Scalable 2d Convex Hull and Trianglulation Algorithms for Coarse Grained Multicomputers

Afonso Ferreira*

LIP ENS-Lyon
46, allée d'Italie
69364 Lyon Cedex 7
France

Andrew Rau-Chaplin†

Technical University of Nova Scotia
P.O. Box 1000,
Halifax, Nova Scotia
Canada B3J 2X4

Stéphane Uéda‡

LIP ENS-Lyon
46, allée d'Italie
69364 Lyon Cedex
France

Absract

In this paper we describe scalable parallel algorithms for building the Convex Hull and a Triangulation of a given point set in $\mathcal{R}^2$. 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 (e.g. mesh, hypercube, omega). They 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 $T_s(n,p)$, for instance when randomized sorting algorithms are used, or for interconnection networks like the mesh for which optimal sorting algorithms exist.*

1 Introduction

Most existing multicomputers (e.g. the Intel Paragon, Cray T3D, Meiko CS-2 and IBM SP2) consist of a set of p *state-of-the-art* processors, each with considerable local memory, connected to some interconnection network (e.g. mesh, hypercube, omega). These machines are usually *coarse grained*, i.e. the size of each local memory is "considerably larger" than $O(1)$. Despite this fact, 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 p processors (e.g., a PRAM, mesh, or hypercube multiprocessor) such that $\frac{n}{p} = O(1)$. However, as noted in [9], to be relevant in practice such algorithms must be *scalable*, that is, they must be applicable and efficient for a wide range of ratios $\frac{n}{p}$.

In this paper we describe scalable parallel algorithms for some fundamental geometric problems. Among the first scalable coarse grained algorithms for solving geometric problems on distributed memory machines were described in [6]. The majority of these algorithms were optimal on architectures for which optimal sorting algorithms were known and required at most sort time otherwise. Another important feature of these algorithms was that they involved only a constant number of global communication rounds, which had the effect of making them very fast in practice. Problems studied in [6] included computing the area of the union of rectangles, $3D$-maxima, $2D$-nearest neighbors of a point set, and lower envelope of non-intersecting line segments in the plane. All of these algorithms used a spatial partioning technique in which a constant number of partitioning schemes of the global problem (on the entire data set of n data items) into p subproblems of size $O(\frac{n}{p})$ were used. Each processor solved (sequentially) a constant number of such $O(\frac{n}{p})$ size subproblems, and a constant number of global routing operations were used to permute the subproblems between the processors. Eventually, by combining the $O(1)$ solutions of its $O(\frac{n}{p})$ size subproblems, each processor determined its $O(\frac{n}{p})$ size portion of the *global* solution.

For most of the problems addressed in [6] the spatial partitioning technique described above were very efficient. One exception was the 2d Convex Hull problem for which the resulting algorithm required $O(\frac{n \log n}{p} + T_s(n,p) + \log p T_s(p,p))$ time and involved $\log p$ global communication rounds. Subsequent work on this problem has focussed on developing randomized algorithms that trade-off guaranteed solutions for efficiency (i.e. a constant number of communication rounds). In [7] Deng and Gu solve the 2d convex hull problem with high probability for any input assuming $\frac{n}{p} \geq p^{2+\epsilon}$ while Dehne, Kenyon and Fabri solve for a given a set of "well distributed inputs" the kd convex

*(ferreira@lip.ens-lyon.fr)

Partially supported by the project Strategeme of the French CNRS and by DRET

†(arc@tuns.ca)

Partially supported by DIMACS, a National Science Foundation center (USA), and the Natural Sciences and Engineering Research Council (Canaca).

‡(ubeda@lip.ens-lyon.fr)

Partially supported by the project PRS (CAPA) of the French CNRS

hull problem, for any constant k, and $\frac{n}{p} \geq p^{\epsilon}$.

In this paper we describe a deterministic scalable algorithm for solving the 2d convex hull problem on the *coarse grained multicomputer* model: p processors with $O(\frac{n}{p}) \gg O(1)$ local memory each, connected to some arbitrary interconnection network. Our algorithm requires 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 involves only a constant number of global communication rounds. 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 in the same space and time.

Since computing either 2d convex hull or triangulation requires time $T_{sequential} = \Theta(n \log n)$ [14] 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 $T_s(n,p)$, for instance when randomized sorting algorithms are used, 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 [11] 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.

The *Coarse Grained Multicomputer* model, or $CGM(n,p)$ for short, is defined in [6]. It consists of a set of p processors with $O(\frac{n}{p})$ local memory each, 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 $O(1)$. Our definition of "considerably larger" (and the one used in [6]) is that $\frac{n}{p} \geq p$. This is clearly true for all currently available coarse grained parallel machines. In the following, all messages are composed of a constant number of words and, for determining time complexities, we will consider both local computation time and inter processor communication time in the standard way.

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 present a new technique for designing efficient scalable parallel geometric algorithms. Our results are independent of the communication network (e.g. mesh, hypercube, fat tree). A particular strength of this approach (which is very different from the one presented in [1, 8]), 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 is performed by calls to a standard highly optimized communication library and very fast in practice (see [6] for evidence of this).

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 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. By means of a merge operation compute for each upper hull X_i, $1 \leq i \leq p$, the upper common tangent line 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 [14]. 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 3.3, 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 3.4, is a more complex merge that uses the first merge as a subprocedure but has a higher degree of scalability in that it requires only that $\frac{n}{p} \geq p$ while still requiring only 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 by Miller and Stout [13].

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* [16].

The remainder of this paper is organized as follows: in the next section we give more details about the coarse grained multicomputer model, $CGM(n,p)$. 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 conclude.

2 The Coarse Grained Model: Definitions and 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. Commonly used interconnection networks for a CGM include the 2D-mesh (e.g. Intel Paragon), hypercube (e.g. Intel iPSC/860) and the fat tree (e.g. Thinking Machines CM-5). For determining time complexities, we limit all messages to a constant number of words and account for both local computation time and inter processor communication time. The term "coarse grained" refers to the fact that the size $O(\frac{n}{p})$ of each local memory will typically be "considerably larger" than $O(1)$. We will assume either $\frac{n}{p} \geq p$ as was assumed in [6].

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 [2] and [12], respectively. Note that for the hypercube an asymptotically better deterministic algorithm exists [4], but it is of more theorical than practical interest. One could also use randomized sorting [15], but in this paper we will focus primarily on deterministic methods. We refer the reader to [2, 3, 10, 11, 12, 15] 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 $p_{j_i+1} \ldots p_{j_{i+1}-1}$. The time for a segmented broadcast operation will be referred to as $T_{sb}(k,p)$.

Segmented gather: In a segmented gather operation, $q \leq p$ processors with ids $j_1 < j_2 < \ldots < j_q$ are selected. Each such processor, p_{j_i}, is sent 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. The time for a segmented gather operation will be referred to as $T_{sg}(k,p)$.

All-to-All broadcast: In an All-to-All broadcast operation, every processor sends one message to all other processors. The time complexity will be denoted as $T_{a2a}(p)$.

Personalized All-to-All broadcast: In a Personalized All-to-All broadcast operation, every processor sends a different message to every other processor. The time complexity will be denoted as $T_{pa2a}(n,p)$.

Partial sum (Scan): Every processor stores one value, and all processors compute the partial sums of these values with respect to some associative operator. The time complexity will be denoted as $T_{ps}(n,p)$.

For any $CGM(n,p)$ with $\frac{n}{p} \geq p$ it is easy to show that these operations are no more complex than sorting plus a linear amount of sequential work [6], i.e., they require $O(\frac{n}{p} + T_s(n,p))$.

3 Merging Convex Hulls in Parallel

In this section we show how to complete the upper hull algorithm given in Section 1 by merging p x-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 1, 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 1).*

Definition 2 *Let $X_i = \{x_1, x_2, \ldots, x_m\}$ be an upper hull. Then, $pred_{X_i}(x_i)$ denotes x_{i-1} and $suc_{X_i}(x_i)$ denotes the point x_{i+1} (see Figure 2).*

Given two upper hulls $UH(S_i)$ and $UH(S_j)$ 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 2).

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 $q' \in S$, q' to the right of p.*

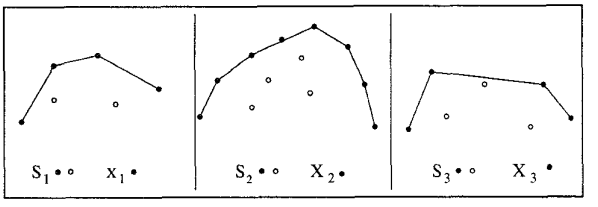

Figure 1: In this example $S = \{S_1, S_2, S_3, S_4\}$ and the points in S_i that are filled in are the elements of the upper hull of S_i, namely X_i.

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 $Next_{Y \cup S_j, j>i}(y^*) \notin S_i$).*

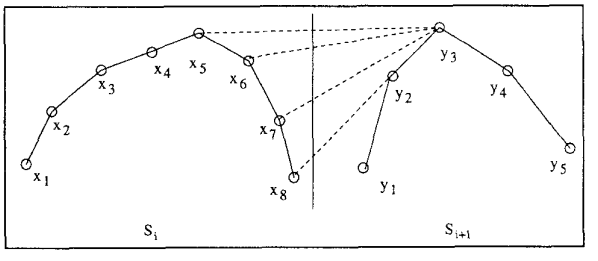

Figure 2: Let $S' = \{S_i, S_{i+1}\}$ then $x_3 = Next_{S'}(x_2)$, $suc(x_j) = x_{j+1}$, $Next_{S'}(x_3) = x_4$, $Next_{S'}(x_4) = x_5$, $Next_{S'}(x_5) = y_3$, $Next_{S'}(x_6) = y_3$, $Next_{S'}(x_7) = y_3$, $Next_{S'}(x_8) = y_2$, and $lm(S_i) = x_5$.

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. Below, we present a sequential algorithm called **QueryFindNext** to search for $q = Next_X(c)$. Figure 3 illustrates one step of this algorithm. This binary search process takes time $O(\log |X|)$.

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).

3.1 Characterization of the upper hull

A classical way of characterizing the upper hull of a point set S, as given in [14], is based on the observation that "A line segment $\overline{ab}$ is an edge of the upper

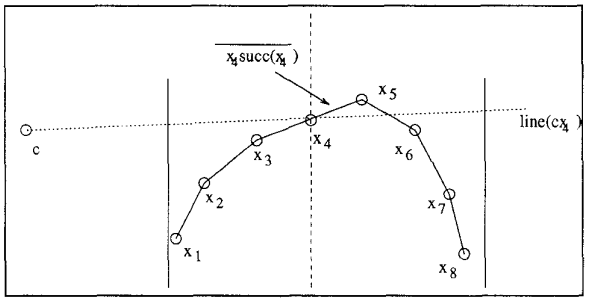

Figure 3: One step in the binary search procedure QueryFindNext. Since the line segment $\overline{x_4 suc(x_4)}$ is above the line (cx_4) the algorithm will recurse on $\{x_4, \ldots, x_8\}$.

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 (a,b)". 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 $\overline{x_i^* Next_{\bigcup X_j, j>i}(x_i^*)}, 1 \leq i < p\}$, where $x_i^* = lm(X_i)$.*

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

Theorem 1 $S' = UH(S)$

Proof: We must consider two cases.

- $UH(S) \subseteq S'$

  Suppose $y \in UH(S)$, $y \notin S'$. Then $\exists i$ such that $\overline{x_i^* Next_{X_j, j>i}(x_i^*)}$ dominates y. Therefore, $y \notin UH(S)$, which is a contradiction.

- $S' \subseteq UH(S)$

  Suppose $y \in S'$, $y \notin UH(S)$. Then $\exists p, q$ with $p \in UH(S), q \in UH(S)$, and $q = Next_S(p)$, such that $\overline{pq}$ dominates y.

  Therefore, both q and p cannot belong to X_i (since $y \in X_i$, meaning that y is not dominated by any line segment with endpoints in S_i). Thus, p and q belong respectively to X_j and X_k, $(j \neq k)$. Hence, since $q = Next_S(p)$, $p \in UH(S)$, and $q \in UH(S)$, then $\exists i$ such that $p = x_i^*$, where $x_i^* = lm(X_i)$. Therefore, y is dominated by a segment $\overline{x_i^* Next_{X_j, j>i}(x_i^*)}$, which is a contradiction.

□

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 4).*

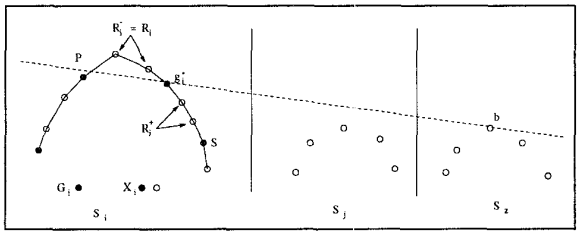

Figure 4: Filled points are element of G_i which is a subset of X_i composed of both of hollow and filled points. Since the R_i^+ doesn't contained any point above the line $(g_i^* b)$ we have $R_i = R_i^-$.

We have then the following lemma.

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 4).*

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 Merge Algorithm for the Case $n/p \geq p^2$

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 algorithms scalability will be addressed 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_j but for a subset of s 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.

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

Input: The set of π upper hulls X_i consisting of a total of at most n points from S, where X_i is stored on processor q_i, $1 \leq i \leq \pi$ and an integer 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. Each processor q_i sequentially identifies a splitter set G_i composed of s 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 x_i^* to all q_j, $j > i$, and computes its own S_i' according to Definition 5. By Theorem 1 this is a distributed representation of $UH(S)$.

Analysis

Steps 2 and 4 of the algorithm above are implemented via the procedure FindLMSubset which is described in detail the next section. The key idea in procedure FindLMSubset is to send the elements of G_i or R_i, in Steps 2 and 4, respectively, to all larger-numbered processors so that processors receiving $G = \bigcup G_j, j < i$ (at Step 2) or $R = \bigcup (R_j \cup g_j^*), j < i$ (at Step 4) can sequentially compute the required points using Procedure QueryFindNext described in Section 3. The algorithm MergeHulls1 requires $s\pi$ space at Step 2, and $\frac{|UH_i| * \pi}{s} \leq \frac{n}{s}$ space at Step 4. Thus, if we set $\pi = s = p$, then this algorithm requires $n/p \geq p^2$ space per processor. With respect to time complexity note that only a constant number of sort and communication rounds are required, plus sequential computations. The time required by the algorithm is therefore $O(\frac{n \log n}{p} + T_s(n,p))$.

3.4 Merge Algorithm for the Case $n/p \geq p$

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 given in the previous section this algorithm requires only $n/p \geq p$ memory space per processor. It is based on algorithm MergeHulls1, except that now two phases are used.

1. In the first phase, the algorithm MergeHulls1 is used to find the upper hull of groups of $\pi = \sqrt{p}$ processors, with s 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 $s = \sqrt{p}$ 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, $\frac{n\sqrt{p}}{ps}$, implying that Step 4 requires up to $\frac{n}{\sqrt{p}}$ 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 the same method udes in MergeHulls1 to find $x_i^* = lm(R_i \cup g_i^*)$.

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

Procedure FindLMSubset(X_i, k, w, G_i, g_i^*)

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

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, $j > i, 1 \leq z \leq w$. 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 $Next_{X_i}(g)$ 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 $Next_{X_i}(g)$, $\forall g \in G_j$. Each processor q_1^i, for all i, receives for each $g \in G_i$ the computed $Next_{X_j}(g)$, $\forall j > i$.
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 $\overline{gNext_{X_j}(g)}$, $j > i$, finding $Next_{G_i \cup X_j, j>i}(g)$. Then, it computes $g_i^* = lm(G_i)$.
6. Broadcast g_i^* to $q_z^i, 1 \leq z \leq w$.

Procedure FindLMSubHull(X_i, k, w, G_i, x_i^*)

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

1. FindLMSubset(X_i, k, w, G_i, g_i^*).
2. If $\overline{g_i^* suc_{X_i}(g_i^*)}$ is above $\overline{g_i^* \text{Next}_{X_j, j>i}(g_i^*)}$ then R_i is composed of all points of X_i between g_i^* and $\text{suc}_{G_i}(g_i^*)$; else R_i is composed of all points of X_i between $\text{pred}_{G_i}(g_i^*)$ and g_i^*.
3. Let $G_i \longleftarrow R_i \cup \{g_i^*\}$.
4. If $|G_i| * k \leq p$
then FindLMSubset(X_i, k, w, G_i, x_i^*)
else FindLMSubHull(X_i, k, w, G_i, x_i^*).

Algorithm MergeHulls2$(X_i (1 \leq i \leq \pi), S, n, \pi, s, UH)$

Input: The set of π upper hulls X_i consisting of a total of at most n points from S, where X_i is stored on processor q_i, $1 \leq i \leq \pi$ and an integer 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. Each processor q_i sequentially identifies a splitter set G_i composed of s evenly spaced points from X_i.
2. Divide the π processors into $\sqrt{\pi}$ groups of $\sqrt{\pi}$ processors each. Do in parallel for each group: FindLMSubHull$(X_i = UH(S_i), k = \sqrt{\pi}, w = 1, G_i, x_i^*)$.
3. Each processor q_i broadcasts its x_i^* to all q_j, $j > i$.
4. Each processor q_j computes its own S'_j according to Definition 5. By Theorem 1 this is a distributed representation of the upper hull of the points in each of the $\sqrt{\pi}$ groups. Let $\Gamma_i, 1 \leq i \leq \sqrt{\pi}$, denote such upper hulls.
5. Each processor q_i sequentially identifies a splitter set G_i composed of s evenly spaced points from Γ_i.
6. Do in parallel: FindLMSubHull$(X_i = \Gamma_i, k = \sqrt{\pi}, w = \sqrt{\pi}, G_i, x_i^*)$.
7. Each processor q_1^i broadcasts its x_i^* to all q_j^h, $h > i$ and $j > 1$.
8. Each processor q_i computes its own S'_i according to Definition 5. By Theorem 1 this is a distributed representation of $UH(S)$.

3.5 Analysis

In the algorithm MergeHulls2, if we let $\pi = p$ and $s = \sqrt{p}$, then Step 2 requires n/p space, as seen in the analysis of the algorithm MergeHulls1. Notice that the space complexity of procedure FindLMSubset is $|G_i| * k$ (at Steps 2 and 4). Thus, Step 7's call to procedure FindLMSubset requires $|G_i| * k = p$ space.

Now, let us analyze the space requirements of Step 7's call to procedure FindLMSubHull. At Step 1 of FindLMSubHull, the call to procedure FindLMSubset requires $|G_i| * k = p$ space. At Step 5, $|G_i| * k$ is greater than p implying that a recursive call takes place in

order to reduce the size of the considered sets. Both new calls to FindLMSubset within the recursive call to FindLMSubHull then require p space. Therefore, Step 7 requires $n/p \geq p$ space, and this is the space requirement for the MergeHulls2 algorithm as a whole.

With respect to time complexity note that only a constant number of sort and communication rounds are required, plus sequential computations. The time required by the algorithm is therefore $O(\frac{n \log n}{p} + T_s(n,p))$.

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 made by Wang and Tsay and used in a PRAM algorithm [16]. 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 [16].

A *funnel polygon* consists of two x-monotone chains and a top and a bottom line segment (see Figure 5). 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 6). Funnel polygons can easily be triangulated as will be shown in the following algorithm. For details on funnel polygons in the contex of triangulations, we refer the interested reader to [16].

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. Let X_i denote the upper hull of the set S_i on processor i.
3. Label the upper funnel polygons.
4. Identify each point to its upper funnel polygon.
5. Triangulate the upper funnel polygons.
6. Repeat Steps 3-5 for the lower funnel polygons.

In Step 1 the points of S are globally sorted. Step 2 starts by connecting the horizontal extreme points of consecutive upper hulls X_i (see Figure 6). Then, $x_i^* = lm(X_i)$ and $Next_{X_j, j>i}(x_i^*)$ are computed as in Section 3.4. 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 3.4.

Steps 3 and 4 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 6). 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 5 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 5). 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.

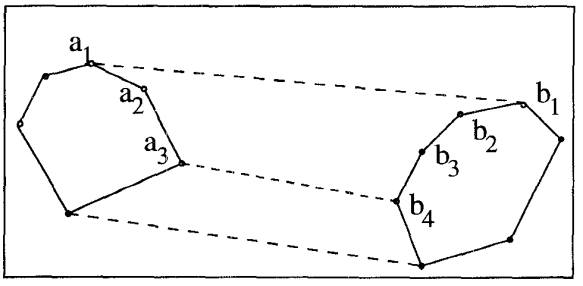

Figure 5: Two funnel polygons.

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 FindLMSubset 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, and a constant number of global communication and sort rounds.

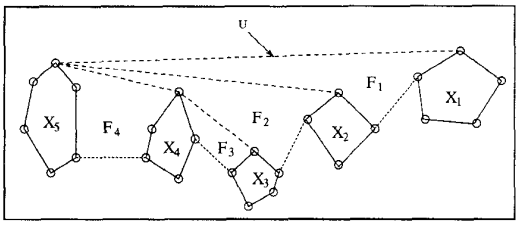

Figure 6: Note that the region between the hulls can be seen as a set of funnel polygons (U is $\overline{lm(X_1)Next_S(lm(X_1))}$).

5 Conclusion

In this paper we described scalable parallel algorithms for building the Convex Hull and a Triangulation of a planar point set for 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, either running 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 $T_s(n,p)$, for instance when randomized sorting algorithms are used, or when $T_s(n,p)$ is optimal. For practical purpose the restriction that $\frac{n}{p} \geq p$ is not onerous, but for theoretical perspective it is interesting to note that by performing $\lceil \log(2/\epsilon) \rceil$ $(0 < \epsilon < 1)$ phases (rather than two phases) in procedure MergeHull2 this restriction can be weakened to $\frac{n}{p} \geq p^{\epsilon}$ (as in [5]) while still maintaining only a constant number of communication rounds.

The algorithms proposed in this paper were 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.

References

[1] 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.

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

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

[4] 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.

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

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

[7] Deng and Gu Good algorithm design style for multiprocessors. *6th IEEE Symposium on Parallel and Distributed Processing*, 1994.

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

[9] *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.

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

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

[12] 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.

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

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

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

[16] 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.