International Journal of Computational Geometry & Applications

A FAST PARALLEL ALGORITHM FOR FINDING THE CONVEX HULL OF A SORTED POINT SET *

OMER BERKMAN
Dept. of Computing, King's College London, The Strand
London WC2R 2LS, England

BARUCH SCHIEBER
IBM Research Division, T.J. Watson Research Center, P.O. Box 218
Yorktown Heights, New York 10598, USA

and

UZI VISHKIN
Institute for Advanced Computer Studies (UMIACS)
and Dept. of Electrical Engineering, University of Maryland
College Park, Maryland 20742, USA
Dept. of Computer Science, Tel Aviv University
Tel Aviv 69978, Israel

Received 28 March 1994
Revised December 8, 1994
Communicated by M. T. Goodrich

ABSTRACT

We present a parallel algorithm for finding the convex hull of a sorted point set. The algorithm runs in $O(\log\log n)$ (doubly logarithmic) time using $n/\log\log n$ processors on a Common CRCW PRAM. To break the $\Omega(\log n/\log\log n)$ time barrier required to output the convex hull in a contiguous array, we introduce a novel data structure for representing the convex hull. The algorithm is optimal in two respects: (1) the time-processor product of the algorithm, which is linear, cannot be improved, and (2) the running time, which is doubly logarithmic, cannot be improved even by using a linear number of processors. The algorithm demonstrates the power of the "the divide–and–conquer doubly logarithmic paradigm" by presenting a non-trivial extension to situations that previously were known to have only slower algorithms.

Keywords: Parallel algorithms, computational geometry, convex hull, PRAM.

1. Introduction

We consider the following problem: given n points in the plane in a sorted order (e.g., by increasing x-coordinates), find which of these points belong to the perimeter of the smallest convex region containing all these points.

*The results of this paper were stated in Berkman *et al.* [4].

The model of parallel computation used in this paper is the Concurrent Read Concurrent Write (CRCW) Parallel Random Access Machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for both read and write operations. We assume a weak CRCW PRAM model in which several processors may attempt to write simultaneously at the same location only if they write the *same* value. (This model is called Common CRCW PRAM.)

Main result. We give a parallel algorithm for the problem that runs in $O(\log\log n)$[a] (doubly logarithmic) time using $n/\log\log n$ processors.

Output representation. The standard convex hull representation is a contiguous array containing the vertices of the convex hull. It is easy to see that the problem of counting the number of 1-bits in an n-bit input can be reduced to the problem of computing a convex hull of n sorted points in this array representation. Thus the lower bound for counting (parity) of Beame and Hastad [3] implies that the computation of such convex hull representation requires $\Omega(\log n/\log\log n)$ time if the number of processors used is polynomial. To break this $\Omega(\log n/\log\log n)$ time barrier, we introduce a novel data structure for convex hull representation. Given this data structure it can be determined for every vertex whether it belongs to the convex hull or not in $O(\log\log n)$ time using $n/\log\log n$ processors. Alternatively, a linked list representation of the convex hull can be computed within the same complexity bounds. A similar implicit representation of the convex hull was previously used by Ghouse and Goodrich [9].

Optimality of the main result. Our algorithm is optimal in two respects: First, its time-processor product matches the lower bound on the time complexity of any sequential algorithm for this problem. This is because any sequential algorithm for this problem requires at least linear time. Second, its running time matches the lower bound on the running time of any CRCW PRAM algorithm for this problem that uses a linear number of processors. This is because our problem is at least as hard as finding the maximum (as explained below), and thus the lower bound for finding the maximum ([17,6]) applies to our problem.

Previous and related results. A linear time sequential algorithm for this problem was given by Graham [11]. Aggarwal *et al.* [1] and Atallah and Goodrich [2] gave parallel algorithms for finding the convex hull of an arbitrary set of points in the plane. They first apply sorting, and then find the convex hull of the sorted point set. Their bound for this problem is $O(\log n)$ time using n processors. Goodrich [10] gave an (optimal) $O(\log n)$ time using $n/\log n$ processors algorithm for finding the convex hull of a sorted point set – the same problem as in the present paper. Both results are for the weaker Concurrent Read Exclusive Write (CREW) PRAM. Fjällström *et*

[a] Throughout this paper the logarithm base is 2, unless otherwise stated.

al. [8] gave the first sublogarithmic algorithm for the problem. Their algorithm runs in $O(\log n/(\log\log n))$ time using $n\log\log n/\log n$ processors on a Common CRCW PRAM. Recently, Ghouse and Goodrich [9] gave a *randomized* algorithm for our problem that runs *almost surely* in constant time using $n\log n$ processors, or almost surely in $O(\log^* n)$ time using $n/\log^* n$ processors on a CRCW PRAM. For a general reference on super fast (i.e., doubly logarithmic time or faster) parallel algorithms see Vishkin [18].

2. Motivation and significance of the main result

Following Valiant [17] and Shiloach and Vishkin [15], most of the (optimal) doubly logarithmic parallel algorithms follow a "divide–and–conquer paradigm" described below. Suppose that we are given a problem instance of size n, for which there is a linear time sequential algorithm. The paradigm has two parts: Prologue and Main Part. To materialize the abstract paradigm the reader may find it useful to visualize it on the problem of finding the maximum.

Prologue: Partition the size n instance into $n/\log\log n$ sub-instances of size $\log\log n$ each. Solve each sub-instance separately using the sequential algorithm in $O(\log\log n)$ time. Concatenate the solutions to form an instance of size $\tilde{n} = O(n/\log\log n)$.

The main Part: The main part is recursive. In the top recursion level the instance of size $\tilde{n}$ is partitioned into $\tilde{n}/m$ sub-instances, each of size approximately m. Each of these sub-instances is solved recursively using m processors, and the solutions are concatenated to form an instance of size $O(\tilde{n}/m)$. Finally, this instance is solved, using $\tilde{n}$ processors.

Allowing a doubly exponential progression for m (i.e., $m = \tilde{n}^{1/c}$, for some constant $c > 1$), gives a total of $O(\log\log n)$ recursion levels. Implementing each recursion level in *constant* time, gives a total running time of $O(\log\log n)$ time using $\tilde{n}$ processors for the main part, and $O(\log\log n)$ time using $n/\log\log n$ processors for the whole algorithm.

The main difficulty in extending this paradigm to cope with the problem of finding the convex hull of a sorted point set is in the implementation of each recursion level in constant time. At first glance it seems that there is no hope for such an implementation: to get a consecutive list of the vertices in the convex hull of each sub-instance (as obtained in previous parallel convex hull algorithms) $\Omega(\log n/\log\log n)$ time is required (with any polynomial number of processors), as follows from the lower bound for parity [3]. We circumvent this by using an alternative representation of the convex hull of each sub-instance. We call this representation the *implicit representation.* At this early stage in the presentation, we only point out that this representation enables answering queries of the form "Is a specific vertex on the convex hull of a specific subchain of points?" in constant time using $O(\log\log n)$ processors. We also show how this implicit representation enables us to determine for each vertex whether it belongs to the convex hull or not, and to output the vertices of the convex hull as a *linked* list.

Finally, we substantiate our claim that the problem is at least as hard as the problem of finding the maximum. Suppose that we need to find the maximum of n (distinct) numbers $a_1, \ldots, a_n$. We find the convex hull of the sorted point set $\{(i, a_i)\}_{i=1}^{n}$. Then, the y-coordinate of the unique point in the convex hull whose y-coordinate is greater than the y-coordinates of its two neighbors in the convex hull is the desired maximum. This point can be found in constant time using n processors. Hence, any algorithm for our problem that runs in $O(t)$ time using n/t processors implies an algorithm with the same bounds for the maximum problem.

The rest of this paper is organized as follows. Section 3 gives an overview of our algorithm. Section 4 describes the data structure used for the implicit representation of the convex hull. Section 5 gives the implementation details. Finally, Section 6 summarizes our results.

3. Overview of the algorithm

Let $P = (p_1, p_2, \ldots, p_n)$ be the input vertices sorted by their x-coordinates. Clearly, p_1 and p_n are in the convex hull of P. They partition the convex hull into two sets: the *upper hull* consisting of points from p_1 to p_n, inclusive, in the clockwise direction, and the *lower hull* consisting of points from p_n back to p_1, inclusive, in the clockwise direction. We focus only on finding the upper hull; finding the lower hull is similar. The algorithm is divided into three parts: Prologue, Main Part and Epilogue.

Prologue: Partition P into $\tilde{n} = \lceil n/\log n \rceil$ subchains $P_i = (p_{(i-1)\lceil \log n \rceil + 1}, \ldots, p_{i\lceil \log n \rceil})$, $1 \leq i \leq \tilde{n}$, of $\lceil \log n \rceil$[b] vertices each. (The last subchain may contain less vertices.) Find the upper hull of each $\log n$-subchain. These upper hulls are the input to the main part of the algorithm.

The Main Part: The main part consists of $O(\log\log n)$ recursion levels. In the top recursion level partition the n vertices of the polygon into $m = n^{1/3}$ groups of $m^2 = n^{2/3}$ vertices each (thus each group contains $n^{2/3}/\log n$ subchains), and compute the upper hull of each such group recursively. Then, merge the m upper hulls $L_1, \ldots, L_m$ into a single upper hull, as follows.

Following Goodrich [10] we define the *upper common tangent* of two upper hulls L_i and L_j, where L_i is to the left of L_j, to be the unique line that passes through a point of L_i and a point of L_j and such that all other points of L_i and L_j are below it. The left (resp. right) *endpoint* of this tangent is where it touches L_i (resp. L_j). We say that this tangent is a *left incoming* tangent of L_j and a *right outgoing* tangent of L_i.

The merging is done in three steps.

Step 1: Find the upper common tangent of each pair of upper hulls L_i and L_j.

Step 2: For each upper hull L_i, find its left incoming tangent with minimum slope and its right outgoing tangent with maximum slope.

Step 3: Given these tangents compute the upper hull of $L_1, \ldots, L_m$.

[b] To avoid cumbersome notations, we henceforth omit the $\lceil . \rceil$ and $\lfloor . \rfloor$ notations. Wherever a fractional quantity is referenced assume that the appropriate integral part is taken.

The above description comes with the following *caveat*: only *implicit representation* of upper hulls is actually computed.

The recursion is terminated when a group consists of a single $\log n$-subchain. It is easy to see that the number of recursion levels is bounded by $\log_{1.5}(\log n)$.

Epilogue: First, mark for each vertex whether it is in the upper hull or not. This is done using the partition of P into $\tilde{n}$ subchains of size $\log n$, as follows. For each subchain P_i, $1 \leq i \leq \tilde{n}$, find (using the output of the main part) its leftmost vertex, p_ℓ, and its rightmost vertex, p_r, which are in the upper hull of P (if such exist). A vertex p_j in P_i is on the upper hull of P if and only if it is in the upper hull of $p_\ell, \ldots, p_r$. This upper hull was computed in the Prologue. Given the markers, compute a linked list of the convex hull vertices.

4. The data structure

Before giving the implementation details of our algorithm we describe the data structure that is used for the implicit representation of the upper hulls. It is a balanced doubly logarithmic height tree T with $\tilde{n}$ leaves. Define the *depth* of a node to be its distance from the root, and the *height* of a node to be its distance from the leaves in its subtree. (Since the tree is balanced all these leaves are at the same distance.) The number of nodes of T in depth d is $n^{1-(2/3)^d}$, each of which has $n^{1/3 \cdot (2/3)^d}$ children. It follows that the height of the tree is bounded by $\log_{1.5}(\log n) = O(\log\log n)$. We relate to the nodes of the tree as 'nodes' and to the vertices of P as 'vertices'.

Each leaf of T corresponds to a $\log n$-subchain of P. For each internal node v of T, define the *vertices of* v to be the vertices of P that are included in the leaves (i.e., in the $\log n$-subchains) of the subtree rooted at v. Observe that the vertices of internal nodes of depth d correspond to the subchains whose upper hull is computed in the d-th recursion level. Define the upper hull of v to be the upper hull of its vertices.

Consider a node v of height h. The data structure consists of the following information for each child v_j of v in T. (See Figure 4.)

- The leftmost vertex $p_{\mathtt{left}(v_j)}$ of v_j that is in the upper hull of v, and the rightmost vertex $p_{\mathtt{right}(v_j)}$ that is in the upper hull of v. (The case that none of the vertices of v_j is on the upper hull of v is denoted by default $\mathtt{left}(v_j) = \mathtt{right}(v_j) = -1$.)
- Two pointers $\mathtt{ls}(v_j)$ and $\mathtt{rs}(v_j)$ that point respectively, to the nearest left sibling and the nearest right sibling of v_j that contain vertices of the upper hull of v, if such exist. (In other words, the nearest siblings of v_j whose `left` and `right` values are not -1.)

Lemma 1 *Let v be a node of T and let v_j be its child. A vertex p_x of v_j is in the upper hull of v if and only if it is in the upper hull of v_j and $x \in [\mathtt{left}(v_j), \ldots, \mathtt{right}(v_j)]$.*

Proof. Let p_x be a vertex in v_j. If $x \notin [\mathtt{left}(v_j), \ldots, \mathtt{right}(v_j)]$, then p_x is not in the upper hull of v. If $x = \mathtt{left}(v_j)$ or $x = \mathtt{right}(v_j)$, then p_x is in the upper

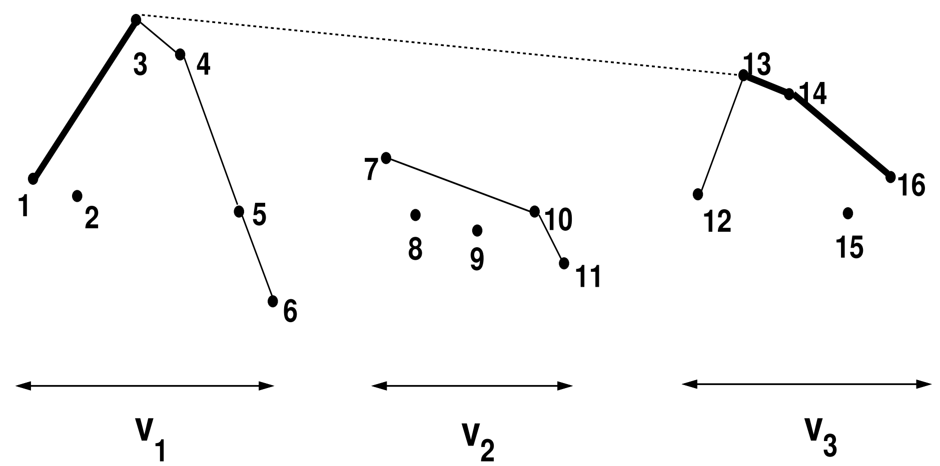

Fig. 1. The vertices of the children of v: v_1, v_2 and v_3.
The bold lines indicate the parts of the upper hulls of v_1, v_2 and v_3 that are also in the upper hull of v. In the example: $\texttt{left}(v_1) = 1$; $\texttt{right}(v_1) = 3$; $\texttt{left}(v_2) = -1$; $\texttt{right}(v_2) = -1$; $\texttt{left}(v_3) = 13$; $\texttt{right}(v_3) = 16$; $\texttt{ls}(v_1) = -1$; $\texttt{rs}(v_1) = v_3$; $\texttt{ls}(v_2) = v_1$; $\texttt{rs}(v_2) = v_3$; $\texttt{ls}(v_3) = v_1$; $\texttt{rs}(v_3) = -1$.

hull of v and in the upper hull of v_j. Suppose that $\texttt{left}(v_j) < x < \texttt{right}(v_j)$. If p_x is in the upper hull of v then it clearly is in the upper hull of v_j. Conversely, assume p_x is not in the upper hull of v and let e denote the edge of the upper hull of v that goes above p_x. Since both $p_{\texttt{left}(v_j)}$ and $p_{\texttt{right}(v_j)}$ belong to the upper hull of v, the edge e must connect two vertices of v_j between $p_{\texttt{left}(v_j)}$ and $p_{\texttt{right}(v_j)}$; thus p_x is not in the upper hull of $p_{\texttt{left}(v_j)}, \ldots, p_{\texttt{right}(v_j)}$, and hence not on the upper hull of v_j. □

In the d-th recursion level we compute the data structure for all nodes of depth d, given the data structure for nodes of larger depth. We concentrate on one node v of T of depth d and height h. In the implementation we need to retrieve the following information about a vertex p_x of a child v_j of v.

1. Find out whether p_x is in the upper hull of v_j.

2. Find the nearest vertices to the left and right of p_x that are in the upper hull of v_j, if such exist.

We show how to retrieve this information from our data structure in constant time using $O(\log\log n)$ processors. Let w be the *highest* node of height at most $h-2$ such that (i) p_x is one of the vertices of w, and (ii) $x \notin [\texttt{left}(w), \ldots, \texttt{right}(w)]$ (which includes the case that $\texttt{left}(w) = \texttt{right}(w) = -1$). Note that if w exists, then it is a proper descendant of v_j. We break into two cases depending on whether such node w exists.

If w exists, then, clearly, p_x is not in the upper hull of the parent of w and is therefore not in the upper hull of v_j. We show how to find the nearest vertices to p_x that are in the upper hull of v, if such exist. Suppose that $x < \texttt{left}(w)$, the cases

$x > \texttt{right}(w)$ and $\texttt{left}(w) = \texttt{right}(w) = -1$ are handled similarly. Then, by definition, $\texttt{left}(w)$ is the index of the nearest vertex to the right of p_x on the upper hull of the parent of w, and if $\texttt{ls}(w) \neq -1$ then $\texttt{right}(\texttt{ls}(w))$ is the index of the nearest vertex to the left of p_x in the upper hull of the parent of w. We claim that these two vertices are also in the upper hull of v_j. If v_j is the parent of w, then this is trivial. Otherwise, this claim can be proved inductively, since for all ancestors z of w up to (and not including) v_j, $x \in [\texttt{left}(z), \ldots, \texttt{right}(z)]$, and hence also $\texttt{right}(\texttt{ls}(w)), \texttt{left}(w) \in [\texttt{left}(z), \ldots, \texttt{right}(z)]$. (This property follows from our selection of w.)

If w does not exist, then p_x is in the upper hull of v_j if and only if it is in the upper hull of the leaf s of T that contains the $\log n$-subchain of p_x. If p_x is not in the upper hull of s then the nearest vertices to the left and right of p_x that are in the the upper hull of s are also in the upper hull of v_j. Suppose that p_x is in the upper hull of s. We show how to find the nearest vertex to the left of p_x that is also in the upper hull of v_j. The nearest vertex to the right of p_x that is also in the upper hull of v_j is found similarly. Test, using the procedure above, whether p_{x-1} is in the upper hull of v_j. If p_{x-1} is in the upper of v_j then it is the needed vertex to the left of p_x. Otherwise, the test with respect to p_{x-1} provides the nearest vertex to the left of p_{x-1} (and thus the nearest vertex to the left of p_x) that is in the upper hull of v_j.

The only difficulty in implementing the above information retrieval in constant time using $O(\log \log n)$ processors is the computation of the node w. This is done as follows. Allocate a processor to each node in the path from s, the $\log n$-subchain that contains p_x, to (and not including) v_j (totaling at most $\log_{1.5}(\log n)$ processors). Mark each such node z by 1 if $x \notin [\texttt{left}(z), \ldots, \texttt{right}(z)]$, and by 0 otherwise. If we index these nodes starting from the child of v_j, then w is the node with minimum index that is marked 1. This minimum index can be found in constant time with the allocated processors, using the algorithm given by Fich *et al.* [7]. (Fich *et al.* [7] show how to find the maximum of t numbers in the range $[1, \ldots, t]$ in constant time using t processors.)

5. Implementation and correctness

Prologue: The Prologue is implemented by applying the algorithm given in [10] to each of the $n/\log n$ subchains of length $\log n$. This takes $O(\log \log n)$ time using $n/\log \log n$ processors.

The Main Part: Again, we concentrate on one node v of depth d. Recall that v has $m = n^{1/3 \cdot 2/3^d}$ children each of which corresponds to a subchain of length m^2.

Step 1: In this step we find the upper common tangent of the upper hulls of all pairs of children of v. Let v_ℓ and v_r be two children of v, where $\ell < r$. Let L be the upper hull of v_ℓ and R be the upper hull of v_r. The length of each of these hulls is bounded by m^2. The following Lemma by Overmars and Van Leeuwen [13] implies a binary-search-like procedure for finding the tangent of L and R in $O(\log m)$ time using one processor.

Lemma 2 ([13]) *Let $L = (l_1, \ldots, l_a)$ and $R = (r_1, \ldots, r_b)$ be two upper hulls such* *that L is to the left of R. Given any i and j, $1 \le i \le a$, $1 \le j \le b$, one processor can find in constant time at least one of the following five conditions that holds:*

1. *The upper common tangent of L and R is (l_i, r_j).*
2. *None of the vertices $l_1, \ldots, l_i$ is an end point of the upper common tangent.*
3. *None of the vertices $l_{i+1}, \ldots, l_a$ is an end point of the upper common tangent.*
4. *None of the vertices $r_1, \ldots, r_j$ is an end point of the upper common tangent.*
5. *None of the vertices $r_{j+1}, \ldots, r_b$ is an end point of the upper common tangent.*

Furthermore, this can be done by examining the five edges (l_i, r_j), (l_{i-1}, l_i), (l_i, l_{i+1}), (r_{j-1}, r_j), and (r_j, r_{j+1}).

Following Karp and Miranker [12] and Snir [16], we show how to speed the search by a factor of $\log p$ using $p \log\log n$ processors. The extra factor of $\log\log n$ is needed since L and R are not given explicitly.

Lemma 3 *Let v_ℓ and v_r be two children of v. Let L and R be their upper hulls, respectively. The tangent of L and R can be found in $O(\log_p m)$ time using $p \log\log n$ processors.*

Proof. Denote the vertices of v_ℓ by $l_1, \ldots, l_{m^2}$ and the vertices of v_r by $r_1, \ldots, r_{m^2}$ (where $l_1, \ldots, l_{m^2} = p_\alpha, \ldots, p_{\alpha+m^2-1}$ for some α and $r_1, \ldots, r_{m^2} = p_\beta, \ldots, p_{\beta+m^2-1}$ for some β). Let $s = m^2/(\sqrt{p}+1)$. Allocate $\log\log n$ processors to each pair of vertices $(l_{i\cdot s}, r_{j\cdot s})$ of v_ℓ, v_r, $1 \le i, j \le \sqrt{p}$ (totaling in $p \log\log n$ processors). Using these processors find the nearest vertices to the right of $l_{i\cdot s}$ and $r_{j\cdot s}$ that are in L and R, respectively. Denote these vertices $\mathtt{succ}(l_{i\cdot s})$ and $\mathtt{succ}(r_{j\cdot s})$. This can be done in constant time, as described (with respect to the vertex p_x) above. Mark the vertex $\mathtt{succ}(l_{i\cdot s})$ if all vertices to its left or all vertices to its right are not endpoints of the upper common tangent. (That is, if conditions 1,2 or 3 of Lemma 2 hold with respect to $\mathtt{succ}(l_{i\cdot s})$ and $\mathtt{succ}(r_{j\cdot s})$.) Similarly, mark the vertex $\mathtt{succ}(r_{j\cdot s})$ if all vertices to its left or all vertices to its right are not endpoints of the upper common tangent. (That is, if conditions 1,4 or 5 of Lemma 2 hold.) Notice that at least one of these two vertices is marked. To apply Lemma 2 above, the nearest vertices of $\mathtt{succ}(l_{i\cdot s})$ and $\mathtt{succ}(r_{j\cdot s})$ that are on L and R, respectively, have to be computed. This can be done in constant time using the available processors, as described (with respect to the vertex p_x) above.

Since the above marking process is done for all pairs, it follows that either all vertices $\mathtt{succ}(l_{i\cdot s})$, or all vertices $\mathtt{succ}(r_{i\cdot s})$, for $1 \le i \le \sqrt{p}$ are marked. Suppose that all vertices $\mathtt{succ}(l_{i\cdot s})$ are marked. (The other case is analogous.) Then, there is a unique index $1 \le j \le \sqrt{p}$ such that all vertices to the left of (and including) $\mathtt{succ}(l_{j\cdot s})$ and all vertices to the right of $\mathtt{succ}(l_{(j+1)\cdot s})$ are not endpoints of the upper common tangent. This implies that the only vertices of v_ℓ that are candidates for being endpoints of the upper common tangent are the vertices of v_ℓ between $\mathtt{succ}(l_{j\cdot s})$ and $\mathtt{succ}(l_{(j+1)\cdot s})$. Notice that the index of $\mathtt{succ}(l_{j\cdot s})$ is at least $j \cdot s$, and that none of the vertices between $l_{(j+1)\cdot s}$ and $\mathtt{succ}(l_{(j+1)\cdot s})$ (excluding

$\texttt{succ}(l_{(j+1)\cdot s})$) are in L. Hence, the number of vertices of v_l that are candidates for being the left endpoint of the upper common tangent is bounded by s. Finding j in constant time using p processors is straightforward.

The above procedure reduces the size of one of the chains by a factor of at least $\sqrt{p}$. Thus the upper common tangent can be found by applying this procedure $O(\log_p m)$ times, where each application takes constant time. □

To implement Step 1 in constant time we take $p = m^{0.8}$, so that $O(\log_p m) = O(1)$. To calculate the number of processors used, consider a node v of depth d. For each of the m^2 pairs of children of v the implementation requires $p \log\log n = m^{0.8} \log\log n$ processors. Since there are n/m^3 nodes of depth d, the total number of processors is

$$m^2 \cdot m^{0.8} \log\log n \cdot \frac{n}{m^3} = \frac{n \log\log n}{m^{0.2}} = O(n/\log\log n).$$

The last equality follows from the fact that the recursion terminates when the number of leaves of each node is $\log n$.
Step 2: To find the left incoming tangent with minimum slope and the right outgoing tangent with maximum slope, for each child v_j of v, we solve one minimum problem and one maximum problem each with input size at most m. This is done in constant time with $m^{1.6}$ processors using the algorithm for finding the maximum of [15]. Over all children of v this requires $m^{2.6}$ processors which sums to $n/m^{0.4}$ processors over all nodes of depth d.
Step 3: To compute the information for each child v_j of v we use the following lemma from [10].
Lemma 4 ([10]) *Let $L_1, L_2, \ldots, L_m$ be upper hulls, where L_i is to the left of L_{i+1}, for $1 \leq i \leq m-1$. Let $t_{i,\ell}$ be the tangent with minimum slope among the left incoming tangents of L_i, and let $t_{i,r}$ be the tangent with maximum slope among the right outgoing tangents of L_i. Then, by computing the angle between $t_{i,\ell}$ and $t_{i,r}$, one processor can determine in constant time which of the following two cases occur.*

1. *No vertex of L_i is in the upper hull of $L_1, \ldots, L_m$.*
2. *All the vertices between (and including) the right endpoint of $t_{i,\ell}$ and the left endpoint of $t_{i,r}$ are in the upper hull of $L_1, \ldots, L_m$.*

It follows that a single processor that is allocated to each child v_j of v determines the values $\texttt{left}(v_j)$ and $\texttt{right}(v_j)$ in constant time. We compute the $\texttt{ls}(v_j)$ and $\texttt{rs}(v_j)$ for each child v_j of v as follows. Allocate m processors to each vertex v. Mark each child v_j of v by 0 if $\texttt{left}(v_j) = \texttt{right}(v_j) = -1$, and by 1 otherwise. Consider a child v_j. It is easy to see that $\texttt{ls}(v_j)$ (resp. $\texttt{rs}(v_j)$) is the index of the nearest left (resp. right) sibling of v_j that is marked by 1. These indices can be easily found in constant time by allocating $m^{1.6}$ processors to each of the m children (totaling to $O(n/\log\log n)$ processors).
Epilogue: We show how to find for each subchain P_i, $1 \leq i \leq \tilde{n}$. the leftmost vertex, p_ℓ, and the rightmost vertex, p_r, which are in the upper hull of P_i (if such

exist). Consider such a subchain P_i. We find p_ℓ as follows. Pick $p_{i'}$, the leftmost vertex of P_i (specifically $i' = (i-1)\log n + 1$) and test whether it is in the upper hull of P. This can be done using the implicit representation of the upper hull of P in constant time and $\log\log n$ processors as demonstrated above. If $p_{i'}$ is not in the upper hull of P, then $p_\ell \in P_i$ is the nearest vertex to the right of $p_{i'}$ that is in the upper hull of P (if no such vertex exists then P_i does not have any vertices in the upper hull of P). This nearest vertex can also be found in constant time using $\log\log n$ processors as demonstrated above. The vertex p_r is found similarly. The correctness of this computation follows from Lemma 1. Finally, we use the chaining algorithm of [5] or [14] to chain all the marked vertices into a linked list within the stated complexity bounds.

6. Summary

We have presented an optimal doubly logarithmic time parallel algorithm for finding the convex hull of a sorted point set. The model of computation used is Common CRCW PRAM. To break the $\Omega(\log n/\log\log n)$ time barrier we introduced a novel data structure for representing the convex hull. It enables answering queries of the form "Is a specific vertex on the convex hull of a specific subchain of points?" in constant time using $O(\log\log n)$ processors. The data structure can be constructed without using a prefix sums routine.

Postscript

Recently, Wagener [19] presented an optimal $O(\log\log n)$-time $n/\log\log n$-processor algorithm for the more general problem of constructing the convex hull of a *simple* polygon.

Acknowledgements

We thank the referees for valuable comments and for pointing out References [8], and [9].

Part of this work was carried out while the first author was at the University of Maryland Institute for Advanced Computer Studies (UMIACS), and the Dept. of Computer Science, Tel Aviv University. The first author was partially supported by NSF grant CCR-8906949. The third author was partially supported by NSF grants CCR-8906949 and CCR-9111348.

References

1. A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, "Parallel computational geometry", in *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science* (1985) pp. 468–477.
2. M.J. Atallah and M.T. Goodrich, "Efficient parallel solutions to some geometric problems", *J. of Parallel and Distributed Comput.* **3** (1986) 492–507.
3. P. Beame and J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM", in *Proc. of the 19th Ann. ACM Symp. on Theory of Computing* (1987)

pp. 83–93.

4. O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, "Highly-parallelizable problems", in *Proc. of the 21st Ann. ACM Symp. on Theory of Computing* (1989) pp. 309–319.
5. O. Berkman and U. Vishkin, "Recursive *-tree parallel data-structure", *SIAM J. Comput.* **22** (1993) 221–242.
6. F.E. Fich, F. Meyer auf der Heide, P.L. Ragde, and A. Wigderson, "Lower bounds for parallel random access machines with unbounded shared memory", in *Advances in Computing Research, Vol. 4*, ed. F.P. Preparata (JAI Press, Greenwich, CT, 1987) pp. 1–15.
7. F.E. Fich, P.L. Ragde, and A. Wigderson, "Relations between concurrent-write models of parallel computation", *SIAM J. Comput.* **17** (1988) 606–627.
8. P-O. Fjällström, J. Katajainen, C. Levcopoulos, and O. Petersson, "A sublogarithmic convex hull algorithm", *BIT*, **30** (1990) 378–384.
9. M.R. Ghouse and M.T. Goodrich, "In-place techniques for parallel convex hull algorithms", in *Proc. of the 3rd Ann. ACM Symp. on Parallel Algorithms and Architectures* (1991) pp. 192–203.
10. M.T. Goodrich, "Finding the convex hull of a sorted point set in parallel", *Information Processing Letters*, **26** (1987) 173–179.
11. R.L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters*, **1** (1972) 132–133.
12. R.M. Karp and W.L. Miranker, "Parallel minimax search for a maximum", *J. of Combinatorial Theory*, **4** (1968) 19–34.
13. M.H. Overmars and J. Van Leeuwen, "Maintenance of configurations in the plane", *J. Comput. System Sci.* **23** (1981) 166–204.
14. P. Ragde, "The parallel simplicity of compaction and chaining", in *Proc. of the 17th Int. Colloquium Automata, Languages and Programming* (1990) pp. 744–751.
15. Y. Shiloach and U. Vishkin, "Finding the maximum, merging, and sorting in a parallel computation model", *J. Algorithms* **2** (1981) 88–102.
16. M. Snir, "On parallel searching", *SIAM J. Comput.* **14** (1985) 688–707.
17. L.G. Valiant, "Parallelism in comparison problems", *SIAM J. Comput.* **4** (1975) 348–355.
18. U. Vishkin, "Structural parallel algorithmics", in *Proc. of the 18th Int. Colloquium Automata, Languages and Programming* (1991) pp. 363–380.
19. H. Wagener, "Optimal parallel hull construction for simple polygons in $O(\log\log n)$ time", in *Proc. of the 33rd IEEE Ann. Symp. on Foundation of Computer Science* (1992) pp. 593–599.