ELSEVIER

Computational Geometry 7 (1997) 219–235

Computational Geometry
Theory and Applications

Fast randomized parallel methods for planar convex hull construction ☆

Mujtaba R. Ghouse [a,1], Michael T. Goodrich [b,*,2]

[a] *Quintus Corporation, 301 E. Evelyn Ave., Mountain View, CA 94041, USA*

[b] *Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218-2694, USA*

Communicated by C.K. Yap; submitted 1 October 1994; accepted 8 September 1995

Abstract

We present a number of efficient parallel algorithms for constructing 2-dimensional convex hulls on a randomized CRCW PRAM. Specifically, we show how to build the convex hull of n presorted points in the plane in $O(1)$ time using $O(n \log n)$ work, with n-exponential probability, or, alternately, in $O(\log^* n)$ time using $O(n)$ work, with n-exponential probability. We also show how to find the convex hull of n unsorted planar points in $O(\log n)$ time using $O(n \log h)$ work, with n-exponential 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 algorithm for unsorted inputs depends 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 reordering the input. In order to achieve our n-exponential confidence bounds we use a new parallel technique called *failure sweeping*.

Keywords: Parallel algorithms; Convex hulls; Randomization; Computational geometry; CRCW PRAM

1. Introduction

The problem of constructing the convex hull of a set of n points in the plane is perhaps the most studied problem in computational geometry. The problem is generally defined as that of constructing the boundary of the smallest convex set containing all n points, and it can be solved sequentially in $O(n \log n)$ time. (See [12,25,26] for references.) This bound is, in fact, optimal, for Yao [29] shows that identifying the points on the boundary of a 2-dimensional convex hull has an $\Omega(n \log n)$ lower

☆ This research was announced in preliminary form in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, 1991, 192–203.

* Corresponding author. E-mail: goodrich@cs.jhu.edu.

[1] This research supported in part by NSF and DARPA under Grant CCR-8908092. E-mail: mujtaba.ghouse@quintus.com.

[2] This research supported in part by the National Science Foundation under Grants CCR-9003299 and CCR-9300079, by NSF and DARPA under Grant CCR-8908092, and by ARO under Grant DAAH04-96-1-0013.

SSDI 0925-7721(95)00036-4

bound in the algebraic computation tree model. Nevertheless, Kirkpatrick and Seidel [22] 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 (which can be as small as 3 in some cases). Such an algorithm is said to be *output-size sensitive*. Strictly speaking, their method does not contradict the lower bound arguments, however, as these arguments assume that the output size is linear. In addition, their method assumes that the input points are unsorted, whereas one can beat their bound for sorted [18,25,26] or even partially-sorted [16] inputs. The running time in the (fully) sorted case is $O(n)$, for example.

Two-dimensional convex hull construction has also been well-studied in parallel. Chow [9,10] is the first to study this problem, achieving $O(\log^2 n)$ time with n processors in the CREW PRAM model, the synchronous parallel shared-memory model that allows for simultaneous concurrent-reads but requires that all writes be exclusive. The *work* performed by this algorithm, i.e., the total number of computations, is suboptimal, however, as it is $O(n \log^2 n)$. The running time using n processors was subsequently (and independently) improved to $O(\log n)$ time by Aggarwal et al. [1] and Atallah and Goodrich [3], thus achieving optimal-work parallel methods. Miller and Stout [24] show how to achieve the same result on an EREW PRAM, the parallel model where both reads and writes must be exclusive. If the input is presorted, Goodrich [15] shows how to achieve $O(\log n)$ time with an optimal $O(n/\log n)$ number of processors on a CREW PRAM, and Chen [7] shows how to achieve these same bounds in the EREW PRAM model. Recently, Berkman et al. [5] have shown how to achieve $O(\log \log n)$ time with $O(n/\log \log n)$ processors in the CRCW PRAM model, the synchronous parallel model where simultaneous concurrent reads and writes are allowed, with concurrent writes being resolved so that one succeeds arbitrarily [3].

None of these previous algorithms achieve the sequential output-sensitive work bounds described above, however. Nevertheless, using standard parallel techniques, one can use a parallel linear programming algorithm of Deng [11] to implement a parallel version of the algorithm of Kirkpatrick and Seidel [22] to run in $O(\log^2 n)$ time, using $O(n \log h)$ work. It is not clear how one could improve this time bound without the introduction of new techniques, however. Nor is it clear from this work whether randomization helps in the parallel construction of planar convex hulls.

In this paper we present efficient parallel algorithms for convex hull construction in the randomized CRCW PRAM model. We address both the presorted and unsorted cases. For the presorted case we give an algorithm that runs in $O(1)$ time using $O(n \log n)$ processors on a randomized CRCW PRAM, with n-exponential probability [4]. 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 [5]. We call such an algorithm *point-hull invariant*, and show how it can be used to derive a solution to the convex hull problem running in $O(\log^* n)$ time with an optimal number of processors, with n-exponential probability. For the unsorted case we give an efficient output-sensitive parallel method that runs in $O(\log n)$ time using $O(n \log h)$ work with n-exponential probability. Our analysis does not depend on any assumptions about the input distribution.

[3] There are other possible conflict resolution methods, as well (e.g., see [13]), but we will be assuming this "arbitrary" resolution rule throughout this paper.

[4] We say an event occurs with *n-exponential probability* if the probability that this will not occur is at most c^{-n^d}, where $c > 1$, $d > 0$ are constants.

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

Our methods are also based on various adaptations of a method of Alon and Megiddo [2] for performing linear programming in fixed dimensions in O(1) time using O(n) processors, with n-exponential probability, in a randomized CRCW PRAM model. The general approach of our methods is based on that of Kirkpatrick and Seidel [22]: namely, we use linear programming to "probe" the convex hull, either in parallel at many locations or in just a few places so as to 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, since 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, the main idea of this technique is to "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. Incidentally, subsequent to the initial announcement of our results in [14] we have learned that this technique has been independently discovered by Matias and Vishkin [23], who call it the *thinning-out principle*.

Our method for the unsorted case also depends 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 reordering 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.

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 in Section 3.

2. Convex hull algorithms for presorted input

Given n presorted points in the plane (assume, without loss of generality, 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 that we use in our algorithm.

Lemma 2.1 (The approximate compaction lemma (Ragde [27])). *Given an array of size n containing at most k nonzero elements, then there is a method that will either conclude that $k > n^{1/4}$ or compress these k elements into an area of size k^4, in* O(1) *time on a* CRCW PRAM (*deterministically*) *with n processors.*

The next lemma deals with the linear programming problem, where one is given m constraints (half-spaces) in $\Re^d$, and a linear (objective) function, and one desires the point in $\Re^d$ where the objective function is maximized subject to the constraints.

Lemma 2.2 (Alon and Megiddo [2]). *Given n half-space constraints in $\Re^d$, for fixed d, linear programming can be performed in* $O(1)$ *time with n processors on a* CRCW PRAM, *with (n-exponential) probability $1 - 2^{cn^{1/3}}$, where $c > 1$ is a constant.*

Lemma 2.3 (Tail estimation (Chernoff bound [8,20])). *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 $X = \sum_{i=1}^{n} X_i$, $\mu = \sum_{i=1}^{n} p_i$. Then, for all $\delta > 0$,*

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

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

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

We also make use of the following observations.

Observation 2.4. *The minimum (maximum) of n elements can be found by n^2 processors in* $O(1)$ *time, on a* CRCW PRAM.

Lemma 2.5 (Eppstein and Galil [13]). *The first nonzero element of an array of size n can be found in* $O(1)$ *time on a* CRCW PRAM *with n processors.*

Observation 2.6 (Brute force linear programming). *It is possible to solve linear programming in d dimensions in* $O(1)$ *time with n^{d+1} processors, for fixed d.*

Proof. 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. □

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

Lemma 2.7. *For any integer $k \geqslant 1$, one can find the upper hull of n presorted points in the plane in time* $O(k)$, *using $n^{1+2/k}$ processors, deterministically, on a* CRCW PRAM.

Proof. This result is part of the "folklore" of parallel computational geometry. We include a proof here for completeness sake. The proof is by the following inductive algorithm A_k (for which A_1 is the base algorithm), each step of which takes constant time:

1. Split the input into $n^{1/k}$ contiguous groups of size $n^{1-(1/k)}$ each, and apply algorithm A_{k-1} to each such group if $k > 1$ (if $k = 1$, then each group contains just one point). If $k = 1$, then this step (trivially) uses n processors, and if $k > 1$, then, by induction, this step requires $n^{1/k}(n^{1-(1/k)})^{1+2/(k-1)}$ processors, which is $n^{1+(2/k)}$.

2. For each pair of groups, find the common tangent using a method of Atallah and Goodrich [4] that computes the common tangent of two upper hulls of n points each in $O(1)$ time using $n^{1/c}$ processors ($c = 1$ will be sufficient here). This step can be implemented with $n^{2/k}n^{1-(1/k)} = n^{1+(1/k)}$ processors.
3. For each group, find the highest of the common tangents to this group that is to the left of the group, and the highest common tangent that is to the right of the group. If the two of these tangents cross above the hull for the group, then the group is said to be *closed*, otherwise it is said to be *open*. Note that the computation for each open group is now done, and that this step can be implemented using $n^{1/k}n^{2/k} = n^{3/k}$ processors, which is, of course, at most $n^{1+(2/k)}$.
4. For each closed group, find the (unique) upper hull edge that is above it, that is between two hulls that are open. This can be done in $O(1)$ time using $n^{2/k}$ processors.

Thus, one can find the upper hull in $O(k)$ time using $n^{1+(2/k)}$ processors, as claimed. □

2.2. *A more efficient algorithm*

Having reviewed the above results, we now present an $O(n \log n)$-processor algorithm that runs in $O(1)$ time with n-exponential probability. So suppose we are given a set S of n input points in the plane, such that the points in S are sorted by increasing order of x-coordinate. We consider a complete binary tree T built "on top" of these points, ordered left-to-right. For each node v in T let S_v denote the subset of S contained in the subtree of T rooted at v. For any node v in T define the *bridge* for v as the edge of the upper hull of S_v intersected by a vertical line separating the points in the subtrees rooted at v's two children. Kirkpatrick and Seidel [22] design a beautiful sequential algorithm upon the observation that, by a duality between points and lines (e.g., see [12]), the problem of finding a bridge for an n-point set is dual to an n-constraint 2-dimensional linear program. Our methods will also be based upon this observation.

Given $n \log n$ processors in a randomized CRCW PRAM, we can, for each v in T, simultaneously solve the associated linear programming problem using Alon and Megiddo's algorithm (see Lemma 2.2). Having done this, we can then assign $\log n$ processors to each node v, and check whether any of v's ancestors store a bridge that covers (i.e., is strictly above) the bridge for v, in $O(1)$ time (this amounts to an OR). Finally, assigning $\log n$ processors to each point (leaf) p, we can find the lowest ancestor of p that is not covered by another edge: this will be the edge above p. (See Lemma 2.5.)

Thus, it might seem that, as Alon and Megiddo's algorithm takes $O(1)$ time and has confidence bounds that are exponential in the size of the input, we have achieved our result. This is not quite 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 the subproblem, they are not necessarily 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 2.7 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 $\log^3 n \leqslant m \leqslant n^{1/4}$.

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 $m^{4/3}$

processor, O(1)-time, algorithm (see Lemma 2.7, 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 $m^{4/3}n/m = nm^{1/3}$. As the highest level that uses brute force has $m < \log^3 n$, the total number of processors required is

$$< n \log n \left(1 + \frac{1}{2^{1/3}} + \frac{1}{4^{1/3}} + \cdots\right),$$

which is $O(n \log n)$. Note that all these small problems are solved deterministically in O(1) time. Each other problem is solved using Alon and Megiddo's algorithm, with a linear number of processors in O(1) time, with probability of failure $\leqslant 2^{-cm^{1/3}}$, where $c > 1$ and m is the size of the problem (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 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 $(n/m)p(m) \leqslant 1$, where a *failure* is a subproblem that has not yet been solved. The failures are then compacted into an area of size $n^{1/r}$, where $r \geqslant 4$, using Ragde's algorithm (see Lemma 2.1). Then, we assign $n^{1-1/r}$ processors to each failure, and use a "brute-force" method to solve them.

For our convex hull problem, we use Alon and Megiddo's algorithm on all problems of size $\geqslant \log^3 n$, and perform failure sweeping on each level of the binary tree that is of height h such that $3 \log\log n \leqslant h \leqslant (1/4) \log n$. From Lemma 2.2, we have $p(n) = 2^{-cn^{1/3}}$, and from Observation 2.6, 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 $n^{1/4}$. Assuming this compaction succeeds, we use the brute force algorithm of Observation 2.6, with $n^{3/4}$ processors assigned to each failure. (This is a sufficient number of processors for all problems of size $m \leqslant n^{1/4}$ as the brute force technique requires m^3 processors.)

After spending some constant α amount of time on Alon and Megiddo's technique at each level, the expected number of failures is

$$\mu = \frac{n}{m} 2^{-cm^{1/3}},$$

but as $m \geqslant \log^3 n$, we have $\mu \leqslant 1/n^{c-1}$, so $\mu \leqslant 1$.

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

$$f \leqslant \left(\frac{e^{n^{1/16}-1}}{(n^{1/16})^{n^{1/16}}}\right), \quad \text{which implies}$$

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

$$f \leqslant \left(\frac{1}{e}\right)2^{-n^{1/16}((1/16)\log n - \log e)}.$$

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

$$1 - f \geqslant 1 - \left(\frac{1}{e}\right)2^{-n^{1/16}}$$

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

This gives us the following lemma.

Lemma 2.8. *The* 2D *convex hull problem can be solved in* $O(1)$ *time with* $O(n \log n)$ *processors, with* n*-exponential probability.*

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

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.

Suppose we are given n halfplane constraints defined by the duals to n points in the plane, and a linear objective function dual to the problem of finding the bridge passing through a given vertical line. The linear programming algorithm of Alon and Megiddo [2], applied to this set, comprises repeatedly choosing a subset of the constraints, and finding the solution to this subset of constraints with respect to the given objective function. 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. To find the solution to the chosen subproblem, one uses brute force (see Observation 2.6), with the subset being chosen so as to be small enough that this can be done in $O(1)$ time with n processors.

Their algorithm requires only a handful of primitive operations on points and lines. We wish to apply their method to objects that are themselves upper hulls, however. Thus, we must expand the set of primitive operations from operating on points and lines to operate on upper hulls. Fortunately, each such operation on points and lines corresponding to an analogous operation on upper hulls. In particular,

- reading the x or y coordinate of a point $\Leftrightarrow$ finding the intersection of a line with an upper hull;
- finding the line defined by two points $\Leftrightarrow$ finding the common tangent of two upper hulls;
- intersecting two lines $\Leftrightarrow$ intersecting two hulls;
- checking if a point is above a line $\Leftrightarrow$ finding the intersection of a line with an upper hull.

We define an algorithm on n points that would require only these operations to function with n upper hulls as input instead as *point-hull invariant*. Atallah and Goodrich [4] show that each of these

upper hull problems can be solved in $O(1)$ time with $O(n^{1/b})$ 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 2.9. *Both the brute force algorithm and Alon and Megiddo's technique (of Observation 2.6 and Lemma 2.2, respectively) are point-hull invariant. In addition, the constant-time upper hull algorithm (of Lemma 2.8) is also point-hull invariant.*

As a result, given m upper hulls, each containing at most q points, we can run the constant-time convex hull algorithm of Lemma 2.8, using a modification of (the dual 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 [4]. This gives us the following lemma.

Lemma 2.10. *Given m upper hulls, each containing at most q points, one can find the upper hull of their union in $O(1)$ time, with n-exponential probability, using $O(mq^{1/b} \log m)$ processors, where b is any fixed constant.*

2.5. A near-optimal algorithm

We now show how to find the convex hull of a set of presorted 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 $n/\log^* n$, making the algorithm optimal.) Our method uses the constant-time point-hull invariant algorithm of Lemma 2.10 as a subroutine.

1. We split the input of n points into $n/\lceil \log^b n \rceil$ 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 $n^{1/4}$, then, if this compression is successful, we solve them using the brute force technique of Lemma 2.7.
3. We complete the upper hull construction by applying Lemma 2.10, with $m = n/\lceil \log^b n \rceil$ and $q = \lceil \log^b \rceil$ (and with b as in the lemma). This takes $O(1)$ time using $O(n/\log^{b-2} n)$ processors, with n-exponential probability.

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

Lemma 2.11. *The above algorithm runs in time $O(\log^* n)$, using $O(n)$ processors, with n-exponential probability.*

Proof. The running time and processor bounds follow immediately by induction, provided the recursive calls succeed with the claimed probability. Let us therefore focus on establishing the probability bound by induction. The basis case is trivial, so assume that the probability claim is true for all prob-

bound by induction. The basis case is trivial, so assume that the probability claim is true for all problems of size $m < n$. Then, we can define b so that the expected number of failures of recursive calls is

$$\mu \leqslant 2^{-(\log^b n)^{1/b}}, \quad \text{implying that}$$
$$\mu \leqslant 1/n.$$

Thus, $\mu \leqslant 1$, and we can apply the Chernoff bound of Lemma 2.3 to bound the probability that the number of failures is greater than $n^{1/16}$ as being at most

$$f_r \leqslant \frac{1}{e} 2^{-n^{1/16}((1/16)\log n - \log e)},$$

so that $f_r \leqslant 2^{-n^{1/b}}$, for $b > 16$, and for all n larger than a constant. As described above, we then use Ragde's algorithm [27] which will succeed in compressing all the failures into an area of size $n^{1/4}$, to be solved by brute force, if and only if the number of failures is $\leqslant n^{1/16}$ (by Lemma 2.1), and, as the rest of the procedure will succeed with probability $1 - f_r$, if the approximate compaction succeeds. □

2.6. *An optimal algorithm*

Here, we show how to reduce the number of processors required in the 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 $O(n/\log^* n)$ processors. We halt the recursion as soon as the current problem size is less than or equal to $\log^* n$ (where, here, n is the original problem size). In this base case, each processor is assigned $O(\log^* n)$ contiguous points, for which it finds the convex hull using a sequential, deterministic algorithm (e.g., see [12,25,26]). We call this hull of $O(\log^* n)$ points a *mini-hull.* Then, another array is set up, this time of size $n/\log^* n$, such that the ith element contains the extremal x coordinates of the ith mini-hull. Any access to these mini-hulls in the recursive case is via this array. This ith element also contains a record of the one or two edges that might cover this mini-hull in the recursive case, i.e., the (at most) two edges of the currently known hull that are not part of the mini-hull, but are directly above it. (It is simple to show that there can be at most two such edges.)

Also, there must be one slight modification to the algorithm: in the final step of the method of Lemma 2.10 (finding the edge on the hull above each point), at each level of recursion, instead of finding the edge on the new hull that covers every point, we use the same technique to find the (at most two) edges that cover each mini-hull of size $O(\log^* n)$. This information is then used to update the array of size $n/\log^* n$. Note that while finding the edges of the upper hull only requires $O(n/\log^{b-2} n)$ processors, this last step will require at most $O(n/\log^* n)$ processors in total. Finally, after all recursion levels have been dealt with, we update the hull edge above every individual point, by checking whether each point is beneath either of the edges that covers its mini-hull. By arguments similar to those given above, then, we have the following theorem.

Theorem 2.12. *One can find the convex hull of n presorted points in the plane in time $O(\log^* n)$ time with n-exponential probability, using $O(n/\log^* n)$ 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 point sets.

3. In-place techniques

In this section we describe several techniques used in our algorithm for the general (unsorted input) case. These are all in-place techniques in that they do not require any reordering of the data, and none of them require that the points be at contiguous array locations. We achieve this in-place property through the use of a small work space of nonconstant size. This is a slight extension of the use of the term "in-place", which is usually applied to sequential algorithms that only use $O(1)$ additional memory.

3.1. Random sampling

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.

Our 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 reattempt 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.
5. 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 these steps can be performed in $O(1)$ time with m processors, on the CRCW PRAM.

By a Chernoff bound [8,20] (Lemma 2.3), fewer than $4k$ processors will attempt a write, with probability

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

and more than k processors, with probability

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

Thus, given that m' processors attempt to write, with $k \leqslant m' \leqslant 4k$, the probability of such a processor suffering a collision is $\leqslant 1/4$. So the number of processors that do suffer a collision is $\leqslant k/2$ with probability $\leqslant (e/2)^{-k}$ (from the Chernoff bound). Thus, m'', the number of processors that write and do not suffer a collision, is $\geqslant k/2$, with probability $\geqslant 1 - (e/2)^{-k}$. This gives us the following lemma.

Lemma 3.1. *A random sample of size $\Theta(k)$, from an array of size n, can be found in-place in* $O(1)$ *time, with k processors on a randomized* CRCW PRAM, *using work space of size $\Theta(k)$. It is uniformly random with k-exponential probability.*

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 $O(1)$ time (see Lemma 2.5). This gives us the following corollary.

Corollary 3.2. *An in-place random vote, choosing one out of n elements in an array, can be performed in* $O(1)$ *time with n processors on a randomized* CRCW PRAM. *It uses $\Theta(k)$ work space and it is uniformly random with k-exponential probability.*

3.2. In-place approximate compaction

Lemma 2.1 describes the result of an elegant approximate compaction technique due to Ragde [27]. 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 3.3. *Given an array of size m, containing at most k nonzero elements, one can determine whether $k > m^{\varepsilon}$, or one can perform an in-place approximate compaction of these elements into an area of size k^4, all deterministically, using $\max\{k, m^{4\varepsilon+\delta}\}$ processors on a* CRCW PRAM, *with workspace of size $m^{4\varepsilon+\delta}$, where $\delta < 1$ and $\varepsilon < (1-\delta)/4$ are constants.*

Proof. We split the array into $m^{4\varepsilon+\delta}$ groups of size $m^{1-(4\varepsilon+\delta)}$, and for every nonzero element in group i, we write a 1 into the ith location of an array of bits, of size $m^{4\varepsilon+\delta}$. Note that there can be at most $\min\{m^{4\varepsilon+\delta}, k\}$ nonzero 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 $m^{4\varepsilon}$. If the mapping fails, then $k > m^{\varepsilon}$. If it does not fail, we then split each original group into m^{δ} groups of size $m^{1-4\varepsilon-2\delta}$, and repeat the procedure, ignoring all of the original groups that were found to contain no nonzero elements. We can iterate this process at most $1/\delta$ times, after which we will have compressed the nonzero elements into an area of size $m^{4\varepsilon}$, or determined that $k \geqslant m^{\varepsilon}$. □

3.3. In-place bridge finding

Recall the bridge-finding problem: we are given a set of unsorted points, and we wish to find the *bridge*, the convex hull edge 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.

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 linear programming algorithm [2] assumes contiguous input

for one problem. In this subsection we show how to solve an m-constraint linear programming using m processors in-place in O(1)-time, with m-exponential probability. 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. 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 2.6). The work space used for the base problem is $16k$. In more detail, the problem is solved as below, with $p \geqslant m$ processors and $k = p^{1/3}$:

1. Apply the random sample Lemma 3.1 to find a base problem of size $\Theta(k)$.
2. Solve the base problem deterministically in O(1) time, which, by Observation 2.6, requires $k^3 = p$ processors.
3. After solving the base problem, check for every point (even if it was not previously a survivor), whether it violates the solution just found. All points 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, $p_1 = (2k)/p$, and $p_j = 1$ for $j > 4$.)
4. Repeat steps 1–3 for β iterations (where β will be set in the analysis). Then perform in-place approximate compaction (see Lemma 3.3) 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. First we review a useful result due to Alon and Megiddo.

Lemma 3.4 (Upper bound on the survivors (Alon and Megiddo [2])). *The number of survivors, s, that violate the solution to the base problem, given that there were m_1 surviving points before the solution, is given by*

$$\text{Prob}\left(s > m_1^{(2/3)+\varepsilon}\right) < e^{-\Omega(n^{\varepsilon})}.$$

This gives us the following lemma.

Lemma 3.5. *With probability $\geqslant 1 - e^{-\Omega(k^r)}$, where $1 > r > 0$ is a constant, the number of survivors will be reduced to $\leqslant k^{1/5}$ within a constant number of iterations.*

Proof. This follows from repeated application of Lemma 3.4. □

Thus, by applying a duality between points and lines, we get the following lemma.

Lemma 3.6. *Let S be a set of m points, scattered throughout some array. Given a point p in S, the edge of the convex hull of S intersected by vertical line through p can be found in a constant number of iterations of solving base problems, in-place using m processors, with k-exponential probability.*

Proof. From Lemma 3.5, after a constant, β, number of iterations, the number of survivors will be less than $k^{1/5}$, with k-exponential probability. Once this is the case, we perform in-place approximate compaction, which then compresses the survivors into an area of size $O(k)$ (see Lemma 3.3), 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 $> k^{1/4}$, which from Lemma 3.5, is true with probability $e^{-\Omega(k^r)}$. □

3.4. In-place approximate median-finding

The final in-place method we develop is for approximate median-finding. Suppose we are given a set $X = \{x_1, x_2, \ldots, x_n\}$ of numbers. The approximate median-finding problem is to locate a number x such that the rank of x in X is between εn and $(1-\varepsilon)n$, for some constant ε. Sen [28] gives a randomized CRCW PRAM method for finding an approximate median in $O(1)$ time using $O(n)$ processors, with $\varepsilon = 1/4$ with n-polynomial probability. As Goodrich et al. [17] observe, Sen's algorithm can be adapted to achieve an n-exponential success probability while remaining an $O(1)$-time computation using $O(n)$ processors.

We can easily convert this to an in-place median-finding method. We do this by applying the random-sample procedure of Lemma 3.1 to find a random sample of size $O(k)$. We can then apply the adaptation of Sen's algorithm to find a number x whose rank in the sample is between $k/4$ and $3k/4$ with k-exponential probability. We can apply Sen's algorithm in this case, as the random sample is stored in a contiguous work space. It is easy to show, by the Chernoff bound of Lemma 2.3, that x has rank between $n/8$ and $7n/8$ in X with k-exponential probability. Note, this is true even if $k > n$, for in this case our method will "oversample" X, still yielding its result with k-exponential probability. This gives us the following lemma.

Lemma 3.7. *Given a set $X = \{x_1, x_2, \ldots, x_n\}$ of numbers, there is an in-place constant-time method for finding a number x whose rank in X is between $n/8$ and $7n/8$ with k-exponential probability, using k processors and a working space of size $O(k)$.*

4. A convex hull algorithm for unsorted input

Having presented our primary subroutines, we now give our method for finding the convex hull of an unsorted set of points. So, suppose we are given an array of n points in the plane, in no particular order. We wish to compute the upper convex hull of these points, ordered from left-to-right.

At a very high level of abstraction, our algorithm is similar in structure to the sequential "marriage-before-conquest" algorithm of Kirkpatrick and Seidel [22], which runs in $O(n \log h)$ time, where h is the number of hull points. Our method runs in $O(\log n)$ time using $O(n \log h)$ work with n-exponential probability.

Initially, every point has a virtual processor assigned to it, and every virtual processor is *active*, i.e., each one has a task to perform. But, if at some time during the computation a point is found to be strictly 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 *deactivated.*

The algorithm consists of two phases. The first phase consists of a series of $O(\log n)$ constant-time in-place divide steps, and the second phase consists of a single compaction and deterministic convex hull computation for all remaining points.

So, suppose we are given an array A containing all the input points (again, in no particular order). Each point in A is assigned to belong to problem j, which initially is "1", and we have an work space W_j associated with problem j that is initially of size n. We perform Phase 1 on A as follows:

1. Let X_j denote all the (possibly noncontiguous) elements in X assigned to problem j. We apply the in-place approximate median procedure of Lemma 3.7 to find a (splitting) point p whose x-coordinate has rank between $n_j/8$ and $7n_j/8$ in the set of x-coordinates in X_j, where $n_j = |X_j|$.
2. We then apply the in-place bridge-finding procedure (see Lemma 3.6) to find the bridge edge e above p.
3. For each point q in X_j, if q is strictly below e, then we deactivate the processor for q. Otherwise, if q has smaller (respectively larger) x-coordinate than p, then we assign q to problem $2j$ (respectively $2j + 1$).
4. We then recurse on problems $2j$ and $2j + 1$, evenly dividing between the two of them the work space W_j to form work spaces W_{2j} and W_{2j+1}. Of course, if either problem has no remaining active points, then we need not recurse on that subproblem.

We repeat the above steps for $i = (3/4)\log_2 n$ recursive levels. Note that at this point each subproblem has a work space of size $n/2^i = O(n^{1/4})$; hence, the computation for each subproblem j succeeds with n-exponential probability. Thus, since there are at most $2^{i+1} = O(n^{3/4})$ subproblems in total, this implies that all the Phase 1 computations complete successfully with n-exponential probability.

We begin Phase 2 by then using a parallel prefix sum computation (e.g., see [21]) to compact together all the remaining (active) points. We then apply an optimal (deterministic) parallel method of, say Atallah and Goodrich [4], to compute the upper convex hull of the remaining points. By then merging this hull with the hull of edges found in Phase 1, we can complete the construction of the upper hull in $O(\log n)$ time using $O(n)$ work (e.g., see [19]).

Lemma 4.1. *The total work required by the above algorithm is $O(n \log h)$, with n-exponential probability.*

Proof. Let $W(n, h)$ denote the work required to find the convex hull of n points. Then, assuming every call returns without failing, the work for Phase 1 satisfies the following recurrence relation [22]:

$$W(n, h) = cn + W(n_1, h_1) + W(n_2, h_2),$$

where c is a constant, $h_1 + h_2 = h$, and $n_1 + n_2 \leqslant n$ (for we only count active points). The base cases occur when $h = 2$ or $n \leqslant c$ (or at the bottom-level when we switch to Phase 2). We claim that $W(n, h) = O(n \log h)$, which is clearly true for the base cases (we will account for the bottom-level work separately).

For the inductive case, we can bound

$$W(n, h) \leqslant cn + dn_1 \log h_1 + dn_2 \log h_2,$$

where d is a constant. Let n_i' denote all the points in subproblem i, include those no longer active, so $n = n_1' + n_2'$. Without loss of generality, we can assume $h_1 \leqslant h_2$; hence,

$$W(n, h) \leqslant cn + dn_1 \log h/2 + dn_2 \log h \leqslant cn + dn_1' \log h/2 + dn_2' \log h.$$

Assuming the approximate median-finding procedure succeeds, we know that $n'_1 \geqslant n/8$. Thus,

$$W(n,h) \leqslant dn \log h + cn - (d/8)n.$$

By taking d large enough, then, this establishes the claim that $W(n,h)$ is $O(n \log h)$ for Phase 1 (assuming all recursive calls return successfully).

Let us therefore consider the extra work we perform in Phase 2. Assuming each call to approximate median-finding returned correctly, the maximum size of a subproblem is $(7/8)^i n = n^{1-(3/4)\log(8/7)}$, which is less than $n^{.86}$. Let us distinguish two possible cases, then, based upon n', the number of elements still active after Phase 1 completes.

1. $n' \geqslant n^{.96}$. In this case, there must be at least $n^{.1}$ subproblems. This, in turn, implies that $h > n^{.1}$, however, for there must be at least one undiscovered hull edge left for each subproblem. Thus, the additional work that is performed is $O(n \log n) = O(n \log h)$.
2. $n' < n^{.96}$. In this case the total additional work performed is clearly $O(n + n^{.96} \log n)$, which is $O(n)$.

Therefore, if all the calls to randomized operations succeed, then the total work is $O(n \log h)$. The lemma follows, then, as we have already argued that all such calls succeed with n-exponential probability. □

4.1. Processor allocation

As described above, our algorithm assumes that there is a virtual processor assigned to each element in the input. More importantly, our $W = O(n \log h)$ bound on the total work needed to achieve a running time of $T = O(\log n)$ depends upon the assumption that once a point becomes deactivated its processor becomes inactive. Of course, this raises a significant issue regarding processor allocation in the PRAM model of computation, where we must simulate the work of these virtual processors using some given number, P, of processors. Fortunately, we can apply the following lemma of Matias and Vishkin [23] to resolve this issue.

Lemma 4.2. *Given an algorithm with a work bound W, and time bound T, then, assuming that W is known, the algorithm can be simulated with P processors in time $T' = O(T + W/P + T_c \log T)$ with work $W = O(PT + W + PT_c \log T)$, where T_c is the time to perform linear approximate compaction using P processors.*

There are several methods for performing linear approximate compaction (where the target array is proportional to the number of elements being compressed) in $O(\log^* n)$ time using $O(n)$ processors, with n-exponential probability (e.g., see [23]). Of course, to apply the above lemma to our convex hull algorithm, we must also know the value of W, which seems to require that we know the value of h. We can get around this seeming circular argument, however, by a well-known trick (e.g., see [6]). To apply this trick to our method we begin by setting $h' = 2$ and we then run our algorithm to construct the convex hull in $O(\log n)$ time and $W = O(n \log h')$ work, with n-exponential probability, except that we stop processing should the method of Lemma 4.2 determine that the total work will exceed W work. We then square the value of h' and try again, using the value of W determined by this new value of h'. Should this run of our algorithm also be cut short, then we continue squaring the value of h' and rerunning our algorithm until we complete the process without exceeding the current bound

on W. Since we square the value of h' with each iteration, the running time of this process will still be $O(\log n)$ with n-exponential probability, and the total work needed will be bound by the final run of the algorithm, which will be $O(n \log h') = O(n \log h)$. Therefore, we have the following theorem.

Theorem 4.3. *Given n points in the plane, one can find their convex hull in $O(\log n)$ time using $O(n \log h)$ work, with n-exponential probability, on a randomized* CRCW PRAM, *where h is the total number of vertices in the hull.*

Acknowledgements

We would like to thank Omer Berkman for discussions that led to the development of the $O(\log^* n)$ time algorithm for presorted input 2D convex hull computation. We would also like to thank Mikhail Atallah and Richard Cole for discussions related to the folklore Lemma 2.7 and Theorem 2.12, and Yossi Matias for some helpful discussions regarding processor allocation.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing and C. Yap, Parallel computational geometry, Algorithmica 3 (1988) 293–327.

[2] N. Alon and N. Megiddo, Parallel linear programming in fixed dimension almost surely in constant time, in: Proc. 31st IEEE Symp. Found. Comput. Sci. (1990) 574–582.

[3] M.J. Atallah and M.T. Goodrich, Efficient parallel solutions to some geometric problems, J. Parallel Distrib. Comput. 3 (1986) 492–507.

[4] M.J. Atallah and M.T. Goodrich, Parallel algorithms for some functions of two convex polygons, Algorithmica 3 (1988) 535–548.

[5] O. Berkman, B. Schieber and U. Vishkin, A fast parallel algorithm for finding the convex hull of a sorted point set, Internat. J. Comput. Geom. Appl., to appear.

[6] B. Chazelle and J. Matoušek, Derandomizing an output-sensitive convex hull algorithm in three dimensions, Technical Report, Department of Computer Science, Princeton University, 1992.

[7] D. Chen, Efficient geometric algorithms on the EREW PRAM, IEEE Trans. Parallel Distrib. Syst. 6 (1) (1995) 41–47.

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

[9] A.L. Chow, Parallel algorithms for geometric problems, Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1980.

[10] A.L. Chow, A parallel algorithm for determining convex hulls of sets of points in two dimensions, in: Proc. 19th Allerton Conf. Commun. Control Comput. (1981) 214–223.

[11] X. Deng, An optimal parallel algorithm for linear programming in the plane, Inform. Process. Lett. 35 (1990) 213–217.

[12] H. Edelsbrunner, Algorithms in Combinatorial Geometry, EATCS Monographs on Theoretical Computer Science, Vol. 10 (Springer, Heidelberg, 1987).

[13] D. Eppstein and Z. Galil, Parallel algorithmic techniques for combinatorial computation, Annual Review of Computer Science 3 (1988) 233–283.

[14] M. Ghouse and M.T. Goodrich, In-place techniques for parallel convex hull algorithms, in: Proc. 3rd ACM Symp. Parallel Algorithms Architect. (1991) 192–203.

[15] M.T. Goodrich, Finding the convex hull of a sorted point set in parallel, Inform. Process. Lett. 26 (1987) 176–179.
[16] M.T. Goodrich, Constructing the convex hull of a partially sorted set of points, Computational Geometry: Theory and Applications 2 (5) (1993) 267–278.
[17] M.T. Goodrich, Y. Matias and U. Vishkin, Optimal parallel approximation for prefix sums and integer sorting, in: Proc. 5th ACM-SIAM Symp. on Discrete Algorithms (1994) 241–250.
[18] R.L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, Inform. Process. Lett. 1 (1972) 132–133.
[19] T. Hagerup and C. Rüb, Optimal merging and sorting on the erew pram, Inform. Process. Lett. 33 (1989) 181–185.
[20] T. Hagerup and C. Rüb, A guided tour of Chernoff bounds, Inform. Process. Lett. 33 (1989/90) 305–308.
[21] J. Jájá, An Introduction to Parallel Algorithms (Addison-Wesley, Reading, MA, 1992).
[22] D.G. Kirkpatrick and R. Seidel, The ultimate planar convex hull algorithm? SIAM J. Comput. 15 (1986) 287–299.
[23] Y. Matias and U. Vishkin, Converting high probability into nearly-constant time—with applications to parallel hashing, in: Proc. 23rd ACM Symp. Theory Comput. (1991) 307–316.
[24] R. Miller and Q.F. Stout, Efficient parallel convex hull algorithms, IEEE Trans. Comput. 37 (12) (1988) 1605–1618.
[25] J. O'Rourke, Computational Geometry in C (Cambridge University Press, Cambridge, 1994), code and errata available by anonymous ftp from grendel.csc.smith.edu (131.229.64.23), in the directory /pub/compgeo.
[26] F.P. Preparata and M.I. Shamos, Computational Geometry: An Introduction (Springer, New York, 1985).
[27] P. Ragde, The parallel simplicity of compaction and chaining, in: Proc. 17th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 443 (Springer, Berlin, 1990) 744–751.
[28] S. Sen, Finding an approximate median with high probability in constant parallel time, Inform. Process. Lett. 34 (1990) 77–80.
[29] A.C. Yao, A lower bound to finding convex hulls, J. ACM 28 (1981) 780–787.