Scalable 2d Convex Hull and Triangulation Algorithms for Coarse Grained Multicomputers*

M. Diallo† Afonso Ferreira‡ Andrew Rau-Chaplin§ Stéphane Ubéda¶

Abstract

In this paper we describe scalable parallel algorithms for building the Convex Hull and a Triangulation of a n co-planar points. These algorithms are designed for the *coarse grained multicomputer* model: p processors with $O(\frac{n}{p}) \gg O(1)$ local memory each, connected to some arbitrary interconnection network. They scale over a large range of values of n and p, assuming only that $n \geq p^{1+\epsilon}$ ($\epsilon > 0$) and require time $O(\frac{T_{sequential}}{p} + T_s(n,p))$, where $T_s(n,p)$ refers to the time of a global sort of n data on a p processor machine. Furthermore, they involve only a constant number of global communication rounds. Since computing either 2d Convex Hull or Triangulation requires time $T_{sequential} = \Theta(n \log n)$ these algorithms either run in optimal time, $\Theta(\frac{n \log n}{p})$, or in sort time, $T_s(n,p)$, for the interconnection network in question. These results become optimal when $\frac{T_{sequential}}{p}$ dominates $T_s(n,p)$ or for interconnection networks like the mesh for which optimal sorting algorithms exist.

1 Introduction

Most existing multicomputers consist of a set of p *state-of-the-art* processors, each with considerable local memory, connected to some interconnection network. These machines are usually *coarse grained*, i.e. the size of each local memory is "considerably larger" than $O(1)$. Despite this fact, until recently most theoretical parallel algorithms, in particular those for solving geometric problems, assume a fine grained setting, where a problem of size n is to be solved on a parallel computer with

*A preliminary version of this paper appeared in the *Proceedings of the 7-th IEEE Symposium on Parallel and Distributed Processing*, San Antonio, Tx., October 1995. Part of this work was done while the authors visited each other in Lyon and in Halifax. Support from the respective Institutions is acknowledged.

†LIP ENS-Lyon, 46, allée d'Italie, 69364 Lyon Cedex 7, France. E-mail: mdiallo@lip.ens-lyon.fr.

‡CNRS. Projet SLOOP, INRIA Sophia Antipolis, 2004, Route des Lucioles, 06902 Sophia Antipolis Cedex, France. Partially supported by an International Research Fellowship of the Région Rhône-Alpes.

§Technical University of Nova Scotia, P.O. Box 1000, Halifax, Nova Scotia, Canada B3J 2X4. E-mail: arc@tuns.ca. Partially supported by DIMACS, a NSF center (USA), and the Natural Sciences Engineering Research Council (Canada).

¶LIP ENS-Lyon, 46, allée d'Italie, 69364 Lyon Cedex 7, France. E-mail: ubeda@lip.ens-lyon.fr. Partially supported by the project PRS (CAPA) of the French CNRS.

p processors such that $\frac{n}{p} = O(1)$. However, to be relevant in practice such algorithms must be *scalable*, that is, they must be applicable and efficient for a wide range of ratios $\frac{n}{p}$.

Recently, there has been a growing interest in coarse grained computational models [10, 11, 15, 35] and the design of coarse gained algorithms [15, 21, 13, 16, 25, 14, 19]. The work on computational models has tended to be motivated by the observation that "fast algorithms" for fine-grained models rarely translate to fast code running on coarse grained machines. The BSP model, described by Valiant [35], uses slackness in the number of processors and memory mapping via hash functions to hide communication latency and provide for the efficient execution of fine grained PRAM algorithms on coarse grained hardware. The LogP model was then introduced which, using Valiant's BSP model as a starting point, focuses on the technological trend from fine grained parallel machines towards coarse grained systems and advocates portable parallel algorithm design [10]. Other coarse grained models focus more on utilizing local computation and minimizing global operations. These include the C^3 model [11], and the Coarse Grained Multicomputer (CGM) model used in this paper [15]. In this mixed sequential/parallel setting, there are three important measures of any coarse grained algorithm:

1. The amount of local computation required;

2. The number and type of global communication phases required;

3. The scalability of the algorithm, that is, the range of values for the ratio $\frac{n}{p}$ for which the algorithm is efficient and applicable.

This paper describes scalable parallel algorithms for two fundamental geometric problems, namely the 2d Convex Hull and Triangulation problems within the coarse grained multicomputer model that behave almost optimally with respect to all three measures above (See Table 1).

The Model

The *Coarse Grained Multicomputer* model, or $CGM(n, p)$ for short, is defined in [15]. It consists of a set of p processors treating a problem of size n. Each processor has $O(\frac{n}{p})$ local memory and they are connected to some arbitrary interconnection network or a shared memory. The term "*coarse grained*" refers to the fact that (as in practice) the number of words of each local memory $O(\frac{n}{p})$ is defined to be "considerably larger" than $O(1)$. This is clearly true for all currently available coarse grained parallel machines. In the following, when determining time complexities both local computation time and inter-processor communication time are considered in the standard way.

In this model, all global communications are performed by a small set of standard communications operations - Segmented broadcast, Segmented gather, All-to-All broadcast, Personalized All-to-All broadcast, Partial sum and Sort, which are typically efficiently realized in hardware. If a parallel machine does not provide these operations they can be implemented in terms of a constant number of sorting operations [15].

Moreover, recently it was shown that, given $p < n^{1-\frac{1}{c}}$ $(c \geq 1)$, sorting $O(n)$ elements distributed evenly over p processors in the BSP (or LogP) model can be achieved in $O(\log n/\log(h+1))$ communication rounds and $O(n \log n/p)$ local computation time, for $h = \Theta(\frac{n}{p})$, i.e. with optimal local computation and $O(1)$ h-relations, when $\frac{n}{p} \geq p$ [21]. Therefore, using this sort, the communication operations of the $CGM(s,p)$ can be realized in the BSP (or LogP) models in a constant number of h-relations, where $h = \Theta(\frac{s}{p})$.

Finding an optimal algorithm in the CGM model is equivalent to minimizing the number of global communication rounds as well as the local computation time. It has been shown that minimizing the number of round also results in improved portability across different parallel architectures[35, 36].

The 2D Convex Hull and Triangulation Problems

The 2D Convex Hull problem is perhaps the most fundamental problem in computational geometry and certainly the most studied [2]. In fact, it appears to be the first problem in computational geometry for which parallel algorithms were designed [32, 9, 1]. In the fine grained parallel setting, algorithms have been described for many architectures including the CRCW PRAM [1], the CREW PRAM [9], the Hypercube [31] and the Mesh [30]. In the coarse grained setting, there has recently been many of new results, as shown in Table 1. All of these new coarse grained results assume a machine with n data elements evenly over p processors and require $O(\frac{n \log n}{p})$ local computation time. Let ϵ be a fixed constant such that $\epsilon > 0$.

In [15], a deterministic convex hull algorithm requiring $O(\log n)$ communication phases, and being applicable for $n \geq p^2$, was presented. This algorithm had the advantage of being simple, deterministic and scalable over a large range of values of the ratio $\frac{n}{p}$, but required a non-constant number of communication rounds.

	Algorithm Type	Communication Phases	Scalability Assumption
Simple Merge	Deterministic	$O(\log n \log p)$	$n \geq p^2$
Dehne, Fabri, Rau-Chaplin (1993)	Deterministic	$O(\log n)$	$n \geq p^2$
Dehne, Fabri, Kenyon (1994)	Randomized	$O(1)$	$n \geq p^{3+\epsilon}$
Deng, Gu (1994)	Deterministic	$O(\log p)$	$n \geq p^{3+\epsilon}$
	Randomized	$O(1)$	$n \geq p^{3+\epsilon}$
This paper	Deterministic	$O(1)$	$n \geq p^{1+\epsilon}$

Figure 1: Comparison of Recent Coarse Grained Convex Hull Algorithms.

Then, in [14], a randomized convex hull algorithm was designed, requiring with high probability $O(1)$ communication phases and being applicable for $n \geq p^{3+\epsilon}$. This algorithm had the significant advantage of being able to solve the convex hull problem for points in more than two dimensions, but assumed that the points were uniformly distributed, worked in $O(1)$ communication phases only with high probability, and was only scalable over a significantly reduced range of values of the ratio $\frac{n}{p}$ in that it required $n \geq p^{3+\epsilon}$.

Finally, two new 2D convex hull algorithms, requiring $n \geq p^{3+\epsilon}$, were shown in [16]. Their deterministic algorithm required $O(\log p)$ communication phases, while their randomized algorithm required with high probability $O(1)$ communication phases. These algorithms had the advantage of not being restricted to uniformly distributed point sets, but still required more than a constant number of communication phases in the deterministic case, were rather complex, and again were only scalable over a significantly reduced range of values of the ratio $\frac{n}{p}$ in that they required $n \geq p^{3+\epsilon}$.

The Convex Hull algorithm described in this paper is deterministic, requires only $O(1)$ communication phases in the worst case and is highly scalable in that it is efficient and applicable for $n \geq p^{1+\epsilon}$.

The Triangulation Problem of a planar point set has many practical applications as in the finite element method and in numerical analysis. Several parallel algorithms have been proposed to solve it work-optimally in PRAM's [29, 37], time-optimally in linear arrays [8] and in sorting time in hypercubes [27]. All these algorithms are for fine-grained models. The coarse grained triangulation algorithm described in this paper is again deterministic, requires only $O(1)$ communication phases in the worst case and is highly scalable in that it is efficient and applicable for $n \geq p^{1+\epsilon}$.

The Results

In this paper we first describe a scalable coarse-grained deterministic algorithm for solving the 2d convex hull problem. Our algorithm requires time $O(\frac{n \log n}{p} + T_s(n,p))$. Furthermore, it involves only a constant number of global communication rounds and local memory space $\frac{n}{p} \geq p^{\epsilon}$. Based on this algorithm we also give an algorithm for solving the triangulation problem for points in $\mathcal{R}^2$ for the same model and with the same space and time complexities.

Since computing either 2d convex hull or triangulation requires time $T_{sequential} = \Theta(n \log n)$ [33] our algorithms either run in optimal time $\Theta(\frac{n \log n}{p})$ or in sort time $T_s(n,p)$ for the interconnection network in question. Our results become optimal when $\frac{T_{sequential}}{p}$ dominates $T_s(n,p)$ or when $T_s(n,p)$ is optimal.

Consider, for example, the *mesh* architecture. For the fine grained case ($\frac{n}{p} = O(1)$), a time complexity of $O(\sqrt{n})$ is optimal. Hence, simulating the existing fine grained results on a coarse grained machine via Brent's Theorem [26] leads to a $O(\frac{n}{p}\sqrt{n})$ time coarse grained method. Our algorithm runs in time $O(\frac{n}{p}(\log n + \sqrt{p}))$, a considerable improvement over both simulation and the existing methods.

For the problems studied in this paper, we are interested in algorithms which are optimal or at least efficient for a wide range of ratios $\frac{n}{p}$. We use a new technique for designing efficient scalable parallel geometric algorithms which depends neither on the partitioning technique of [15] nor the randomized techniques used in [14, 13, 16]. The key idea is to use "splitters", as introduced in [31], to sample local data and to perform computation (of supporting lines) that would be redundant in the sequential setting, but which reduces communication in our parallel setting. Our results are independent of the communication network. A particular strength of this approach (which is very different from the one presented in [4, 22]), is that all inter-processor communication is restricted to a constant number of usages of a small set of simple communication operations. This has the effect of making the algorithms both easy to implement, in that all communications are performed by calls to a standard highly optimized communication library, and very fast in practice (see Section 5 for evidence of this).

Outline of the Algorithms

Convex hull

Our Convex Hull algorithm is described below and has the following basic structure. The entire data set for a given problem is assumed to be initially distributed into the local memories and remains there until the problem is solved. Given a set S of n points and a p processor coarse grained multicomputer we show how to compute the upper hull of S. The lower hull, and therefore the complete hull, can be computed analogously. In the remainder we assume without loss of generality that all points are in the first quadrant.

Upper Hull(S)

Input: Each processor stores a set of $\frac{n}{p}$ points drawn arbitrarily from S.
Output: A distributed representation of the upper hull of S. All points on the upper hull are identified and labeled from left to right.

1. Globally sort the points in S by x-coordinate. Let S_i denote the set of $\frac{n}{p}$ sorted points now stored on processor i.

2. Independently and in parallel, each processor i computes the upper hull of the set S_i. Let X_i denote the result on processor i.

3. Compute for each upper hull X_i, $1 \leq i \leq p$, the upper common tangent lines between it and all upper hulls X_j, $i < j \leq p$, and label the upper hull of S by using the upper tangent lines.

Step 1 of algorithm UpperHull(S) can be completed by using a global sort operation as described in Section 2. Step 2 is a totally sequential step and can be completed in time $O(\frac{n \log n}{p})$ using well known sequential methods [33]. The main challenge is in performing Step 3. This step amounts to a merge algorithm in which p disjoint upper hulls are merged into a single hull. We present two different merge procedures: MergeHulls1 and MergeHulls2. The first, described in Section 3.4, is a straightforward merge requiring a constant number of global communication rounds and $\frac{n}{p} \geq p^2$ local memory per processor. The second merge procedure (MergeHulls2), described in Section 3.5, is a more complex merge that uses the first merge as a subprocedure but has a higher degree of scalability in that it can be implemented with only $\frac{n}{p} \geq p^{\epsilon}$ local memory, while still requiring a constant number of communication rounds. Both algorithms use the idea of selecting splitter sets which was introduced in the context of solving convex hulls in [31].

Triangulation

Our algorithm for triangulating n points in $\mathcal{R}^2$ is based on our convex hull algorithm and the observation that a set S of points can be triangulated by first triangulating x-disjoint subsets from S and then triangulating the regions between the convex hulls formed by these triangulated regions, which is a simpler subproblem as these regions belong to a class of simple polygons called *funnel polygons* [37]. The algorithm is as follows.

Triangulate (S)

Input: Each processor stores a set of $\frac{n}{p}$ points drawn arbitrarily from S.
Output: A distributed representation of a triangulation of S. All segment lines of the triangulation are identified and labeled.

1. Globally sort the points in S by x-coordinate. Let S_i denote the set of $\frac{n}{p}$ sorted points now stored on processor i.
2. Independently and in parallel, each processor i computes the convex hull of the set S_i, and the triangulation of its convex hull.
3. Compute upper and lower common tangent lines between every pair of convex hulls, and horizontal extreme points of consecutive convex hulls.
4. Label the upper funnel polygons.
5. Identify each point to its upper funnel polygon.
6. Triangulate the upper funnel polygons.
7. Repeat Steps 4-6 for the lower funnel polygons.

The remainder of this paper is organized as follows. In the next section we describe some basic operations for the coarse grained multicomputer model. Section 3 presents our Upper Hull algorithm, while Section 4 describes how the Upper Hull algorithm can be adapted to solve the triangulation problem. In Section 5 we show the results obtained with a PVM implementation of one of the versions of our algorithms on a Cray T3E. In Section 6 we present concluding remarks and ways for further research.

2 The Coarse Grained Model: Basic Operations

The *Coarse Grained Multicomputer*, $CGM(n,p)$, considered in this paper is a set of p processors numbered from 1 to p with $O(\frac{n}{p})$ local memory each, connected via some arbitrary interconnection network or a shared memory. We will assume that $\frac{n}{p} \geq p^{\epsilon}$ as was assumed in [14].

Global sort refers to the operation of sorting $O(n)$ data items stored on a $CGM(n,p)$, $O(\frac{n}{p})$ data items per processor, with respect to the *CGM*'s processor numbering. $T_s(n,p)$ refers to the time complexity of a global sort.

Note that, for a mesh $T_s(n,p) = \Theta(\frac{n}{p}(\log n + \sqrt{p}))$ and for a hypercube $T_s(n,p) = O(\frac{n}{p}(\log n + \log^2 p))$. These time complexities are based on [5] and [28], respectively. For the hypercube an asymptotically better deterministic algorithm exists [12], but it is of more theoretical than practical interest. We refer the reader to [5, 7, 24, 26, 28, 34] for a more detailed discussion of the different architectures and routing algorithms.

We will now outline four other operations for interprocessor communication which will be used in the remainder of this paper. All of these operations can be implemented as a constant number of global sort operations and $O(\frac{n}{p})$ time local computation. Note that, for most interconnection networks it would be better in practice to implement these operations directly rather than using global sort as this would typically improve the constants in the time complexity of the algorithms described in the remainder.

Segmented broadcast: In a segmented broadcast operation, $q \leq p$ processors with numbers $j_1 < j_2 < \ldots < j_q$ are selected. Each such processor, p_{j_i}, broadcasts a list of $1 \leq k \leq \frac{n}{p}$ data items from its local memory to the processors $p_{j_i+1} \ldots p_{j_{i+1}-1}$.

Segmented gather: In a segmented gather operation, $q \leq p$ processors with numbers $j_1 < j_2 < \ldots < j_q$ are selected. Each such processor, p_{j_i}, receives a data item from processors $p_{j_i+1} \ldots p_{j_{i+1}-1}$. This operation is the inverse of a segmented broadcast. Note that care must be taken to ensure that the selected processors have enough memory to receive all sent messages.

All-to-All broadcast: In an All-to-All broadcast operation, every processor sends one message to all other processors.

Personalized All-to-All broadcast: In a Personalized All-to-All broadcast operation, every processor sends a different message to every other processor.

Partial sum (Scan): Every processor stores some values, and all processors compute the partial sums of these values with respect to some associative operator.

In [15], it was shown that for any $CGM(n,p)$ with $\frac{n}{p} \geq p$ these operations are no more complex than sorting plus a linear amount of sequential work, i.e., they require $O(\frac{n}{p}+T_s(n,p))$. These results extend easily to the case where $\frac{n}{p} \geq p^{\epsilon}$, provided that each processor is to receive no more than p^{ϵ} data elements.

Therefore, using the sorting procedure from [21] all the above operations can be implemented also in the BSP model with a constant number of communication rounds.

3 Merging Convex Hulls in Parallel

In this section we show how to perform Step 3 of the upper hull algorithm given in the Introduction, by merging p disjoint upper hulls stored on a p-processor CMG, one per processor, into a single upper hull.

We denote by $\overline{ab}$ the line segment connecting the points a and b and by (ab) the line passing through a and b. A point c is said to be *dominated* by the line segment $\overline{ab}$ if and only if c's x-coordinate is strictly between the x-coordinates of a and b, and c is located below the line segment $\overline{ab}$. Definitions 1 and 2, as illustrated by Figure 2, establish the initial condition before the merge step.

Definition 1 *Let $\{S_i\}$, $1 \leq i \leq p$ be a partition of S such that $\forall x \in S_j$, $y \in S_i$, $j > i$, the x-coordinate of x is larger than that of y (see Figure 2).*

Definition 2 *Let $X_i = \{x_1, x_2, \ldots, x_m\}$ be an upper hull. Then, $pred_{X_i}(x_j)$ denotes x_{j-1} and $suc_{X_i}(x_j)$ denotes x_{j+1} (see Figure 3). We define $pred_{X_i}(x_1)= x_1$ and $suc_{X_i}(x_m)= x_m$.*

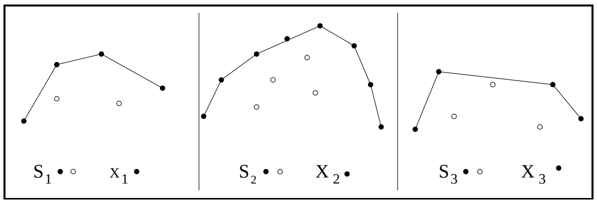


Figure 2: In this example $S = \{S_1, S_2, S_3\}$ and the points in S_i that are filled in are the elements of the upper hull of S_i, namely X_i.

Given two upper hulls $X_i = UH(S_i)$ and $X_j = UH(S_j)$ where all points in S_j are to the right of all points in S_i, the merge operations described in this paper are based on computing for a point

$q \in X_i$ the point $p \in X_i \cup X_j$ which follows q in $UH(S_i \cup S_j)$. The following definitions make this idea more precise (See Figure 3).

Definition 3 *Let $Q \subseteq S$. Then, $Next_Q : S \longrightarrow Q$ is a function such that $Next_Q(p) = q$ if and only if q is to the right of p and $\overline{pq}$ is above $\overline{pq'}$ for all $q' \in Q$, q' to the right of p.*

Definition 4 *Let $Y \subseteq S_i$. Then, $lm(Y)$ is a function such that $lm(Y) = y^*$ if and only if y^* is the leftmost point in Y such that $Next_{Y \cup S_j, j>i}(y^*) \notin S_i$.*

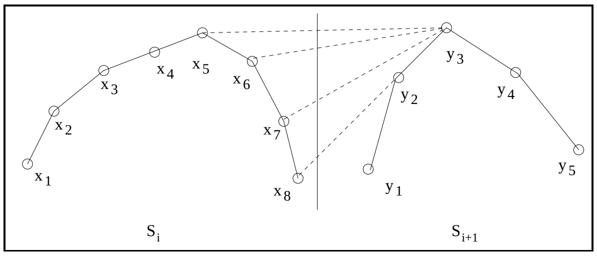


Figure 3: Let $S' = S_i \bigcup S_{i+1}$ then $suc(x_j) = x_{j+1}$, $x_3 = Next_{S'}(x_2)$, $Next_{S'}(x_3) = x_4$, $Next_{S'}(x_4) = x_5$, $Next_{S'}(x_5) = Next_{S'}(x_6) = Next_{S'}(x_7) = y_3$, $Next_{S'}(x_8) = y_2$, and $lm(S_i) = x_5$.

Let X represent the upper hull of a set of n points. Let also c be a point located to the left of this set. For the sake of completeness, we present below a sequential algorithm called **QueryFindNext** to search for $q = Next_X(c)$. Figure 4 illustrates one step of this algorithm. This binary search process takes time $O(\log |X|)$ [33].

Procedure QueryFindNext(X,c,q)

Input: an upper hull $X = \{x_1, \ldots, x_m\}$ sorted by x-coordinate and a point c to the left of x_1.

Output: a point $q \in X$, $q = Next_X(c)$.

1. If $X = \{x\}$ then $q \leftarrow x$ and halt.
2. If $\overline{x_{\lceil m/2 \rceil} suc(x_{\lceil m/2 \rceil})}$ is located below the line $(cx_{\lceil m/2 \rceil})$
   then QueryFindNext($\{x_1, \ldots, x_{\lceil m/2 \rceil}\}$,c,q), else QueryFindNext($\{x_{\lceil m/2 \rceil}, \ldots, x_m\}$,c,q).

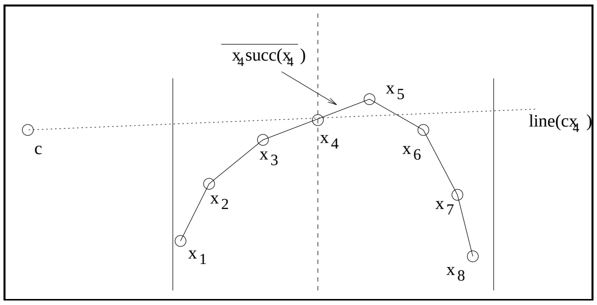


Figure 4: One step in the binary search procedure QueryFindNext. Since the line segment $\overline{x_4 suc(x_4)}$ is above the line (cx_4) the algorithm will recurse on $\{x_4, \ldots, x_8\}$.

3.1 Characterization of the upper hull

A classical way of characterizing the upper hull of a point set S, as given in [33], is based on the observation that "A line segment $\overline{ab}$ is an edge of the upper hull of a point set S located in the first quadrant if and only if all the $n-2$ remaining points fall below the line (ab)". We will work with a new characterization of the upper hull of S based on the same observation, but defined in terms of the partitioning of S given in Definitions 1 and 4.

Consider sets S, S_i and X_i as given in Definitions 1 and 2.

Definition 5 *Let $S' = \{c \in \bigcup X_i \mid c$ is not dominated by a line segment $\overline{x_i^* Next_{\bigcup X_j, j>i}(x_i^*)}, 1 \le i < p\}$, where $x_i^* = lm(X_i)$.*

We then have the following characterization of $UH(S)$.

Fact 1 $S' = UH(S)$.

3.2 Parallel merge algorithms

In this section we describe two parallel merge algorithms based on the characterization of $UH(S)$ given in the previous section and analyze their time and space complexity for the coarse grained model. The following definitions and lemma are needed in the description of the algorithms.

Definition 6 *Let $G_i \subseteq X_i$ and $g_i^* = lm(G_i)$. Let $R_i^- \subseteq X_i$ be composed of the points between $pred_{G_i}(g_i^*)$ and g_i^*, and $R_i^+ \subseteq X_i$ be composed of the points between g_i^* and $suc_{G_i}(g_i^*)$ (see Figure 5).*

We have then the following lemma.

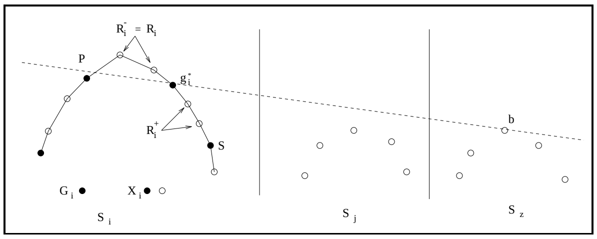


Figure 5: Filled points are element of G_i which is a subset of X_i composed of both of hollow and filled points. Let $p = pred_{G_i}(g_i^*)$ and $s = suc_{G_i}(g_i^*)$. Since the R_i^+ does not contain any point above the line $(g_i^* b)$ we have $R_i = R_i^-$.

Lemma 1 *One or both of R_i^+ or R_i^- is such that all its points are under the line $(g_i^* Next_{X_j, j>i}(g_i^*))$.*

The proof of this lemma is direct, otherwise $g_i^* \notin X_i$.

Definition 7 *Let R_i denote the set R_i^+ or R_i^- that has at least one point above the line $(g_i^* Next_{X_j, j>i}(g_i^*))$ (see Figure 5).*

Note that the size of the sets R_i is bounded by the number of points laying between two consecutive points in G_i.

3.3 Computing g_i^* and x_i^* in parallel

In the following we shall show how to compute $g_i^* = lm(G_i)$, where $G_i \subseteq X_i$. The key idea in Procedure FindLMSubset, described below, is to send the elements of G_i, at Step 2, to all larger-numbered processors so that processors receiving $G = \bigcup G_j, j < i$ can sequentially compute, at Step 3, the required points using Procedure QueryFindNext described in Section 3, and send the answers back at Step 4. Then, the processors can independently compute all g_i^*.

When the processors are divided into groups, let q_z^i denote the z-th processor of the i-th group.

Procedure FindLMSubset$(\Delta_i, k, w, G_i, g_i^*)$

Input: Upper hulls $\Delta_i, 1 \le i \le p^k$; represented each in p^w consecutively numbered processors $q_z^i, 1 \le z \le p^w$, and a set $G_i \subseteq \Delta_i$.
Output: The point $g_i^* = lm(G_i)$. There will be a copy of g_i^* in each $q_z^i, 1 \le z \le p^w$.

1. Gather G_i in processor q_1^i, for all i.
2. Each processor q_1^i sends its G_i to all processors q_z^j, $j > i, 1 \le z \le p^w$. Each processor q_z^i, for all i, z, receives $\mathcal{G}^i = \bigcup G_j, \forall j < i$.
3. Each processor q_z^i, for all i, z, sequentially computes $Next_{\Delta_i}(g)$ for every $g \in \mathcal{G}^i$, using procedure **QueryFindNext**.
4. Each processor q_z^i, for all i, z, sends back to all processors q_1^j, $j < i$, the computed $Next_{\Delta_i}(g)$, $\forall g \in G_j$. Each processor q_1^i, for all i, receives for each $g \in G_i$ the computed $Next_{\Delta_j}(g)$, $\forall j > i$.
5. Each processor q_1^i, for all i, computes for each $g \in G_i$ the line segment with the largest slope among $\overline{gsuc_{G_i}(g)}$ and $\overline{gNext_{\Delta_j}(g)}$, $j > i$, finding $Next_{G_i \cup \Delta_j, j > i}(g)$. Then, it computes $g_i^* = lm(G_i)$.
6. Each processor q_1^i, for all i, broadcasts g_i^* to $q_z^i, 1 \le z \le p^w$.

Lemma 2 *Procedure FindLMSubset computes $g^* = lm(G_i)$ in a constant number of communication rounds. Furthermore, it requires local memory space $\frac{n}{p} \ge p^k|G_i|$.*

Proof: The correctness of FindLMSubset stems from Definitions 3 and 4, and Procedure QueryFindNext. Its space requirements are: $|G_i|$ at Step 1, $p^k|G_i|$ at Step 2, and p^k at Step 4, amounting to a total of $O(p^k|G_i|)$ space. All sequential operations are in $O(p^k|G_i| \log n)$ time and only four communication rounds are needed. □

3.4 Merge algorithm for the case $n/p \ge p^2$ (i.e., $\epsilon = 2$)

In this section, we describe an algorithm that merges p upper hulls, stored one per processor on a p processor CGM, into a single upper hull using a constant number of global communication rounds. This algorithm requires that $\frac{n}{p}$ be greater than or equal to p^2 and thus exhibits limited scalability. This limitation on the algorithm scalability will be lifted in the next section.

In order to find the upper common tangent between an upper hull X_i and an upper hull X_j (to its right) the algorithm computes the *Next* function, not for the whole set X_i but for a subset of p equally spaced points from X_i. We call this subset of equally spaced points a *splitter* of X_i.

This approach based on splitters greatly reduces the amount of data that must be communicated between processors without greatly increasing the number of global communication rounds that are required. Indeed, there are only five communication rounds, performed in three different phases, namely:

Phase 1: every processor sends a regular sample (the splitters) of size at most p (less than p if the local convex hull is of size smaller than p) of its data to all larger numbered processors. The supporting tangents are then computed and sent back.

Phase 2: every processor sends the data laying between two splitters to all larger numbered processors. The supporting tangents are then computed and sent back.

Phase 3: every processor send its $Next_S$ to all larger numbered processors. They can then compute their own points in the final upper hull.

Algorithm MergeHulls1($X_i(1 \leq i \leq p), S, n, p, UH$)

Input: The set of p upper hulls X_i consisting of a total of at most n points from S, where X_i is stored on processor q_i, $1 \leq i \leq p$.
Output: A distributed representation of the upper hull of S. All points on the upper hull are identified and labeled from left to right.

1. Each processor q_i sequentially identifies a splitter set G_i composed of p evenly spaced points from X_i.
2. The processors find in parallel $g_i^* = lm(G_i)$. This is done via a call to Procedure **FindLMSubset**.
3. Each processor q_i computes its own R_i^- and R_i^+ sets according to Definition 6, and the set R_i of points which are above the line $(g_i^* Next_{G_i \cup X_j, j>i}(g_i^*))$, according to Lemma 1.
4. The processors find in parallel $x_i^* = lm(R_i \cup g_i^*)$, using Procedure **FindLMSubset**. Note that by definition $Next_S(x_i^*) \notin X_i$.
5. Each processor q_i broadcasts its $Next_S(x_i^*)$ to all q_j, $j > i$, and computes its own S_i' according to Definition 5.

Lemma 3 *Algorithm MergeHulls1 computes* $UH(S)$ *in* $O(\frac{n \log n}{p} + T_s(n,p))$ *time. It requires* $\frac{n}{p} \geq p^2$ *local memory space and a constant number of communication rounds.*

Proof: By Fact 1, the sets S_i' computed at Step 5 are a distributed representation of UH(S). Algorithm MergeHulls1 calls Procedure FindLMSubset twice. At Step 2, the parameters can be set as $\Delta_i = X_i$, $k = 1$, $w = 0$ and $G_i = G_i$, implying that $|G_i| = p$. At Step 4 they are the same again, but for $G_i = R_i \bigcup g_i^*$. Therefore, $|G_i| = |R_i \bigcup g_i^*| = \frac{|X_i|}{p} \leq \frac{n}{p^2}$. Thus, the required local memory space is $\frac{n}{p} \geq p^2$, by Lemma 2. The same lemma guarantees that the sequential operations in MergeHulls1 take $O(p^2 \log n) = O(\frac{n}{p} \log n)$ time and that only five communication rounds are used. $\square$

3.5 General case $n/p \geq p^\epsilon$

In this section, we describe an algorithm, MergeHulls2, that merges p upper hulls, stored one per processor on a p processor CGM, into a single upper hull using a constant number of global communication rounds. Unlike the algorithm MergeHulls1, this algorithm requires only $n/p \geq p^\epsilon$ memory space per processor, $0 < \epsilon \leq 1$. For instance and the sake of clarity, imagine that $\epsilon = 1$. The algorithm would be as follows.

1. In the first phase, the algorithm MergeHulls1 is used to find the upper hull of groups of $\sqrt{p}$ processors, in five communication rounds, with $|G_i|$ equal to $\sqrt{p}$. Note that the space required in this phase is $O(p)$.

2. The second phase merges these $\sqrt{p}$ upper hulls of size at most $\frac{n\sqrt{p}}{p}$ each, instead of the p initial ones. Thus, with $|G_i| = \sqrt{p}$ again, Step 2 requires only p space. However, we should be careful, because the size of each set R_i is, in the worst case, $\frac{n\sqrt{p}}{p|G_i|}$, implying that Step 4 requires up to $\frac{n\sqrt{p}}{p}$ space, which is too much. Thus, a further reduction of their sizes is needed. This is accomplished through the simple observation that the sets R_i are upper hulls themselves, and we can recursively apply to the R_i, for all i, the same method used in MergeHulls1 to find $x_i^* = lm(R_i \cup g_i^*)$. Only seven communication rounds are required in this phase.

Applying the same idea further we get a generalization to the case where $\frac{n}{p} \geq p^\epsilon$. The algorithms are described and analysed in the following. Recall that the processors are divided into groups, and q_z^i denote the z-th processor of the i-th group.

Procedure BuildHulls$(\Phi_i, \Psi_j, p, k, w, \epsilon)$

Input: Upper hulls $\Phi_i, 1 \leq i \leq \frac{p}{p^w}$; represented each in p^w consecutively numbered processors $q_z^i, 1 \leq z \leq p^w$. The parameter ϵ reflects the size of the local memory of each of the p processors.

Output: Upper hulls $\Psi_j = UH(\bigcup_{i=(j-1)p^k+1}^{jp^k} \Phi_i)$, for $1 \leq j \leq \frac{p}{p^{w+k}}$.

1. $G_i \longleftarrow \Phi_i$.
2. While $|G_i|p^k > p^\epsilon$ do
   (a) Each group of processors q_z^i, $1 \leq z \leq p^w$, identifies a splitter set G_i' composed of $p^{\epsilon/2}$ evenly spaced points from G_i.
   (b) FindLMSubset$(\Phi_i, k, w, G_i, g_i^*)$.
   (c) If $\overline{g_i^* suc_{\Phi_i}(g_i^*)}$ is above $\overline{g_i^* \text{Next}_{\Phi_j, j>i}(g_i^*)}$ then R_i is composed of all points of Φ_i between g_i^* and $\text{suc}_{G_i}(g_i^*)$; else R_i is composed of all points of Φ_i between $\text{pred}_{G_i}(g_i^*)$ and g_i^*.
   (d) Let $G_i \longleftarrow R_i \cup \{g_i^*\}$.
3. FindLMSubset$(\Phi_i, k, w, G_i, x_i^*)$.
4. Each processor q_1^i broadcasts its x_i^* to all q_j^h, $h = (i\text{mod}(p^k+1))p^k+1, \ldots, (i\text{mod}(p^k+1))p^k+p^k$ and $1 \leq j \leq p^w$.
5. Each processor computes its own S_i' according to Definition 5.

Lemma 4 *Procedure BuildHulls computes* $\Psi_j = UH(\bigcup_{i=(j-1)p^k+1}^{jp^k} \Phi_i)$, *for* $1 \leq j \leq \frac{p}{p^{w+k}}$ *in* $O(\frac{n \log n}{p} + T_s(n,p))$ *time. Let* $\alpha = \max\{(k+\epsilon/2), \epsilon\}$. *It requires* $\frac{n}{p} \geq p^\alpha$ *local memory space and a constant number of communication rounds.*

Proof: The memory requirements of Procedure BuildHulls are as follows. At Step 2(b), $\frac{n}{p} \geq p^{k+\epsilon/2}$. At Step 3, $\frac{n}{p} \geq p^\epsilon$. And at Step 4, $\frac{n}{p} \geq p^k$. The computation takes $O(\frac{n}{p}\log n + p^k|G_i|) = O(\frac{n}{p}\log n)$ time. With respect to the number of communication rounds, notice that at Step 2 there are at most $p^{\alpha+w}$ points in Φ_i. Each passage at Step 2(a) reduces this size by a factor of $p^{\epsilon/2}$. Thus, there are $\frac{2(\alpha+w)}{\epsilon}$ phases, where a phase is a call to Procedure FindLMSubset. □

Algorithm MergeHulls2$(X_i, S, n, p, UH(S), \epsilon)$

Input: The set of p upper hulls X_i consisting of a total of at most n points from S, where X_i is stored on processor p_i, $1 \leq i \leq p$. The parameter ϵ reflects the size of the local memories.
Output: A distributed representation of the upper hull of S. All points on the upper hull are identified and labeled from left to right.

1. Let $k = \frac{\epsilon}{2}$, $w = 0$, and $X_i^0 = X_i$.
2. Do
   (a) BuildHulls$(X_i^w, X_j^{w+k}, p, k, w, \epsilon)$.
   (b) $w \leftarrow w + k$.

   Until $w \geq 1$.

Theorem 1 *Algorithm MergeHulls2 computes $UH(S)$ in $O(\frac{n \log n}{p} + T_s(n,p))$ time. It requires $\frac{n}{p} \geq p^\epsilon$ local memory space and a constant number of communication rounds.*

Proof: Lemma 4 implies that $\frac{n}{p} \geq p^{k+\epsilon/2}$, and so $\frac{n}{p} \geq p^\epsilon$. With respect to the number of communication rounds, Procedure BuildHulls is called $\frac{2}{\epsilon}$ times. By Lemma 4, there are $\frac{2(\epsilon+w)}{\epsilon}$ phases in each call. Hence, since in the t-*th* passage at Step 2(a) it holds that $w = (t-1)\frac{\epsilon}{2}$, then there is a total of $\Sigma_{t=1}^{2/\epsilon} t = O((\frac{2}{\epsilon})^2)$ communication rounds. Finally, this implies that local computation takes $O(\frac{1}{\epsilon^2}\frac{n}{p}\log n)$ time. □

Corollary 1 *The convex hull of n planar points can be computed on a CGM(n,p) in time $O(\frac{T_{sequential}}{p} + T_s(n,p))$, where $T_s(n,p)$ refers to the time of a global sort of n data on a p processor machine. Furthermore, they involve only a constant number of global communication rounds and $n/p \geq p^\epsilon$, for an arbitrarily small but constant $\epsilon > 0$.*

4 Triangulation of a Point Set

In this section we describe how the same ideas developed in the previous section can be used to find a triangulation of a planar point set. The algorithm is based on geometric observations originally used in a PRAM algorithm [37]. We extend their basic technique in order to ensure that the resulting algorithm is both scalable over a large range of the ratio n/p, and uses only a constant number of global operations rounds. Note that the triangulation yielded by the algorithm Triangulate presented below is not the same as the one obtained in [37].

A *funnel polygon* consists of two x-monotone chains and a top and a bottom line segment (see Figure 6). Given p x-disjoint convex hulls, X_i $1 \leq i \leq p$, and a set of upper and lower common tangent lines the regions between the hulls form a set of funnel polygons if the horizontal extreme points of consecutive upper hulls X_i are connected (see Figure 7). Funnel polygons can easily be triangulated as will be shown in the following algorithm. For details on funnel polygons in the context of triangulations, we refer the interested reader to [37].

Triangulate (S)

Input: Each processor stores a set of $\frac{n}{p}$ points drawn arbitrarily from S.
Output: A distributed representation of a triangulation of S. All segment lines of the triangulation are identified and labeled.

1. Globally sort the points in S by x-coordinate. Let S_i denote the set of $\frac{n}{p}$ sorted points now stored on processor i.
2. Independently and in parallel, each processor i computes the convex hull of the set S_i, and the triangulation of its convex hull.
3. Compute upper and lower common tangent lines between every pair of convex hulls, and horizontal extreme points of consecutive convex hulls.
4. Label the upper funnel polygons.
5. Identify each point to its upper funnel polygon.
6. Triangulate the upper funnel polygons.
7. Repeat Steps 4-6 for the lower funnel polygons.

In Step 1 the points of S are globally sorted and the local convex hulls are computed in Step 2. Step 3 starts by connecting the horizontal extreme points of consecutive upper hulls X_i (see Figure 7). Then, $x_i^* = lm(X_i)$ and $Next_{X_j, j>i}(x_i^*)$ are computed as in Section 3.5. Finally, an all-to-all broadcast is performed so that each processor i knows $x_j^* = lm(X_j)$ and $Next_{X_z, z>j}(x_j^*)$, for all $i \geq j$. Clearly, the time complexity of this Step is dominated by the computation of x_i^* and $Next(x_i^*)$, that can be implemented through the procedures FindLMSubset and FindLMSubHull described in the Section 3.5.

Steps 4 and 5 are locally performed on each processor, using the information received at the end of Step 2. Note that, each funnel polygon is delimited by the segment line $\overline{x_i^* Next(x_i^*)}$, and labeled F_i (see Figure 7). Given that each processor stores a table with all common tangent lines, they can identify for each point of their hull the funnel polygon they are part of by a simple sequential scan of their hull points.

Step 6 is the most technical one. Let a_i and b_j denote the points in the left and right chain of a funnel polygon, respectively (see Figure 6). Form records containing, for all points a_i on the left chain, the slope from a point a_i to its successor a_{i+1} in the chain. Also, form records containing, for all points b_j on the right chain, the absolute value of the slope from b_j to its successor b_{j+1} in the chain. Note that if we sort the a's and b's together by first key: funnel polygon label; second key: slope records; third key: left chain first, right chain next; then we get a sequence of the form $\{\ldots a_k a_{k+1} b_h a_{k+2} b_{h+1} b_{h+2} b_{h+3} b_{h+4} a_{k+3} \ldots\}$. The set of line segments that forms a triangulation of the funnel polygon can be easily constructed by forming the set $\{(b_i a_j) \cup (a_r b_s)\}$, where a_j is the first a to the left of b_i and b_s is the first b to the left of a_r.

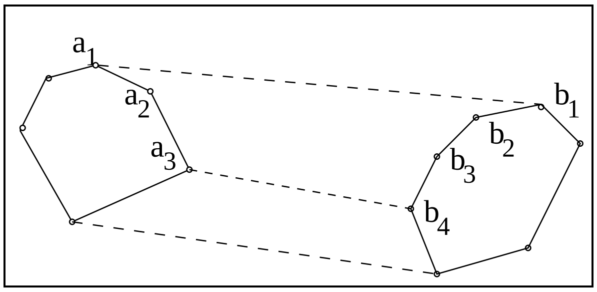


Figure 6: Two funnel polygons.

In order to implement this step, we need only a global sort operation on the a's and b's, followed by a segmented broadcast, where each a is broadcast to every element until the next a to its right in the sorted list, and each b is broadcast to every element until the next b to its right in the sorted list. The line segments composing the triangulation can thus be easily constructed.

Only global sort and global communication operations are used in addition to the call to Find-LMSubset and FindLMSubHull. Therefore, procedure Triangulation above builds a triangulation of a point set in $\mathcal{R}^2$ on a $CGM(n,p)$ in time $O(\frac{n \log n}{p} + T_s(n,p))$, where $T_s(n,p)$ refers to the time of a global sort of n data on a p processor machine. Furthermore, it only requires $n/p \geq p$ local memory (the all-to-all broadcast used in Step 3), and a constant number of global communication and sort rounds. In order to reduce the local memory requirements to $n/p \geq p^{\epsilon}$, $0 < \epsilon < 1$, it suffices to use the technique introduced in Section 3.5, where groups of $p^{\epsilon - \alpha}$ processors were formed, $0 < \alpha < \epsilon$, and this recursively for $O((\frac{1}{\epsilon})^2)$ communication rounds.

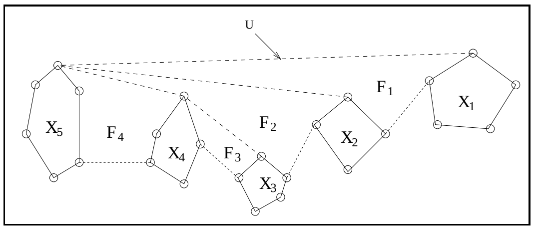


Figure 7: Note that the region between the hulls can be seen as a set of funnel polygons.

5 Experimental Results

To show the practical relevance and the scalability of our upper hull *CGM* algorithm, we have implemented its first version, with $n/p \geq p^2$, on a 256-processors T3E-CRAY computer [3], where 64 processing elements have been dedicated to our experiments in a single user mode. PVM routines have been used for the communications[1].

Most of our experimental analysis were conducted with sorted inputs, so that we could focus on the behavior of the proposed algorithm. Nevertheless, we also provide some experiments with unsorted data for readers who would want to compare our algorithm to others that would not use sorting.

The sequential convex hull algorithm used in our coarse grained algorithm is an adaptation to sorted inputs, of Clarkson's code [6]. This code is an implementation of a slight modification of Procedure 8.2 from [18]. Its time complexity is $O(n \log n)$ for an arbitrary set of points. However in the case of pre-sorted data, the complexity becomes linear. Each point is implemented as a pair of two single precision floating point numbers (coordinates of the point).

We remark that if S is a set of uniformly distributed points in the plane, the upper hull of S consists in average of just a few points of S [17]. This data reduction can considerably speedup our parallel merging phase, since the amount of data remaining in this phase is, in average, significantly smaller than the size of the initial set S. Therefore, in the remainder we report the behavior of our algorithm with respect to the worst case input (i.e., all the points of the initial set belong to the upper hull), intermediate data sets (i.e., data sets with an important number of points on the upper hull), and random data sets.

Recall that in the merging phase of the version where $n/p \geq p^2$, there are three main phases:

Phase 1: every processor sends a regular sample (the splitters) of size at most p (less than p if the

[1]The codes are available upon request.

local convex hull is of size smaller than p) of its data to all larger numbered processors. The supporting tangents are then computed and sent back.

Phase 2: every processor sends the data laying between two splitters to all larger numbered processors. The supporting tangents are then computed and sent back.

Phase 3: every processor send its $Next_S$ to all larger numbered processors. They can then compute their own points in the final upper hull.

Processor p is the one receiving the largest amount of data i.e., $(p-1)p$ in Phase 1, $(p-1)\frac{n}{p^2}$ in Phase 2 and $p-1$ in Phase 3. When sending back the results, the same amount of data moves again and processor i plays the role of processor $p-1-i$. Thus the total communication time is

$$T_{com} = O(p^2 + \frac{n}{p}). \tag{1}$$

With respect to the local computation, every processor computes with its n/p local data and all the data it receives from the others. Hence, the local computation time is

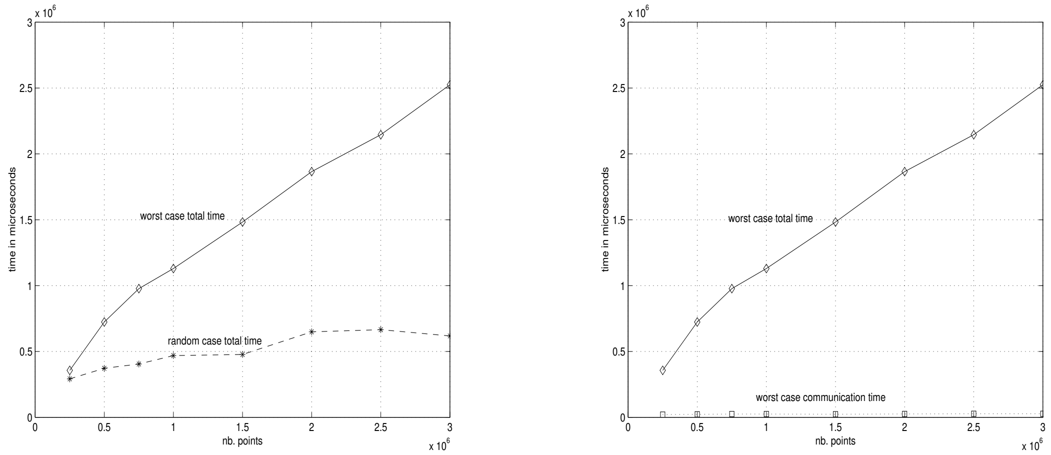
$$T_{loc} = O(\frac{n}{p} + p^2 \log \frac{n}{p} + \frac{n}{p} \log \frac{n}{p}), \tag{2}$$

and the total time is

$$T_{tot} = T_{loc} + T_{com}. \tag{3}$$

In the following, we present our experimental results. The running time for the random points set is the average of the running times on 10 different random points sets. We notice that conclusions are difficult to draw in the case random data sets because of the unpredictable way in which data reduction occurs. Therefore, we also experimented our algorithm with a worst case input, i.e., an instance where all the points belong to the final upper hull. For that, we forced all the points to lay on a straight line.

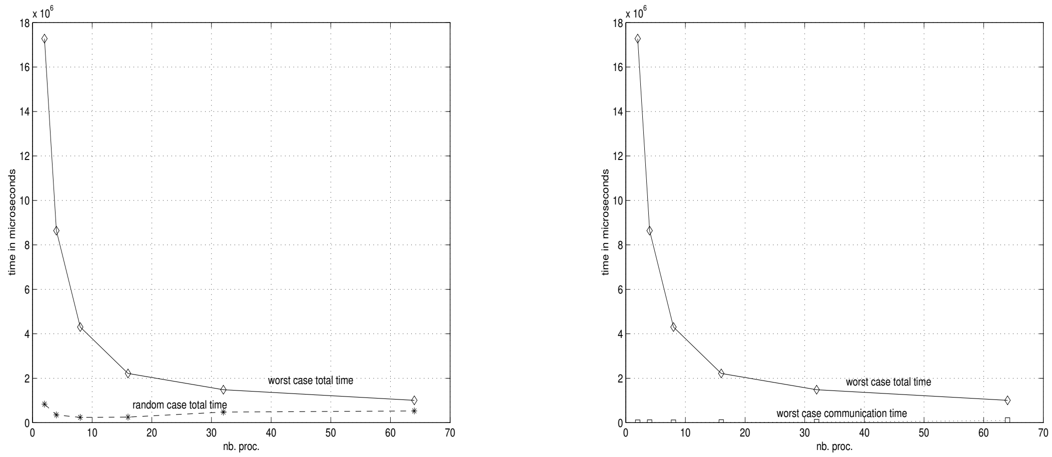
Figure 8 plots data for an increasing number of points when the number of processors is fixed. Figure 8(a) shows both the random and the worst case. We can notice the divergence of the two curves as n grows. According to Equations 2 and 1, the local computation time grows proportional to $n \log n$ and the communication time grows proportional to n. As mentioned above, for random points sets the data reduction plays an important role, and the computation time described in Figure 8 accounts mainly for the computation of the initial upper hull.



(a) (b)

Figure 8: Fixed p (p = 32).

Figure 8(b) shows plotted data for the worst case. The local time grows proportional to $n \log n$ (Equation 2). The total communication time grows very smoothly.

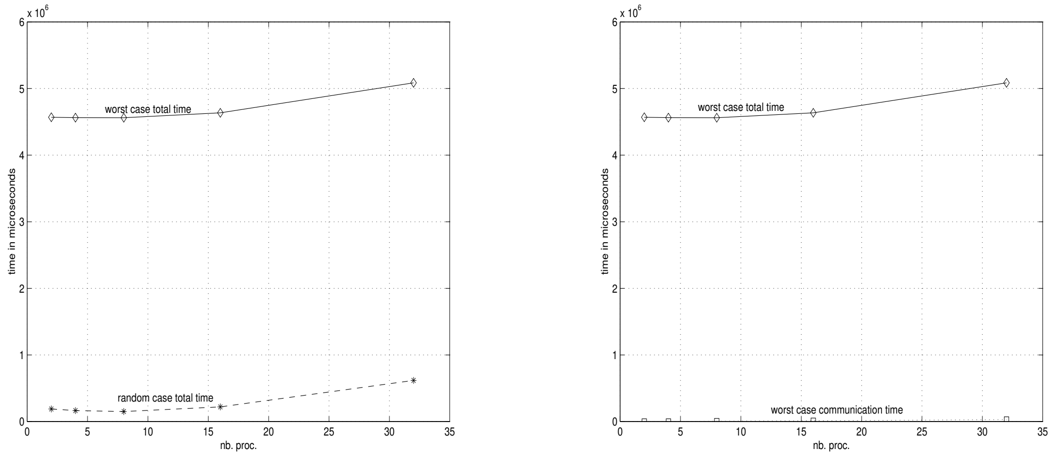


(a) (b)

Figure 9: Fixed n (n = 1,500,000).

Figure 9 plots data for an increasing number of processors when the number of points is fixed. Figure 9(a) shows speedups for both random and worst cases. For the random case, a coherent behaviour is observed when p is small. However, when p increases, so does the total number of messages and the number of communication startups. Hence, the good behaviour can be maintained until the communication time is no longer negligeable compared to the local computation time.

In the worst case (Figure 9(b)), the observed speedup is better because the amount of local computation (computation of the local hull and the supporting tangents) is large enough when compared to communication, as shown in Equations 1 and 2.



(a) (b)

Figure 10: Fixed ratio n/p (n/p = 200,000).

Figure 10 shows the behaviour of the algorithm when there is a fixed amount of data per processor and the number of processors increases, $\frac{n}{p} = 200,000$. In both random and worst case, the curves have the same shape. For small values of p we have a good scalability. But as p grows, the p^2 factor starts to appear (see Equation 3).

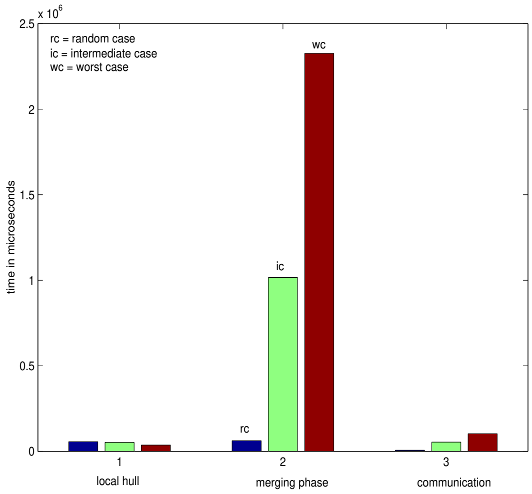


Figure 11: Breakdown 1 (p = 8, n = 800,000).

Figure 11 gives the breakdown of our algorithm for the random, the intermediate and the worst case data sets for $p = 8$ and $n = 800,000$. The first group of vertical bars corresponds to the computation of the local hull; the second group to the total computation in the merging phase; and the third group to the total communication time. We remark that the total computation time is essentially composed of the computation time in the merging phase.

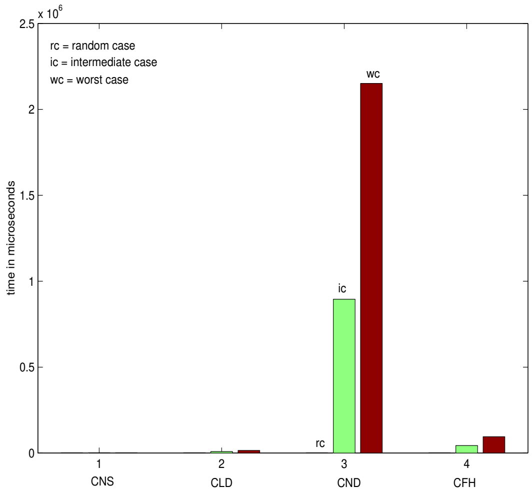


Figure 12: Breakdown 2 (p = 8, n = 800,000).

Figure 12 shows the breakdown of the computation in the merging phase. The first group of vertical bars corresponds to the computation time of the $Next_S$ of splitters. The second group to the computation of the leftmost points and the data (between two splitters) to be sent during the second communication phase. The third group, is the computation time of the $Next_S$ of those data sent during the second communication phase. The last group corresponds to the computation time of the points on the final hull. Hence, the computation time during the merging phase is almost totally due to the computation of the $Next_S$ of data between two splitters. This computation phase seems to be the bottleneck of our algorithm and, in case it comes of use in real application, further work should try to improve this phase.

Figure 13 shows the evolution of run time when we pass progressively from a random points set to the worst case points set. The curve was obtained with $n = 800,000$ and $p = 4$. The time corresponding to a totally random input set is shown by the dashed line and, as we can see, the average number of points on the upper hull for this input set is roughly 25 percent of the total number of points.

Finally, Figure 14 presents results of experiments on building upper hulls of unsorted data. A sample sort algorithm [20] has been implemented. We have also implemented a parallel prefix algorithm at the end of the sorting procedure, to make sure that every processor holds n/p points.

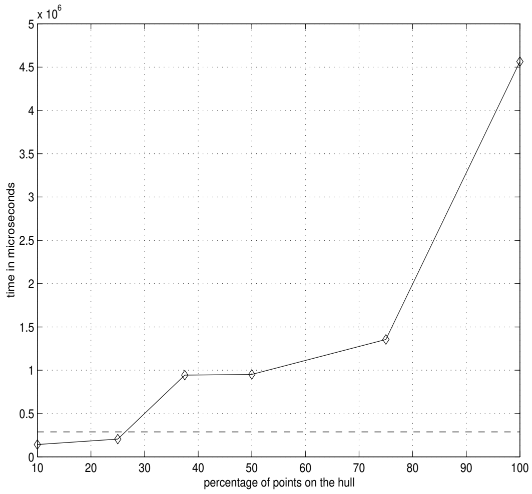


Figure 13: Fixed p and n (p = 4, n = 800,000).

These experiments are reported only for the sake of completeness, in order to allow for comparison with other parallel convex hull algorithms that do not use sorting.

6 Conclusion

In this paper we described scalable deterministic parallel algorithms for building the Convex Hull and a Triangulation of a planar point set using the coarse grained multicomputer model. Our algorithms require time $O(\frac{n \log n}{p} + T_s(n,p))$, where $T_s(n,p)$ refers to the time of a global sort of n data on a p processor machine. Furthermore, they involve only a constant number of global communication rounds and scale over a large range of values of n and p, assuming only that $n \geq p^{1+\epsilon}$ $(\epsilon > 0)$. The algorithms proposed in this paper are based on a variety of techniques arising from more theoretical models for parallel computing, such as the PRAM and the fine-grained hypercube. It would be interesting to identify those parallel algorithm design techniques for theoretical models that can be extended to the very practical *coarse grained multicomputer* model.

Acknowledgments

The authors are grateful to the anonymous referees, whose insightful remarks helped improve the manuscript a great deal. We also thank Serge Miguet for helping us with processing hours on the T3E.

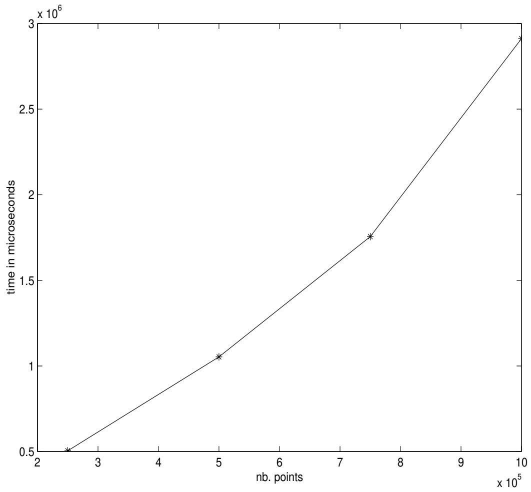


Figure 14: Fixed p (p = 4).

References

[1] S.G. Akl, A Constant-Time Parallel Algorithm for Computing Convex Hulls, *BIT*, Vol. 22, pages 130–134, 1982.

[2] S.G. Akl and K.A. Lyons, *Parallel Computational Geometry*, Prentice-Hall, New York, 1993.

[3] E. Anderson, J. Brooks, C. Grassl and S. Scott, Performance of the CRAY T3E Multiprocessor. *Proc. Supercomputing 97*, 1997. URL: http://www.cray.com/products/systems/crayt3e/1200/performance.html.

[4] M. J. Atallah and J.-J. Tsay, On the parallel-decomposability of geometric problems. *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 104–113, 1989.

[5] K.E. Batcher, Sorting networks and their applications. *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[6] J. L. Bentley, K. L. Clarkson, and D. B. Levine, Fast linear expected-time algorithms for computing maxima and convex hulls, In *Algorithmica*, pages 168–183, 1993.

[7] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[8] B. Chazelle, Computational Geometry on Systolic Chip, *IEEE Trans. on Computers*, Vol. C-33, No. 9, pages 774-785, 1984.

[9] A.L. Chow, Parallel Algorithm for Determining Convex Hull of Sets of Points in Two Dimension, *Proc. of the Nineteenth Annual Allerton Conf. on Communication, Control and Computing*, pages 214-223, 1981.

[10] Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subrarnonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.

[11] S.E. Hambrusch and A.A. Khokhar. C^3: An architecture-independent model for coarse-grained parallel machines. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, October, Dallas, USA*, 1994.

[12] R. Cypher and C. G. Plaxton, Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *ACM Symposium on Theory of Computing*, 193–203. ACM, 1990.

[13] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Khokhar, A randomized parallel 3D convex hull algorithm for coarse grained multicomputers, *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, (1995), 27-33.

[14] F. Dehne, A. Fabri and C. Kenyon Scalable and architecture independent parallel geometric algorithms with high probability optimal time. *Proceedings fo the 6th IEEE SPDP, IEEE Press*, 586–593, 1994.

[15] F. Dehne, A. Fabri and A. Rau-Chaplin, Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. *ACM Symposium on Computational Geometry*, 1993.

[16] X. Deng and N. Gu. Good algorithm design style for multiprocessors. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, October, Dallas, USA*, 1994.

[17] L. Devroye and G.T. Toussaint A note on linear expected time algorithms for finding convex hulls, In *Computing*, vol. 26, pp. 361-366, 1981.

[18] H. Edelsbrunner Algorithms in combinatorial geometry, In *EATCS Monographs on Theoretical Computer Science*, W. Brauer, G. Rozenberg and A. Salomaa (Editors), Springer-Verlag, Berlin, 1987.

[19] A. Ferreira, A. Rau-Chaplin, and S. Ubéda. Scalable 2d convex hull and triangulation for coarse grained multicomputers. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing*, San Antonio, USA, 1995.

[20] A.V Gerbessiotis and L.G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, pages 251–267, 1994.

[21] T. Goodrich. Communication-efficient parallel sorting. In *Proc. of the* 28^{th} *annual ACM Symposium on Theory of Computing (STOC), May 22-24, Philadephia, USA*, 1996.

[22] M.T. Goodrich, J.J. Tsay, D.E. Vengroff, and J.S. Vitter, External-memory computational geometry. *Foundations of Computer Science, 1993.*

[23] *Grand Challenges: High Performance Computing and Communications*, The FY 1992 U.S. Research and Development Program. A Report by the Committee on Physical, Mathematical, and Engineering Sciences. Federal Councel for Science, Engineering, and Technology. To Supplement the U.S. President's Fiscal Year 1992 Budget.

[24] R. I. Greenberg and C. E. Leiserson, Randomized Routing on Fat-trees. *Advances in Computing Research*, 5:345–374, 1989.

[25] H. Li and K.C. Sevick. Parallel sorting by overpartitioning. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, pages 46–56, 1994.

[26] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[27] P.D. MacKenzie and Q.S. Stout, Asymptotically Efficient Hypercube Algorithmes for Cmputational Geometrie, *Proc. of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 8-11, 1980.

[28] J.M. Marberg and E. Gafni, Sorting in constant number of row and column phases on a mesh. *Proc. Allerton Conference on Communication, Control and Computing*, 1986, pp. 603-612.

[29] E. Merks, An Optimal Parallel Algorithm for Triangulation of a Set of Points in the Plane, *International Journal of Parallel Programming*, Vol. 15, No. 5, pages 399-411, 1986.

[30] R. Miller and Q.F. Sout, Computational Geometry on a Mesh-Connected Computer (Preliminary version), *Proc. of the 1984 International Conference on Parallel Processing*, pages 236-271, 1984.

[31] R. Miller and Q. Stout, Efficent Parallel Convex Hull Algorithms. *IEEE Transcations on Computers*, 37:12:1605–1618, 1988.

[32] D. Nath, S.N. Maheshwari and P.C. Bhatt, *Parallel Algorithm for the Convex Hull Problem in Two Dimensions*, Technical Report EE 8005, Dept. of Elect. Eng., Indian Inst. of Tech, New Delhi, 1980.

[33] F. P. Preparata and M. I. Shamos, *Computational Geometry: an Introduction*. Springer-Verlag, New York, NY, 1985.

[34] J. H. Reif and L. G. Valiant, A Logarithmic Time Sort for Linear Size Networks. *J. ACM*, Vol.34, 1:60–76, 1987.

[35] L. G. Valiant. A bridging model for parallel computation. *Communication of ACM*, 38(8):103–111, 1990.

[36] L.G. Valiant. *General purpose parallel architecture*. J. van Leewen, North Holland, 1990.

[37] C. Wang and Y. Tsin, An O(log n) Time Parallel Algorithm for Triangulating A Set Of Points In The Plane. *Information Processing Letters*, 25:55–60, 1987.