis, D., "On the complexity of finding the convex hull of a set of points," Report SOCS 79.2, (1979), School of Computer Science, McGill University,

[5] Atallah, M. J., and Goodrich, M. T., "Efficient Parallel Solutions to Some Geometric Problems," *Journal of Parallel and Distributed Computing*, **3** (1986), pp. 492-507.

[6] Atallah, M. J., and Goodrich, M. T., "Parallel Algorithms for Some Functions of Two Convex Polygons", *Algorithmica* **3** (1988), pp. 535-548.

[7] Bentley, J. L., Faust, G. M., and Preparata, F. P., "Approximation algorithms for convex hulls," *Comm. ACM* **25**, (1982) pp.64-68.

[8] Berkman, O., Breslauer, D., Galil, Z., Schieber, B., and Vishkin, U., "Highly Parallelizable Problems," *Proc. 21st ACM STOC* (1989), pp.309-319.

[9] O. Berkman and U. Vishkin, "Recursive Star-Tree Parallel Data-Structure", UMIACS-TR-90-40 CS-TR-2437, University of Maryland Institute of Advanced Computer Studies (1990).

[10] Chand, D.R., and Kapur, S.S., "An algorithm for convex polytopes," *JACM* **17**(1), (1970), pp.78-86.

[11] Chernoff, H., "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of the observations," *Annals of Math. Stat.*, **23** (1952), pp. 493-509.

[12] Chow, A., "Parallel Algorithms for Geometric Problems," PhD dissertation, Computer Science Department, University of Illinois at Urbana-Champagne, 1980.

[13] Cole, R., and Vishkin, U., "Approximate and exact parallel scheduling with applications to list, tree and graph problems," *Proc. 27th IEEE FOCS*, (1986) pp.478-491. pp.128-142.

[14] Cole, R., Zajicek, O., "An Optimal Parallel Algorithm for Building a Data Structure for Planar Point Location", *J. Parallel and Distributed Computing* **8** (1990) pp. 280-285.

[15] Dadoun, N., and Kirkpatrick, D.G., "Parallel Construction of Subdivision Hierarchies," Tech. Report 87-13 (1987), Department of Computer Science, University of British Columbia.

[16] H. Edelsbrunner and W. Shi, "An $O(n \log^2 h)$ Time Algorithm for the Three-dimensional Convex Hull Problem", *SIAM J. on Computing* **20** (1991), pp. 259-269.

[17] Eppstein, D., and Galil, Z., "Parallel Algorithmic Techniques for Combinatorial Computation", *Ann. Rev. Comput. Sci.*, 3, pp. 233-283.

[18] Graham, R.L., "An efficient algorithm for determining the convex hull of a planar set," *Info. Proc. Lett.* **1**, (1972) pp.132-133.

[19] Jarvis, R.A., "On the identification of the convex hull of a finite set of points in the plane," *Info. Proc. Lett.* **2**, (1973) pp. 18-21.

[20] Karloff, H. J., and Raghavan, P., "Randomized Algorithms and Pseudorandom Numbers", *Proc. 20th ACM STOC* (1988), pp. 310-321.

[21] Kirkpatrick, D., G., and Seidel, R., " The ultimate planar convex hull algorithm?", *SIAM J. Comput.* **15** (1), (1986), pp. 287-299.

[22] Ladner, R. E., Fischer, M. J., "Parallel prefix computation," *JACM* **27** (1980), pp. 831-838.

[23] Mathews, "Number Theory", Chelsea Publications, New York, (1961).

[24] Matias, Y., and Vishkin, U., "Converting High Probability into Nearly-Constant Time - with Applications to Parallel Hashing", to appear in *Proc. 23rd ACM STOC* (1991).

[25] Miller, R., and Stout, Q., F., "Parallel Algorithms for convex hulls," *Proc. Comp. Vision and Pat. Recogn.* (1988).

[26] Overmars, M.H., van Leeuwen, J., "Maintenance of configurations in the plane," *J. Comput. and Syst. Sci.* **23** (1981), pp.166-204.

[27] Preparata, F. P., and Shamos, M. I., "Computational Geometry *An Introduction*", published by Springer-Verlag (1985).

[28] Ragde, P., "The Parallel Simplicity of Compaction and Chaining," *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag Lecture Notes in Computer Science: 443, 1990, 744–751.

[29] Raghavan, P., "Lecture Notes on Randomized Algorithms", unpublished manuscript.

[30] Reif, J. H., and Sen, S., "Polling: A New Randomized Sampling Technique For Computational Geometry", *Proc. 21st ACM STOC* (1989) pp. 394-404.

[31] Seidel, R., "A convex hull algorithm optimal for points in even dimensions," M. S. Thesis, Tech. Rep. 81-14, (1981) Dept. of Comput. Sci., Univ. of British Columbia, Vancouver, Canada.

[32] R. Seidel, "Output-Size Sensitive Algorithms for Constructive Problems in Computational Geometry", (PhD Thesis) TR 86-784, Department of Computer Science, Cornell University, (1986), pp 15-16.

[33] Stout, Q. F., "Constant-Time Geometry on PRAMs", *Proc. of the 17th International Conference on Parallel Processing* 1988, pp 104-107.

[34] Yao, A.C., "A lower bound to finding convex hulls," *J. ACM* **28** (1981) pp.780-787.