# Parallel Triangulation of a Set of Points for Coarse Grained Multicomputers[1]

Gabriela Hristescu
Rutgers University
Department of Computer Science

[1]Technical Report DCS-TR-313, September 1994

## Abstract

In this study we address the problem of efficient parallel triangulation methods for a finite set of points in the plane. The main goals of the research were to identify scalable algorithms which achieve a significant speedup over the sequential solutions and to implement and evaluate their performance on a parallel machine. Two approaches for parallel triangulation, one of which relies on global sorting, are fully described and implemented on a hypercube. Extensive performance evaluation in expected average and worst cases is reported for both methods.

# Contents


**1 Introduction** **3**

1.1 Problem Description . . . . . 3

1.1.1 Motivation and Significance . . . . . 3

1.1.2 Previous Work in Parallel Triangulation . . . . . 3

1.2 Preliminaries . . . . . 5

1.2.1 Geometric Definitions . . . . . 5

1.2.2 Parallel Models of Computation . . . . . 6

1.3 Overview of Point Set Triangulation Algorithms . . . . . 7

1.4 Organization of the Essay . . . . . 9

**2 Triangulation Scheme for Coarse Grained Multicomputers** **10**

2.1 Computing Convex Hull and Inside Triangulation . . . . . 11

2.2 Parallel Sorting . . . . . 12

2.3 Computing All-Tangents . . . . . 13

2.4 Triangulating the Funnel Polygons . . . . . 15

**3 Algorithm Implementation** **16**

3.1 Sample Sort Algorithm . . . . . 16

3.2 Algorithms for Convex Hull and the Triangulation of Local Points . . . . . 18

3.3 All-Tangents Parallel Algorithm . . . . . 18

3.4 Triangulating the Funnel Polygons . . . . . 20

3.4.1 Triangulation by Local Merging . . . . . 21

3.4.2 Triangulation using Global Sort . . . . . 24

3.5 Overall Time Complexity Analysis . . . . . 25

**4 Experimental Results** **26**

4.1 Hypercube Basic Performance . . . . . 26

4.2 The Sequential Algorithm . . . . . 27

4.3 Speedup and Communication/Computation Ratio . . . . . 29

4.4 Data Distribution Among Processors . . . . . 30

4.5 Work Distribution Among Phases . . . . . . . . . . 30
4.6 Worst Case vs. Expected Average Case . . . . . . . . . . 32
4.7 Message Size . . . . . . . . . . 33
4.8 Global Sorting applied to Funnel Polygon Triangulation . . . . . . . . . . 35

**5 Conclusions** **37**

**6 Acknowledgements** **38**

# 1 Introduction

## 1.1 Problem Description

### 1.1.1 Motivation and Significance

The problem of triangulating a finite set of points in the plane is a fundamental problem of computational geometry. It can be formally stated as follows : Given **n** points in the plane, join them by nonintersecting line segments such that every region interior to the convex hull is a triangle.

Planar triangulation is important because of its multitude of practical applications in surface interpolation, for calculations in numerical analysis and graphical display. Also other important computational geometry problems, like geometric searching and polyhedron intersection, use planar triangulations as a preprocessing phase.

Triangulation has been widely studied in both sequential and parallel settings. Despite the volume of the theoretical results reported in the literature, few of them are efficiently applicable for implementations on large scale parallel machines, mostly because of the differences between theoretical and architectural models. The work reported here is based on a realistic parallel model, and the experimental results confirm the estimated performance of the proposed algorithms.

### 1.1.2 Previous Work in Parallel Triangulation

Much of the theoretical work done so far has focused on designing parallel algorithms for Parallel Random Access Machines (PRAM) .

Goodrich and Atallah [20] have devised a parallel analog of the plane sweep that can triangulate a planar set of points in $O(lognloglogn)$ time using $O(n)$ processors and $O(n)$ space. ElGindy [21] has formulated an algorithm, which can triangulate a planar set of points in $O(log^2n)$ time using $O(n/logn)$ processors, thereby achieving linear speedup.

Merks [4] proposed an algorithm which triangulates a set of n points $S$ in $O(logn)$ time using $O(n)$ processors and $O(n)$ space. This is optimal in respect to both time and number of processors, since a triangulation algorithm can be used to sort and sorting has a parallel time lower bound of $\Omega(logn)$. It is based on reducing the problem of triangulating a set of points in the convex hull of S to triangulating points inside triangles.

An algorithm by Wang and Tsin [3] achieves the same running time, using also a multiway divide and conquer, but with a different decomposition of the problem. They partition the set of points into subsets, triangulating them within the convex hulls and funnel polygons which are

formed.

The simple characteristics of PRAM make it a suitable model for theoretical results in evaluating the complexity of parallel algorithms, but only a small number of real architectures (some bus based multiprocessors like Encore or Sequent) can be considered conceptually similar in design with the PRAM model. As the challenge to solve large and complex problems has constantly increased, achieving high performance computation by using large scale parallel machines became imperative. They are build as distributed memory multicomputers with large number of processors each of them having its own local memory. Processors are interconnected through a high speed network supporting message-based communication (Intel Paragon, Intel iPSC/860, CM-5).

Although any real machine can be used to simulate the PRAM model, it is nevertheless true that algorithms designed for network-based models will better match such architectures.

Miller and Stout [12] have proposed parallel algorithms of cost $O(n^{1/2})$ for some formal geometric problems, running on a mesh computer of size n. They employ different approaches to solve geometric problems than those that have been explored for these problems on serial computers, as the grouping technique (where sorting can also be included).

Their work, as well as the work done for the PRAM model focuses on the fine grained case, when the number of processors p is $O(n)$. However, for parallel geometric algorithms to be relevant in practice, such algorithms must be scalable, that is, they must be applicable and efficient for a wide range of ratios n/p. Miller and Stout [12] tried to extend the applicability of their results using a mapping method to simulate a mesh of size C*n on a mesh of size n and argue that these will not affect the asymptotic running time of the algorithms. In practice, the local computation and the interprocessor communication have different contributions to the total running time and therefore changing the granularity of local processing may affect the ratio of the two and finally the scalability of the algorithms. Even worse, in the case of PRAM model, the fine grained assumption (p=$O(n)$) used in the model runs against any usefulness of the theoretical results for real bus-based architectures, since we want n to be large but p is limited by architectural constraints (bus is a bottleneck). Therefore, there is a tremendous need to develop algorithms for those theoretical models of parallel computation, which can be easily mapped on the existing large scale parallel architectures.

Some progress in this direction has been done by Dehne, Fabri and Rau-Chaplin [6] who have studied scalable parallel computational geometry algorithms for the Coarse Grained Multicomputer model (CGM). They also offered a new systematic perspective in approaching computational geometry problem parallelization, showing that many geometric problems can be solved by algo-

rithms that use a constant number of one single global routing operation : global sort. The main idea consists in dividing a constant number of partitioning schemes of the global problem into p subproblems of size $O(n/p)$. Each processor will solve (sequentially) a constant number of such $O(n/p)$ size subproblems, and use a constant number of global routing operations to permute the subproblems between the processors. Eventually, by combining the $O(1)$ solutions of its $O(n/p)$ size subproblems, each processor determines its $O(n/p)$ size problem of the global solution.

Our research focuses on scalable parallel algorithms for the point set triangulation on the CGM model. We adapted the PRAM based algorithms to the CGM model and analyzed extensively the cost of computation and communication under more realistic assumptions, starting from their theoretical complexity and architectural model . We have also implemented the algorithms on a hypercube and made time measurements for both expected average and worst cases.

## 1.2 Preliminaries

### 1.2.1 Geometric Definitions

The elementary objects considered in computational geometry are normally a set of points in Euclidian space.

The **triangulation** of a set of n points in the plane consists of joining them by nonintersecting straight line segments so that every region internal to their convex hull is a triangle (see Figure 1).

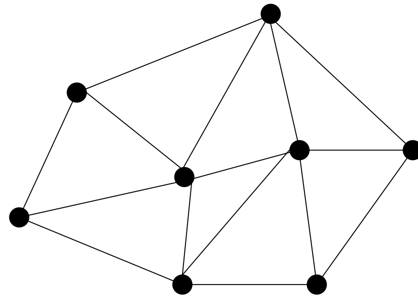

Figure 1: A valid triangulation

Given a set of points in the plane **S** = $p_1$, $p_2$,..., $p_n$, the **convex hull** of S , **CH(S)** is the boundary of the smallest convex domain in the plane containing S. Usually the convex hull is specified by listing the vertices in clockwise order starting from the point with the smallest x-coordinate. The point with the smallest (largest) x-coordinate is called the leftmost (rightmost) point.

A **funnel polygon** is a one-sided monotone polygon that consists of a single edge followed by

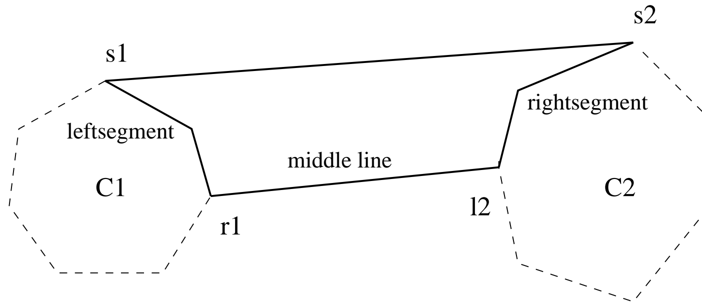


Figure 2: A funnel polygon

a convex chain followed by a single edge or vertex followed by another convex chain.

Let $C_1$ and $C_2$ be two convex polygons such that the points of $C_1$ have smaller x-coordinates than those of $C_2$. Let $r_1$ be the rightmost point of $C_1$ and $l_2$ be the leftmost point in $C_2$, call the line joining $r_1$ and $l_2$ *middle line*. Let the upper common tangent line of the polygons touch $C_1$ in $s_1$ and $C_2$ in $s_2$. The segment of $C_1$ from $s_1$ to $r_1$ is called leftsegment and the segment of $C_2$ from $s_2$ to $l_2$ is called rightsegment. The polygon formed by the line segment $s_1 s_2$, the middle line, the leftsegment, and the rightsegment form an upper funnel polygon (see Figure 2). Analogously, the lower funnel polygon can be defined.

Two points of a funnel polygon are mutually **visible**, if the segment connecting them does not intersect the boundary of the polygon.

#### 1.2.2 Parallel Models of Computation

Parallel algorithm analysis always refers to parallel models of computation which are abstractions of parallel machines, such as the shared-memory PRAM or the network based models. Usually an algorithm is designed for a particular model, in respect to which its complexity is analyzed. A parallel solution is considered optimal, if $T_{parallel} = O(T_{sequential}/p)$, where $T_{sequential}$ is the sequential time complexity of the problem and p the number of processors.

The **Parallel Random Access Machine** (PRAM) which is an idealized parallel model of computation, consisting of identical processors and a global memory, where all processors have unit time access to any memory location and a unit time communication diameter.

The common PRAM submodel, for which parallel solutions for computational geometry problems have been proposed is CREW PRAM (**Concurrent Read Exclusive Write PRAM**), in which concurrent reads to a memory location are permitted, but simultaneous writes to a memory location are disallowed.

In the **network models**, processors are loosely interconnected, which means that they have their own local memory and communicate through message exchanges. No shared memory is available. A network can be viewed as a graph G = (N,E), where each node $i \in N$ represents a processor, and each edge $(i, j) \in E$ represents a two-way communication link between processor i and j. The network topology (mesh, hypercube, etc) defines the submodel within the basic network model.

The **mesh** has processors arranged in a square lattice, each of them being connected to its four neighbors, if they exist.

A d-dimensional **hypercube** is obtained from n=$2^d$ processors, by connecting any 2 processors whose indexes differ in exactly one bit (assuming the processors are indexed from 0 to n-1). The hypercube is a preferred network model, because of its regularity, small diameter and interesting graph-theoretic properties.

In evaluating the performance of an algorithm on a parallel model of computation, two parameters : the size of the problem **n** and the number of processors **p** are essential. Based on the ratio of these parameters (**n/p**), we can distinguish fine grained and coarse grained models. The ratio n/p= $O(1)$ defines the **fine grained** case, while n/p $\geq$ p defines the **coarse grained** case. The latter is closer in modeling current multicomputer architectures, where the size of the local memory attached to processors is considerably larger than $O(1)$.

The **Coarse Grained Multicomputer model** (CGM) consists of a set of p processors, with size n memory evenly distributed among them, satisfying the coarse grained condition. The processors are connected through some arbitrary interconnection network, the common case being a hypercube of size n.

### 1.3 Overview of Point Set Triangulation Algorithms

Many criteria of what constitutes a good triangulation have been proposed, some requiring maximizing the smallest angle and some minimizing the total edge length. The Delaunay triangulation, which can be defined as the unique triangulation such that the circumcircle of each triangle does not contain any other point in its interior, can be constructed as a byproduct of the Voronoi diagrams or by a direct method that runs in optimal $O(nlogn)$ time. The Delaunay triangulation guarantees that the minimum angle of its triangles is maximum over all triangulations, which is a very useful property.

In what follows we will focus only on methods that produce a triangulation rather than a "good" one.

**Merks's Algorithm [1]:** In the first step, the convex hull of a set of $n$ points $S$ is computed in $O(logn)$ time using $O(n)$ processors using an algorithm from [17]. After the rightmost lowest point O is found and the rest of the points $p_i$ are sorted by angle $\theta_i$ that $p_i$ makes with O in the positive x-direction. This sorted sequence is split by lines through the extreme points of S and O. This gives a partitioning of the convex polygon into triangles. Consider a subsequence of points $p_l$, $p_{l+1}$,...,$p_{i-1}$,$p_i$,$p_{i+1}$,...,$p_{r-1}$,$p_r$ in one of these partitions, where $p_l$ and $p_r$ are the left and right extremes, respectively, given in clockwise order around CH(S). The height $\delta_i$ from the line through O parallel to the line through ($p_l$, $p_r$) is calculated for each point $p_i \in [p_l, p_r]$. An algorithm called *simple triangulation*, which takes a subsequence such as the one defined above and triangulates it, runs in $O(logk)$ time using $O(k)$ processors, where k is the number of points in the subsequence. Since a point can be in at most two subsequences, the processors can be divided among the subsequences so that $O(k)$ processors are used for each subsequence. The simple triangulation algorithm splits a subsequence of size k into $k^{1/2}$ subsequences of size $k^{1/2}$ and recursively triangulates each in a multiway divide-and-conquer process. The $\delta_i$'s are used to connect points to their left higher and right higher neighbors in the subsequence and down to the point O. Merging the triangulated subsequences in $O(logn)$ time with $O(n)$ processors is accomplished using a data structure similar to a segment tree. The entire algorithm takes $O(logn)$ time with $O(n)$ processors on a CREW PRAM.

**Wang and Tsin's Algorithm [1]:** In this case, the set of n points is partitioned into $n^{1/2}$ subsets of size $n^{1/2}$ each. The problem is solved recursively on each subset and during the algorithm, the convex hull of each of the subsets is created. The upper hulls of the $n^{1/2}$ convex hulls, $n^{1/2} - 1$ of their pairwise common upper supporting lines, and $n^{1/2} - 1$ middle lines connecting every two adjacent sets of points, form $n^{1/2} - 1$ funnel polygons.

An algorithm is presented that triangulates funnel polygons in $O(logm)$ time using $O(m)$ processors, where m is the number of vertices in the funnel polygon. A supporting line $l_{ij}$ is chosen for a pair of convex hulls $CH_i$ and $CH_j$ ($CH_j$ on the left of $CH_i$) where the slope of $l_{ij}$ is smaller than the slope of all supporting lines between $CH_i$ and hulls to the left of $CH_i$. The supporting lines can be found in $O(logn)$ time with one processor and since there are at most n supporting lines, in at most $O(logn)$ time using $O(n)$ processors. The algorithm to triangulate each funnel polygon is called and the entire process is repeated for the lower hulls. It is shown that allocating $O(n)$ processors to $n^{1/2} - 1$ funnel polygons so that each funnel polygon P of size m is allocated m-2 processors can be done in $O(logn)$ time using $O(n)$ processors on the CREW PRAM.

This algorithm was adapted to run on an $O(n)$-processor hypercube by MacKenzie and Stout [5] by dividing the set of points in $n^{1/4}$ subsets of size $n^{3/4}$ each. At each stage of the recursion

$O(Sort(n,n))$ time is used, where Sort(n,n) is the time needed to sort n numbers on an $O(n)$-processor hypercube. Using the fastest sorting known algorithm on a hypercube of size n that runs in $O(lognloglogn)$ time this triangulation algorithm runs in $O(lognloglogn)$ time.

### 1.4 Organization of the Essay

In this section we began by introducing some preliminaries. Basic geometric definitions used in algorithm description and parallel models of computation were presented. Prior related work was also discussed here.

We also reviewed proposed parallel algorithms for triangulation of a set of points.

Section 2 describes the triangulation scheme we proposed for the coarse-grained multicomputer model. Several subtasks are identified and analyzed.

Section 3 discusses various issues of the implementation we have done for the hypercube. We analyze the cost of computation and communication for all the steps involved in the sample sorting we implemented. We also analyze a distributed all-tangent computation which we implemented in $O(plog(n/p))$. For the funnel polygon triangulation we proposed both a suboptimal straightforward solution and an optimal one based on global sorting suggested in the model proposed by Dehne, Fabri and Rau-Chaplin [6].

In Section 4 we make further considerations related to the performance analysis of the implementation we made. Data sets for both expected average and worst case are provided and timed. We also discuss the effects of several parameters which affect the computation/communication ratio and thus contribute to optimally tunning the overall performance of the implementation. Load distribution is also measured, since several steps of the implementation (as the sample sorting, but not only) may impact on the balance of data and/or computation among the processors.

In Section 5 we draw the conclusions, emphasizing the strengths of this research.

## 2 Triangulation Scheme for Coarse Grained Multicomputers

The main purpose of this research was to design and implement an efficient parallel algorithm for triangulating a set of $n$ points using $p$ processors, where $n \gg p$. The model of parallel computers that will be used is the CGM, which consists of a set of processors, each containing a local memory and that communicate with one another through messages using a fixed interconnection network. Most of the algorithms described in Section 1 have focused on PRAM model running on machines with $O(n)$ processors, also referred as fine grained case. Although their results are also valuable for coarse grained architectures, since $O(n)$ processors can be simulated using p processors, it is nevertheless true that a scalable solution relevant for practical implementations must have a distinct design, based on the characteristics of the target architecture.

In what follows, we present a step by step description of the parallel triangulation scheme, based on the approach outlined by Wang and Tsin [3].

The triangulation of a set of n points S is accomplished by performing three basic tasks which are distributed among the p processors:

- Computation of the **basic convex hull** at each processor, corresponding to the local subset of points (of size n/p).
- Triangulation of the interior of the basic convex hulls.
- Triangulation of the funnel polygons.

The three tasks are accomplished as follows:

1. The set of points S is globally sorted after x-coordinate, then split into p sequences $S_1$,...,$S_p$, each of size n/p. Subset $S_k$ will reside at processor k.
2. A basic convex hull, $CH_k$ is computed locally at processor k and its interior points are triangulated.
3. The common tangent lines between all pairs of upper and lower hulls are computed. From those tangents, a subset of 2*(p-1) significant tangents are selected. The **significant tangents** are those, which along with the middle lines and edges of basic convex hulls determine the 2*(p-1) funnel polygons. As a byproduct of this phase, the convex hull of the set of n points is also obtained.
4. The 2*(p-1) funnel polygons are triangulated

These phases involve both sequential and parallel work.

The **sequential work** consists of :

- computing the convex hull of the local set of points.
- triangulating the set of points inside the basic convex hull.

These sequential tasks are performed independently at each site without involving any cooperation among processors.

The **parallel work** consists of :

- parallel sorting of the set S of n points.
- computing all pairs of tangents between convex hulls.
- triangulating the funnel polygons.

These tasks are performed by parallel algorithms, involving intense inter-processor communication.

## 2.1 Computing Convex Hull and Inside Triangulation

Since the problem of sorting is linear-time reducible to the Convex Hull problem, finding the ordered convex hull of n points in the plane requires $\Omega(nlogn)$ time [22].

Many algorithms for computing the Convex Hull of n points in the plane have been designed, to mention some of them [2]: Jarvis's march running in $O(n*h)$ where h is number of hull edges, thus having a worst case complexity of $O(n^2)$; the Incremental algorithm based on a presorting of the points by x-coordinate followed by a processing scan, thus running in $O(Sort(n))+O(n) = O(nlogn)$ time; the QuickHull algorithm, based on the same idea as Quicksort, thus suffering from a worst case complexity of $O(n^2)$, but in case of a reasonable distribution of points, it can achieve an $O(n)$ time; Graham's Scan that relies on an angular presorting of the points in counterclockwise about an interior point, followed by a processing scanning, thus having an $O(nlogn)$ time complexity; finally, Kirkpatrick and Seidel designed an $O(nlogh)$ algorithm (where h is the number of points on the convex hull) using a divide-and-conquer method.

Since the implementation uses Graham's Scan method, we will briefly describe the algorithm.

**Graham's Scan Algorithm**:

1. Choose an internal point O (to the convex hull of the n points).

2. Using O as the origin of coordinates, sort the n given points by polar angle and distance to O forming a list $p_1$, $p_2$,..., $p_n$ of points.

3. Process the n points, keeping at each step $i$ a list $l_1$, $l_2$, ... ,$l_j$ from which those points $l_j$ are eliminated, if the angle $l_{j-1}l_jp_i$ is reflex. During this scan, the triangulation of the points inside the convex hull is also achieved.

At the completion of the algorithm the list contains the hull vertices in counterclockwise order. Graham's Scan algorithm is optimal in the worst case ($O(nlogn)$).

## 2.2 Parallel Sorting

There are many parallel sorting algorithms described in the literature for both PRAM and hypercube models. Blelloch et al.[8] have made a useful comparison of several parallel algorithms. For this purpose they implemented and evaluated several algorithms on CM-2. Since their analysis on CM-2 is based on a CGM-like view of the machine, we rely on their results, claiming that they also apply for the CGM model. They show that the sample sorting algorithm which is a theoretical efficient randomized algorithm is the fastest of the tested algorithms when the computation is performed on large data sets ( the number of keys n $\gg$ the number of processors p). Although deterministic sorting algorithms like bitonic or parallel radix sort are more robust and simpler to code, they tend to be slower on large data sets. For instance, the sample sort beats radix sort by more than a factor of two for large data sets. Also, the bitonic sort which is known to be the most efficient one for small data sets is at the same time the least efficient algorithm when tested on large data sets (4 times slower than the sample sort).

Since the sorting we implemented for the first phase was the sample sort, we are going to describe it in more detail.

**Sample Sort** is a randomized sorting algorithm, whose running time does not depend on the distribution of input keys, but on the outputs of a random number generator.

Assuming n input keys are to be sorted on a machine with p processors, the algorithm proceeds in three steps.

**Sample Sort Algorithm :**

**Step 1** : A sets of p-1 splitter keys are picked, that partition the linear order of key values into p buckets.

**Step 2** : Based on their values, the keys are sent to the appropriate bucket, where the i-th bucket is stored at the i-th processor.

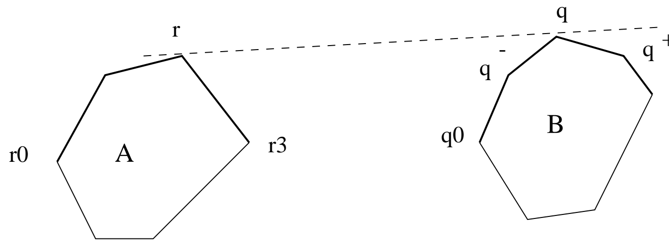


Figure 3: Finding the upper tangent between two convex hulls

**Step 3** : The keys are sorted in each bucket sequentially.

The name “sample sort” comes from the way the p-1 splitters are selected during the first phase. From the n input keys, a sample of p*s << n keys are chosen randomly, where s is a parameter called the **oversampling ratio**. This sample is sorted and the p-1 splitters are selected by including those keys in the sample that have ranks s, 2*s, 3*s... (p-1)*s. Some sample sort algorithms reported in the literature [8] choose an oversampling ratio of s=1, which results in a relatively large deviation of the bucket sizes. If s>1 is chosen as suggested [8], it is guaranteed with high probability that no bucket contains many more keys than the average. The time for the third phase of the algorithms depends highly on the maximum numbers (L) of keys in a single bucket. Since the average bucket size is n/p, the efficiency by which a given oversampling ratio s maintains small bucket sizes can be measured as the ratio L/(n/p) which is called **bucket expansion**. The expected value of the bucket expansion depends on the oversampling ratio s and on the total number of processors p.

It is extremely unlikely that the bucket expansion will be significantly worse than expected. In [8] it is showed that the probability that the bucket expansion is greater than some factor $\alpha > 1$ is $Pr[L > \alpha(n/p)] \leq n * e^{-1(1-1/\alpha)^2 \alpha s/2}$.

The running time of the sample sort depends linearly on both the oversampling ratio and the bucket expansion. In practice, oversampling ratios of s=32 or s=64 yield bucket expansion of less than 2 [8].

## 2.3 Computing All-Tangents

The input of this phase consists of a collection of convex polygons, $CH_i$, each of them residing at a different processor, where $CH_i$ is to the left of $CH_j$, if i<j. The main idea is to initiate a distributed computation for each tangent between any pair of processors for both upper and lower tangents. The core algorithm is an adaptation of the solution described by Mehlhorn [9].

Figure 3 illustrates the computation of the common upper tangent between the convex hull

A and B. To identify the upper tangent $rq$, the points $r \in$ Upper(A) and $q \in$ Upper(B) must be computed ($r$ has to be on the upper chain of $A$, between the leftmost and the rightmost points of $A$). They are computed using a binary search method to halve the search range on at least one side at each iteration. Therefore $O(logn)$ iterations are necessary, where n is the initial search size.

The following algorithm computes the common upper tangent of 2 convex hulls A and B (A on the left of B), by iterative binary searches on the two upper chains Upper(A)= $r_1$, $r_2$,...,$r_m$ and Upper(B) = $q_1$, $q_2$,...,$q_p$ , thus ensuring an $O(logn)$ with n=m+p. The algorithm finds the ranks $h \in (1, m)$ (in Upper(A)) and $k \in (1, p)$ (in Upper(B)) such that line $r_h q_k$ does not intersect either A or B. Starting with ($r_l = r_1$,$r_h = r_m$) and ($q_l = q_1$,$q_h = q_p$) as the initial search ranges, at each step $i = (r_l + r_h)/2$ and $j = (q_l + q_h)/2$ are computed. The oriented line $r_i q_j$ can either touch, enter or leave A and B. According to its relative position, we can distinguish 9 cases.

**All-Tangents Algorithm [9]:**

**Case 1:** $r_i q_j$ touches in $r_i$ and $q_j$. The line is tangent to A and B and the algorithm completes with h=i and k=j.

**Case 2:** $r_i q_j$ touches in $r_i$ and enters in $q_j$. Then $r_h$ certainly does not follow $r_i$ ($h \leq i$) and $q_k$ does not precede $q_j$. Hence $r_l = j$ and $q_h = j$ halves the search ranges on both polygonal chains.

**Case 3:** $r_i q_j$ touches in $r_i$ and leaves in $q_j$.Then $r_h$ does not follow $r_i$ and $q_k$ does not follow $q_j$. Hence $q_h = j$ and $r_h = i$ reduces the size of the problem.

**Case 4:** $r_i q_j$ leaves in $r_i$ and touches in $q_j$. This case is symmetric to case 2.

**Case 5:** $r_i q_j$ enters in $r_i$ and touches in $q_j$. This case is symmetric to case 3.

**Case 6:** $r_i q_j$ leaves in $r_i$ and enters in $q_j$. Then $r_h$ does not follow $r_i$ and $q_k$ does not precede $q_j$. Hence $r_h = i$ and $q_l = j$ reduces the size of the problem.

**Case 7:** $r_i q_j$ leaves in $r_i$ and leaves in $q_j$. Thus $r_h$ does not follow $r_i$ and $r_h = i$ reduces the problem size.

**Case 8:** $r_i q_j$ enters in $r_i$ and enters in $q_j$. This case is symmetric to case 7.

**Case 9:** $r_i q_j$ enters in $r_i$ and leaves in $q_j$. Let **d** be a vertical line such that no point of A (B) is to the right (left) of **d**, and let $t_A$ ($t_B$) be a tangent to A (B) in point $r_i$ ($q_j$). Let **s** be the intersection of $t_A$ and $t_B$. Assume that **s** is to the left or on **d**, the other case being

symmetric. Since all of B is to the right or on **d** and below or on $t_B$ and hence below $t_A$ we conclude that $r_h$ cannot precede $r_i$. Hence $r_l = i$ reduces the size of the problem.

Thus, at each step at least one of the ranges is halved in $O(1)$ time, yielding an $O(logn)$ time complexity.

### 2.4 Triangulating the Funnel Polygons

The final phase of our algorithm is to triangulate the 2*(p-1) funnel polygons which are determined by the p basic convex hulls, along with the significant tangents and the middle lines. This step is a consequence of splitting the triangulation task among processors.

The triangulation of funnel polygons can be reduced to visibility testing, which induces an ordering relation among the points.

Let $a_1$, $a_2$,...,$a_n$ be the points on the leftsegment and $b_1$, $b_2$,...,$b_m$ be the points on the rightsegment. The ordering is defined by: $a_1 \prec a_2 \prec ... \prec a_n$ , $b_1 \prec b_2 \prec ... \prec b_m$ and $a_i \prec b_j$ if the angle $a_{i-1}a_ib_j \leq 180^o$. This relation is used to triangulate the funnel polygon by connecting each point of the leftsegment (rightsegment) with its predecessor on the rightsegment (leftsegment). Therefore each point on the leftsegment is connected with:

- each visible point on the rightsegment if the latter wasn't visible from the previous point on the leftsegment (possibly none).
- the last point on the rightsegment that was visible from the previous point on the leftsegment.

# 3 Algorithm Implementation

We have implemented the above outlined triangulation scheme on a hypercube, which is essentially a distributed memory parallel machine with p processors, with large local memory and an interconnection network of diameter log p. To distinguish between computation and communication time, we consider **R(k)** the cost of sending a message with k points and **C(k)** the cost of a generic operation over k points. We also introduce **Sort(n,p)** as the cost of a generic global sort of n points on p processors. We ignore the additional cost of message routing, considering the same cost for communication between any two processors. This is a reasonable assumption, since in the current multicomputers (Intel Paragon, Intel iPSC/860) direct connection modules do the routing without interrupting the processor, and thus making the routing time negligible.

In this section we will detail the sequential and parallel algorithms we have implemented on a hypercube and make realistic complexity estimates based on this model's assumptions.

## 3.1 Sample Sort Algorithm

We perform a sample sorting using as keys the x-coordinates of the set of points.

**Phase 1: Selecting the splitters**

**Step 1.1**

Each processor selects a set of s keys ( s being the oversampling ratio) from among those stored locally in its memory. This can be done randomly, but in our implementation one every (n/p)/s key is selected. Blelloch et al. [8] propose as typical values for s 32 or 64 for data sizes of order $10^6$, but it must be tunned depending on the number of keys per processor (n/p). We evaluated the bucket expansion as a function of oversampling ratio for different numbers of keys. As the oversampling ratio increases, the bucket expansion decreases, but at the same time it increases the additional cost of sorting the splitters.

**Step 1.2**

Once the samples are chosen, they are sorted across the machine using a parallel mergesort. Since the sample contains much fewer keys than the input, apparently this step runs significantly faster than sorting all the keys using the same algorithm.

The relative contribution of the cost of sorting the samples to the total cost of sample sort decreases as the number of keys increases. For 16K points sorted on 8 processors, an oversampling ratio of 32 adds less than 5% to the total cost of sorting.

**Step 1.3**

The splitters are now chosen as the keys having ranks s, 2*s, 3*s..., (p-1)*s. They are broadcasted by processor 0 (the one where the mergesort of the samples ends).

The dominant time required in phase 1 is the time to sort the samples, plus the cost of broadcasting (step 1.3), yielding:

T1=Sort(p*s,p) + (p-1)*R(p-1)

Notice that the time for phase 1 is independent of the total number of keys n, since the selection of the set of s samples is done without looking at all n/p keys.

**Phase 2: Distributing the keys to buckets.** The p-1 splitters define p intervals for the keys. Each interval is assigned to a processor . Then each processor determines for each of its keys the bucket to which it belongs by binary searching the range of splitters. If it turns out that the key belongs to the local processor, then it is saved in its local buffer, otherwise it must be sent to the appropriate destination. This can be done for each key, but we choose to group them in local buffers, one for each processor that will be sent out when they become full and at the end of the distribution phase. In this manner we can amortize the cost of sending a message over a number of keys equal to the size of the buffer. Therefore the size of the buffer (k) becomes a parameter, which can be tunned in order to get a minimum cost.

The time required by phase 2 consists of the time for binary search and the time for sending the buckets to the corresponding nodes.

T2=(n/p)*C(log p) + ((n/p)/k)*R(k)

**Phase 3: Sorting the keys within each processor.** After all the keys have been distributed, each processor has to sort the keys which belong to its interval, using a quicksort routine provided by the C-library. The time taken by this phase is the time for the processor with the most keys in its bucket to sort. If the expected bucket expansion is $\beta(s,n)$, the largest bucket size will have the expected size (n/p) $\beta(s,n)$. The time is

T3=Sort((n/p)$\beta(s,n)$,1)

As implemented, the sample sorting is suitable only when the number of keys per processor (n/p) is large. When the algorithm has completed, not all the processors have the same number of keys. Because the sorting is used to distribute the computation of the basic convex hulls among processors, the fact that the processor load is not completely balanced is not important.

With an additional cost, a load balancing can be performed but, for reasonable bucket expansion ($\leq 2$), it is not needed.

Thus:

$T_{Sort}$=Sort(p*s,p)+(p-1)*R(p-1)+(n/p)*C(log p)+((n/p)/k)*R(k)+Sort((n/p)$\beta(s,n)$,1)

### 3.2 Algorithms for Convex Hull and the Triangulation of Local Points

At each processor, the computation of the Convex Hull of its *n/p* local points is performed sequentially using Graham's Scan Algorithm [10]:

**Step 1.** The rightmost lowest point $P_0$ is found in C(n/p) time and considered as origin.

**Step 2.** It then sorts the remaining n/p-1 points angularly about $P_0$ breaking ties in favor of the one closest to this origin. No angles are computed, because testing for right turns (reflex angles) is enough to determine whether the point will be part of the convex hull or not. The list $P_0$,$P_1$, ...,$P_{n/p}$ of angularly sorted points is formed. This step takes Sort(n/p,1) time using an implementation of quicksort provided by the C-library.

**Step 3.** The sorted list of points is then scanned and at each step **i** a candidate list for the convex hull chain $l_0$, $l_1$,...,$l_j$ is kept in a stack. While $l_{j-1}l_jP_i$ is a reflex angle, the point $l_j$ is popped out of the stack . Finally $P_i$ is pushed onto the stack for the next step.

At only the cost of a constant amount of work, during this scan, the triangulation of the points inside the convex hull is performed : If $l_{j-1}l_jP_i$ is not a reflex angle, the triangle $P_0l_jP_i$ is added to the triangulation, otherwise two triangles $P_0l_jP_i$ and $P_il_jl_{j-1}$ are added when the first element is popped out of the stack, and only triangle $P_il_jl_{j-1}$ while further popping is performed. Thus, this step takes time:

T= C(n/p).

$T_{CH}$= 2*C(n/p)+Sort(n/p,1).

### 3.3 All-Tangents Parallel Algorithm

The All-Tangents parallel algorithm was implemented by parallelizing the sequential algorithm of Mehlhorn [9] using the scheme suggested by Dehne et al. [6]. Each upper (lower) tangent $u_{ij}$ ($l_{ij}$) is computed in at most 2*log (n/p) steps using binary search on the ranges determined by the leftmost and rightmost points of the basic convex hulls.

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| $U^{0}_{03}$ | $U^{0}_{01}$ | $U^{0}_{12}$ | $U^{0}_{23}$ |
| $L_{01}$ | $L_{12}$ | $U_{02}$ | $U_{13}$ |
| $L_{02}$ | $L_{13}$ | $L_{23}$ | $L_{03}$ |

(a) ITERATION STEP 0

| | | | |
|---|---|---|---|
| $U^{1}_{01}$ | $U^{1}_{12}$ | $U^{1}_{23}$ | $U^{1}_{03}$ |
| $U_{02}$ | $U_{13}$ | $L_{02}$ | $L_{13}$ |
| $L_{03}$ | $L_{01}$ | $L_{12}$ | $L_{23}$ |

(b) ITERATION STEP 1

Figure 4: Example of distribution of the all-tangents computation tasks

Let $U_{ij}$ be the computation task for tangent $u_{ij}$. The O(log (n/p)) steps of $U_{ij}$ which might be necessary to compute $u_{ij}$ will be alternatively executed by processors i and j. Let $U_{ij}^{k}$ be the step k of the computation. Then, if steps $U_{ij}^{0}$, $U_{ij}^{2}$ ... are executed on processor i, then $U_{ij}^{1}$, $U_{ij}^{3}$ ... are executed on processor j. Figure 3.3 shows the distribution of the task for the first two steps on a 4 processor hypercube.

Since p basic convex hulls are computed, there will be p*(p-1)/2 upper tangents and the same number of lower tangents, for a total of p*(p-1) tangents. They define p*(p-1) computation tasks, which, being a multiple of p, can be evenly initiated by the p processors, in order to avoid congestion. Each processor executes at each step a computation task and sends his result to the partner with whom he has to exchange the computation of their common tangent.

Here is the algorithm:

**Preprocessing step:** In the beginning each processor broadcasts its leftmost and rightmost point to all other processors ( necessary for case 9 in Mehlhorn's algorithm). Then the processor initializes 2 tables (one for the upper tangent and one for the lower one) with one entry for each of the p-1 other convex hulls it is computing the tangent lines with. An entry in this table consists of 2 range values, for both left ($r_0$,$r_k$) and right ($q_0$,$q_l$) endpoints of the

tangents, a tentative value for the tangency point (the midpoint of the range) at the remote hull (q), along with its predecessor $q^-$ and successor $q^+$ points on the remote hull (see Figure 3). This step takes:

T=2*C(p-1)+(p-1)*R(2)

**Starting step:** Each processor initiates remotely the computation of p-1 tangents, by sending $\lceil (p-1)/2 \rceil$ messages for upper tangents to the right and $\lfloor (p-1)/2 \rfloor$ to the left. Each message contains the information from the entry in its local table corresponding to the destination of the message. The range is initially set between the leftmost and rightmost extreme points of the hull. This step takes:

T=(p-1)*R(3)

**Iteration step:** At each step, each processor receives at most p-1 messages for outstanding tangent computations, on which it has to progress and to send back the results for the next step.

Based on the provided information in the received message, each processors evaluates the condition of Mehlhorn's algorithm and decides upon updates in the corresponding range values for the tangent endpoints on both his and the remote hulls. It also detects when the tangent is computed, in which case that task is terminated. Therefore, all the tangent lines between the p convex hulls stored on p processors can be computed in :

T=log(n/p)*(C(1) + (p-1)*R(3))

**Postprocessing step:** At this step, 2*(p-1) significant tangents are identified. Having computed all its tangent lines, each processor can now compute the uppermost (lowermost) tangent to its left in C(p) time and broadcast it to every other processor in R(1) time. Thus:

T= C(p-1) + (p-1)*R(1).

$T_{All-Tangents}$= 2*C(p-1)+(p-1)*R(2)+(p-1)*R(3)+log(n/p)*(C(1)+(p-1)*R(3))+C(p-1)+(p-1)*R(1).

### 3.4 Triangulating the Funnel Polygons

We will describe 2 different approaches for the triangulation of the funnel polygons. To make things clearer, we will use the triangulation of the upper funnel polygons in Figure 5 as an example.

The first one is a straight application of a multiway divide and conquer paradigm. It applies the visibility ordering criteria for local merging of the points of the left- and rightsegment of a funnel polygon. Since it needs O(n) time in the worst case, it is an example when trying to parallelize

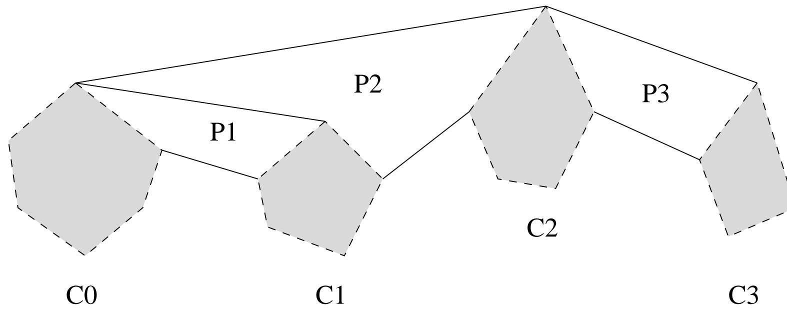


Figure 5: Parallel triangulation of funnel polygons

sequential algorithms using this paradigm yields suboptimal solutions. The second approach is based on a global sort and has an Sort(n,p) time complexity.

### 3.4.1 Triangulation by Local Merging

This method is implemented as 2*(p-1) distributed tasks, which are performed in parallel using the same idea as for computing all tangents in parallel. Each task, $T_i$, corresponds to a funnel polygon and is performed distributedly by all the processors whose hulls contribute points to that funnel polygon (for instance the funnel polygon $P_2$ will be triangulated by the task $T_2$ which will be distributed among processors $C_0$, $C_1$ and $C_2$. Therefore, each processor might be simultaneously involved in several tasks, as many as the number of funnel polygons to which its points are members of (for instance processor $C_0$ will be simultaneously involved in the triangulation of funnel polygons $P_1$ and $P_2$). After a processor has completed a step of a particular task, it stops and saves the complete state of the execution in a message, which is then sent to the appropriate partner. In receiving the next message the processor will resume the execution of another task for which he got that message. This activity is driven through messages of different types (T-POINTS,T-MORE,T-TERMINATED), allowing a processor to save and resume the execution of different tasks in which it participates. Each processor, except $C_0$, is responsible for serving two funnel polygon triangulations (upper and lower - those for which his convex hull determined the rightsegment). In our example processor $C_i$ will be responsible for the upper polygon i. For these funnel polygons, the processor is responsible to provide rightsegment points to the left partner, which tests for visibility and performs the triangulation. It can send one point at a time, but we chose to send them in buckets since we can then tune the size of the bucket for optimal balance between communication and computation at this phase.

**1.Preprocessing step:** At this step, each processor sorts the significant tangents for which he contributes with the left endpoint by y-coordinate and angle. This work is useful to determine

the next convex hull participating in the triangulation of the current funnel polygons. It takes:

T= Sort(p,1)

2. **Initiation step:** Each processor **i** initiates the triangulation tasks $T_i^U$ and $T_i^L$ for the two funnel polygons it is responsible for by sending the first bucket (of size k) with his points (belonging to the rightsegment) to the first left partner, with whom it shares the significant tangent. In our case, for instance, processor $C_2$ will initiate the task $T_2$ for the triangulation of $P_2$, by sending the first bucket of points to $C_0$ (its first left partner). This takes:

   T=2*R(k)

3. **Iteration step:** In receiving a message, a processor resumes the execution of the triangulation task, who's state was saved in that message. The service it performs depends on the type of the message it receives. For each triangulation task, a processor may perform one of the following actions, depending on the type of message it receives:

   - T-POINTS

     This is a message received by a left partner from the right partner or, as a result of a forwarding, from the former left partner. A processor receives such kind of messages, as long as it is the current left partner in the triangulation of that funnel polygon. The message contains a bucket of points from the rightsegment, from which it tries to consume (by triangulating) as many as possible, using the visibility order and its points as leftsegment points. In our case, $C_0$ will triangulate his points with the points he received from $C_2$, performing the task $T_2$. This step terminates when :

     1. the bucket of points is emptied, in which case the processor stops the current task, sends a T-MORE message to the right partner asking for the next bucket of rightsegment points. This triangulation task will be resumed when a new bucket of points will arrive. In our example $C_0$ may ask for more points from $C_2$ if he has finished triangulating those sent in the previous message.
     2. the next point from the rightsegment is not visible any more from the left partner points. In this case, the processor triangulates its remaining points with the last visible right point from the rightsegment and forwards the remaining points in the buckets by sending a message to the next processor (determined by using the sorted list of significant tangents) participating in the current funnel polygon. This processor will become the current left partner. In this case the processor, now being the former left partner, finishes its contribution to that triangulation task. In our

example, at some point, $C_0$ will forward the remaining points in the bucket to $C_1$ which will become the next left partner of $C_2$ in the task $T_2$.

3. the processor learns that the last bucket of points has been sent from the right partner and that he is the last left partner in the funnel polygon. In this case it send a T-TERMINATE message to the right partner containing its last leftpoint (rightmost point of the left partner). This will be the case with $C_1$ which is the last left partner of $C_2$ in the triangulation of $P_2$.

- T-MORE

  This message is received by the right partner in the funnel polygon. By receiving this message, the processor learns about its current left partner, which might have been changed as a result of forwarding, to which it has to send the next bucket of points (marking it if it is the last one). In our case, $C_2$ may receive such messages from $C_0$ and, after $C_0$ has triangulated all its points, from $C_1$. When $C_2$ will receive the first message of type T-MORE from $C_1$, it will know that its left partner was changed (from $C_0$ to $C_1$) and thus the triangulation of $P_2$ will be continued with $C_1$.

- T-TERMINATE

  In this case, the processor is also the right partner in the polygon. By receiving this message, the processor learns that all the points on the leftsegment have been triangulated and it only has to triangulate using the last leftpoint its remaining right points (if any) and then terminate that triangulation task. When $C_2$ will receive a T-TERMINATE message from $C_1$, it will triangulate its remaining points with the rightmost point of $C_1$ which he sent in the message (instead of sending them to $C_1$ for the same thing).

This step of the algorithm has a worst case complexity of :

T= ((n/p)/k+p-2)*R(k) + (p-1)*C(n/p) = $O(n)$

where k is the bucket size.

There is also a load distribution problem here, since some processors might be overloaded if they participate in many funnel polygons.

Thus:

$T_{FunnelPolygons}$ = Sort(p,1)+2*R(k)+((n/p)/k+p-2)*R(k) + (p-1)*C(n/p).

#### 3.4.2 Triangulation using Global Sort

**Preprocessing step:** At this step, the points belonging to the funnel polygons are labeled . Each processor initiates the labeling of the upper and lower funnel polygon for which he determined the tangent. Then, based on the information about the other significant tangents, it identifies the boundaries of the hulls that are part of his funnel polygon's leftsegment and computes the rank range for each of them. This information, along with the funnel polygon identifier is sent to all left partners for both its upper and lower polygons. In our example, $C_2$ may send this information to both $C_0$ and $C_1$. Each processor, after receiving the labeling messages, uses them to label all its n/p points attaching them the following information:

- funnel polygon number
- rank in the funnel polygon
- predecessor point
- left/right segment membership flag

If a point belongs to several funnel polygons, a copy of it is created for each additional funnel polygon (they correspond to the endpoints of the significant tangents which are O(p) and thus cannot increase the space requirements by more than a constant amount).

This step runs in Sort(p,1)+C(n/p) time.

**Global sorting step:** A global sorting is performed by considering successively the following criteria:

- funnel polygon number
- between same-sided points, the rank in the funnel polygon
- between opposite-sided points, the ordering relation based on visibility

This takes T=Sort(n,p) time.

**Triangulation step:** Finally, each processor has to compute its last left and right segment point he has in its local ordered sequence of points in C(n/p) time and has to broadcast them to its right processors in R(2) time. This will allow each processor to choose the highest left and right point from the nearest left processor and start the triangulation of its ordered sequence without waiting for the termination of the ones to his left in C(n/p+p). Thus, this step takes time:

T= C(n/p +p)+C(n/p)+p*R(2)

$T_{FunnelPolygon(GS)}$ = Sort(p,1)+Sort(n,p)+C(n/p +p)+2*C(n/p)+R(2)

### 3.5 Overall Time Complexity Analysis

Based on the previous analysis, the overall time complexity of the algorithm can be estimated as follows:

$T_{TOTAL}$= $T_{Sort}$+$T_{CH}$+$T_{All-Tangents}$+$T_{FunnelPolygons}$

$T_{TOTAL}$= Sort(n,p)+Sort(n/p,1)+p*R($k_0$)*log(n/p)+Sort(n,p)

If we approximate

Sort(n,p) = (n/(p*k))*R(k)+Sort(n/p,1) then

$T_{TOTAL}$ = (n/p)*(R($k_1$)/$k_1$+R($k_2$)/$k_2$) + 3*Sort(n/p,1)

which can be viewed as

$T_{TOTAL} = T_{communication} + T_{computation}$

The contribution of communication decreases as n/p increases (by increasing n) since Sort(n,1) is Ω(n log n) while the cost of communication is linear. Therefore, the overall complexity of the algorithm gets asymptotically close to the complexity of sorting. The communication term can be made less significant even sooner, by reducing the constant factor. This is a minimization problem and clearly, as k (the size of the message) increases, there is a point where the communication cost is minimum.

On the other hand, when n/p decreases (by increasing p), the contribution of computation reduces, making the communication cost dominant.

# 4 Experimental Results

A full implementation of the described algorithms was written in C-code (approximately 2200 lines) for a hypercube multicomputer. Extensive tests have been conducted on sets of points of sizes n $\in$ [1000,250000] points . The number of processors p varied between 1 and 16. Clearly, for sizes greater than 4096 points we were set in a CGM model, since n/p was bigger than p with more than an order of magnitude (for any p in the considered range).

The goal of these experiments was to evaluate the performance of the algorithms described in Section 3. To cover the expected average case analysis we have tested the algorithms on pseudo-random generated points in a square. We have also produced special sets of points (uniformly generated on a circle) to test the worst case behavior of the algorithm.

In this section we will focus on the speedup analysis of the parallel implementation and various alternatives to improve the performance of the algorithms for the chosen architecture. The impact of the computation/communication ratio as well as of the message size on the total execution time is also examined.

Detailed analysis will be made for each of the four phases of the algorithm:

**Phase 1:** Global sorting of points after x-coordinate.

**Phase 2:** Basic convex hull computation and inside triangulation.

**Phase 3:** All-tangent computation.

**Phase 4:** Funnel polygon triangulation.

We will start this section with performance evaluation of basic operations on the machine on which we conducted the tests.

## 4.1 Hypercube Basic Performance

We ran several simple tests on the target machine in order to evaluate its performance characteristics. Our machine was an Intel iPSC/860 with 16 processors, running at 40MHz. We measured the cost of the basic arithmetic operations (Figure 6) by running a test program on an i860 node. All operations were done on double precision floating point numbers. Since the division is the only operation on floating point numbers which is emulated in software, its cost is two orders of magnitude higher than the cost of an addition or multiplication operation. Therefore, we tried to avoid as much as possible divisions inside the basic loops, in order to optimize the performance.

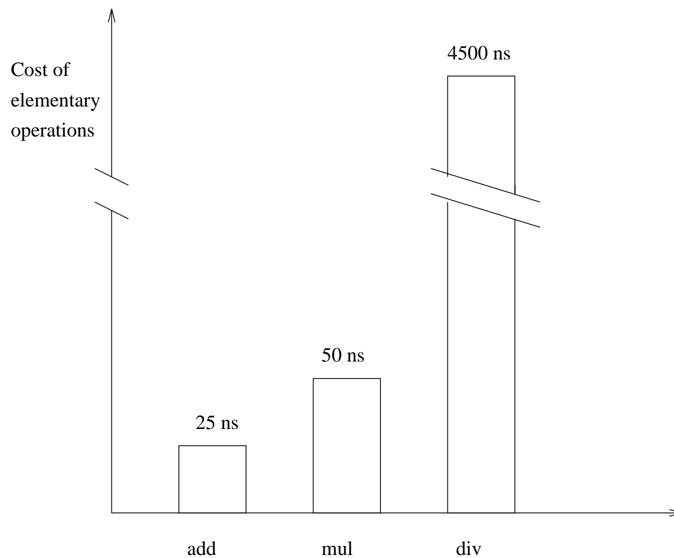


Figure 6: Elementary Operation Costs for Hypercube

Memory access cost is an important issue when a theoretical algorithm is evaluated on a real machine, even though in the theoretical analysis any load/store is considered of unit cost. In reality, the cost of a memory access may differ within an order of magnitude; if the word is missing from the cache its access costs about 15 cycles instead of only one. We identified the effect of the cache by running a test program with increasing data size (Figure 7). The discontinuity in its timing (at 16KB) confirms the existence of an internal data cache of 8K. Blocking is the main solution used to improve the locality of the algorithms. In our implementation we used the qsort routine provided by the C-library for which we cannot control the blocking. Therefore, for input sizes greater than 8KB, the execution time may be affected also by the location of the data in memory.

The main characteristic of the machine which significantly affects any performance evaluation is the cost of communication. The iPSC/860 is an architecture with a hypercube network which provides fast connection between any two nodes. We measured the cost of messages on iPSC/860 (Figure 8). We computed a message latency of $250\mu$s and a transfer rate of approximately 2.2MB/sec for message sizes greater than 100 bytes. We have also found insignificant differences for the case when messages have to go through a number of hops (less than $15\mu$s per hop).

### 4.2 The Sequential Algorithm

We examined the behavior of the sequential algorithm for triangulation. In Figure 9 we plot the total execution time along with the time of its two main phases: sorting and convex hull triangulation. The dependence is linear for triangulation and almost linear in the size of the

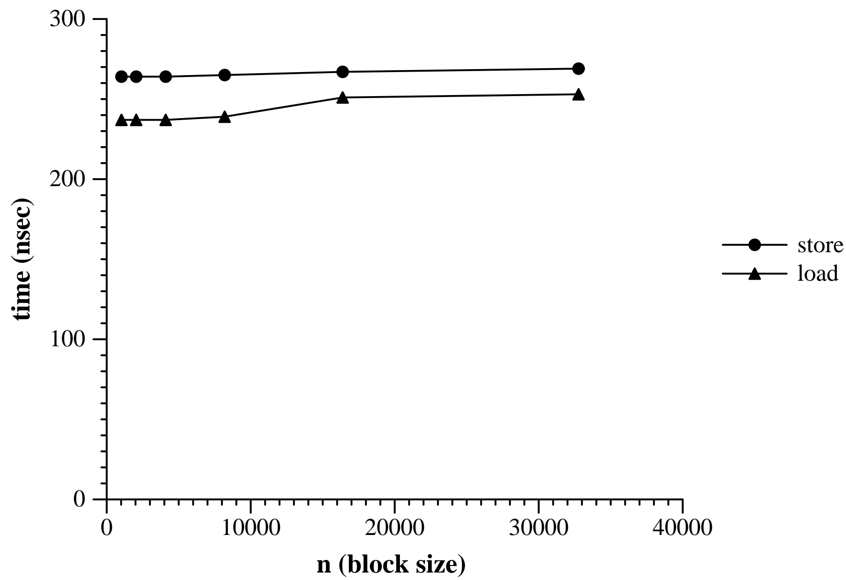


Figure 7: Memory Access Cost for Hypercube

input for sorting and total execution time, results which corresponds to the theoretical complexity evaluation. In our C implementation of the triangulation algorithm, an execution on 64K points with double precision floating point representation of their coordinates, takes approximately 10 sec. About one third (3.2 sec) is the cost of sorting and two thirds (7.2 sec) the cost of the convex hull computing and inside triangulation. In order to ensure a fair evaluation of the speedup, the sequential implementation used the same basic algorithms as the parallel one.

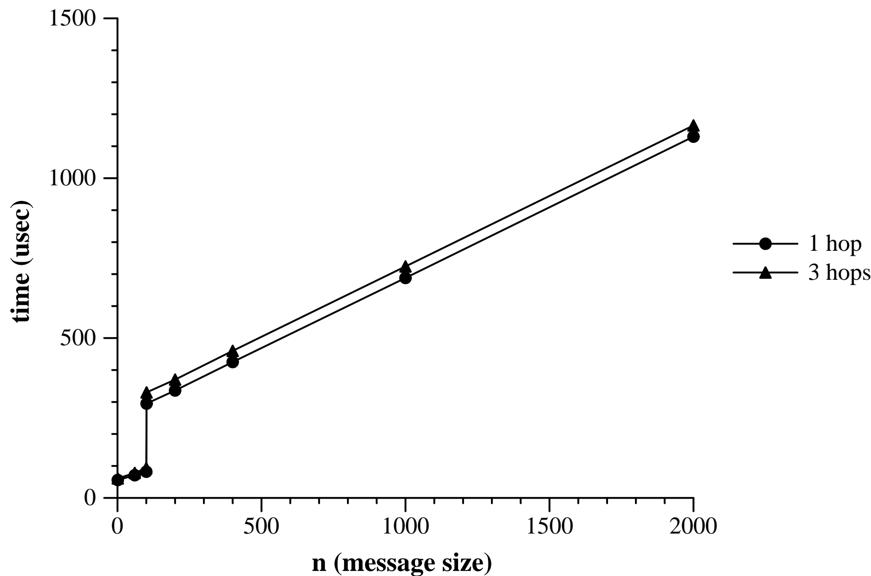


Figure 8: Communication Cost for Hypercube

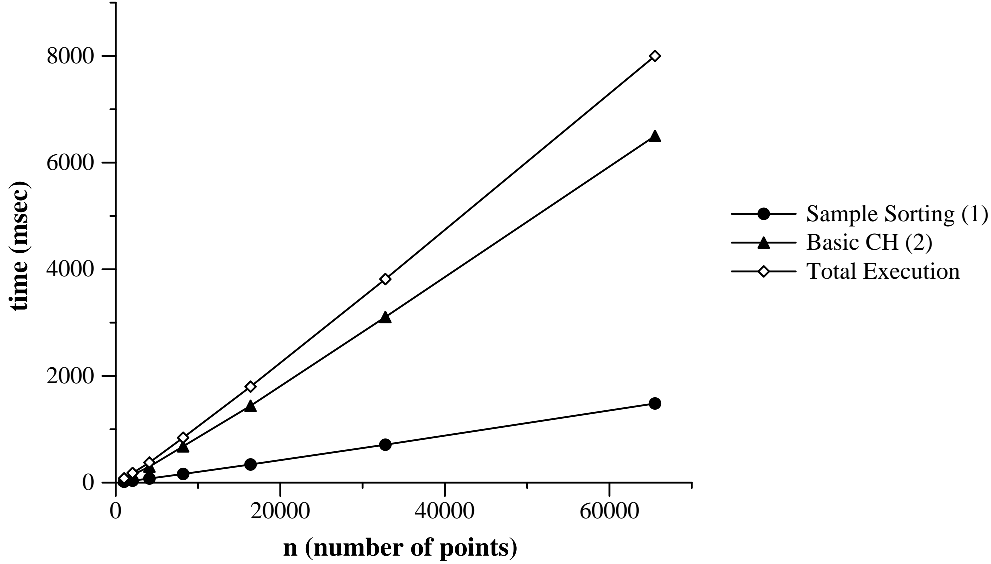


Figure 9: Execution Time for the Sequential Triangulation

### 4.3 Speedup and Communication/Computation Ratio

The speedup is the gain in execution time of the parallel implementation over the sequential one and is defined as the ratio of the two:

**Speedup** $= T_{sequential}/T_{parallel}$

The ideal speedup is equal to $p$, the number of processors, but this can never be achieved, since no problem is totally parallelizable.

In Figure 10, we show the speedup for the total execution time, as well as for its main component, sorting. The test was performed on pseudo-random generated input data of size 64K. The set size was limited by the amount of physical memory available on each processor. The experimental results strongly confirmed the theoretically estimated complexities from Section 3.5. For a small number of processors (n/p large), the speedup is very close to the ideal case, but starts degrading as the number of processors increases. This degradation would have started for larger numbers of processors, had we been able to run our test with larger sets of points. This happens because a given size of point set offers a limited potential of speedup through parallelization. As the number of processors increases (n/p small), the number of points per processor decreases below the threshold where the communication cost is justifiable by the local computation cost. Figure 11 illustrates this motivation showing the ratio of computation (expressed as the number of local points) over communication (expressed as the number of message exchanges) for the sample sort. For p=2, about 105 points are processed between any 2 consecutive message exchanges, while as p increases, this ratio falls to less than 50, making the parallelization less profitable.

### 4.4 Data Distribution Among Processors

In the beginning, input data are equally distributed among processors. Since a global sample sort is used in phases 1 and 4, the quality of sorting expressed by the bucket expansion (defined in Section 2.2) determines the data distribution. In turn, data distribution is an essential parameter in the distribution of computation among processors. We measured the bucket expansion at the end of sorting in phases 1 and 4 for an oversampling ratio 32 points (Figure 12). We found that sample sorting behaves very well, since for the input data sets we used, we always got a bucket expansion less than 1.5.

### 4.5 Work Distribution Among Phases

We analyzed the dependence of the distribution of execution times among the four phases on the number of processors (Figure 13). In the expected average case (considered as the case when the input points have pseudo-random generated x and y coordinates), as a result of the first two phases, most of the points fall within one of the basic convex hulls and therefore are triangulated locally. Thus, the remaining points which have to be triangulated within the funnel polygons in phase 4, represent only a small fraction of the initial input set. Therefore, phases 3 and 4 don't count too much in the total execution time (less than 0.05% for phase 4). In these cases, most of the work is split between the first two phases. For small p, basic convex hull computation and local triangulation dominates. This phase is mainly sequential work on a set of n/p points executed at

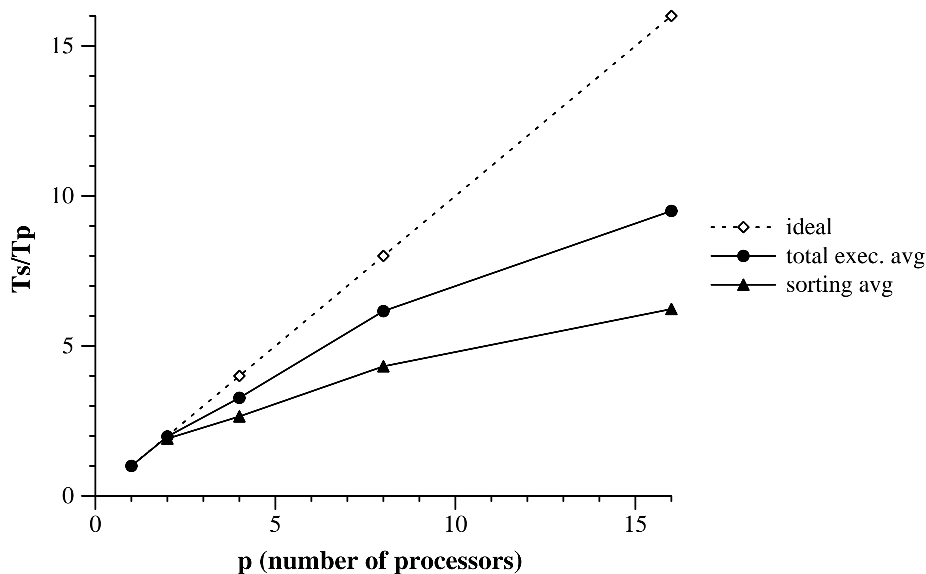


Figure 10: Speedup for expected average case

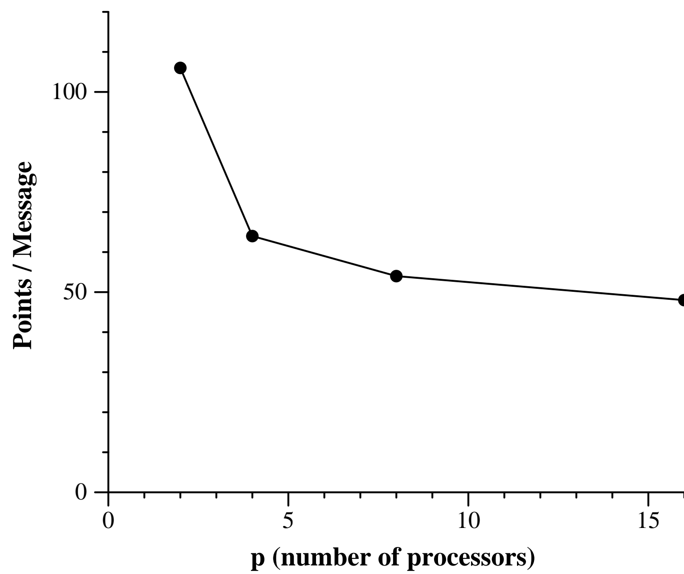


Figure 11: Number of points per message for sample sort in the expected average case

each processor. This is why for small p, the performance of the algorithm is very close to optimal. As the number of processors increases, the triangulation becomes dominated by the sorting phase which will further determine the overall performance behavior.

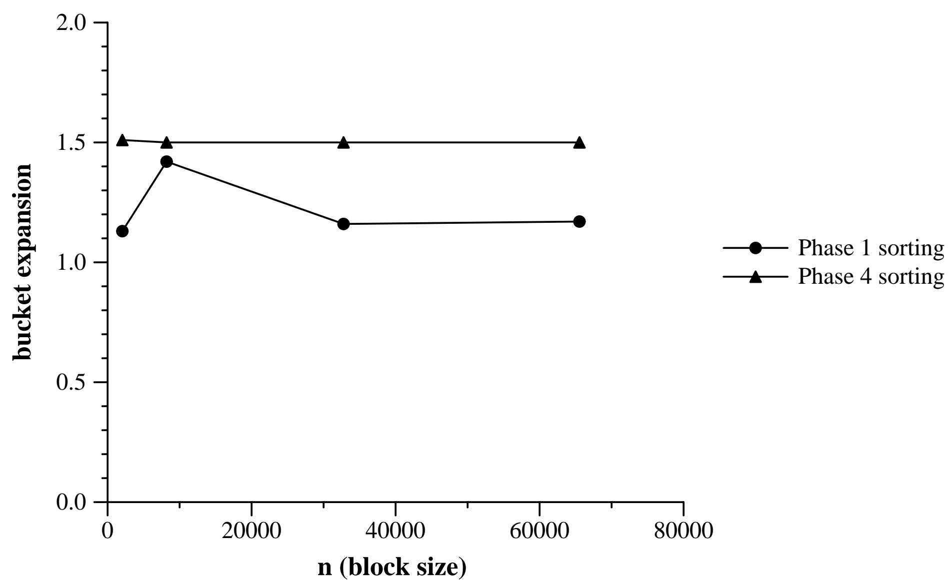


Figure 12: Bucket Expansion for Sample Sorting

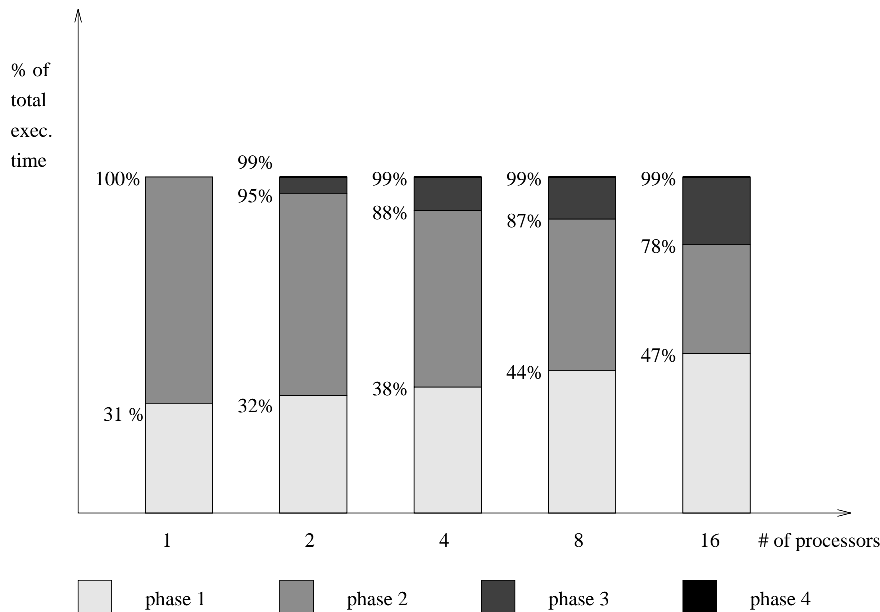


Figure 13: Distribution (I) of execution time among phases in the expected average case

### 4.6 Worst Case vs. Expected Average Case

The dependence of the execution time on the input size for the expected average case for 8 processors, is given in Figure 14. As n increases (n/p large) the speedup closely follows an nlogn dependence, proving a dominant sorting behavior, as estimated in Section 3.5. The 64K points are triangulated on 8 processors in about 1.7 sec while for 256k, 7.9 sec are needed. The distribution of the execution time among phases as a function of the input size is given in Figure 15. For the expected average case, the contribution of phase 2 increases and becomes dominant as the input size increases. This means that, as n/p increases and most of the points fall inside the basic convex hulls, the algorithm becomes mainly a sorting followed by a local triangulation inside those convex hulls.

The worst case running time of the algorithm corresponds to the case when most of the points are triangulated at phase 4. In order to observe the behavior of the algorithm in phases 3 and 4 for this case, we had to generate special input data sets : Points were generated uniformly on a number of circles corresponding to the number of processors p. For such distributions of the input data, all the points will lie on the basic convex hulls after phase 2. As a result, the all-tangent phase (phase 3) will have to work on almost the entire set of points. Thus, all the points will be triangulated in phase 4 instead of phase 2, since most of them are distributed in the funnel polygons. Nevertheless, this is not quite the worst case for phase 4 (a worst case would correspond to a single funnel polygon containing almost all the points).

As we can observe in Figure 18, about half of the execution time might be spent in phase 4. This is why it is very important for the overall performance of the algorithm to speedup the triangulation of the funnel polygons.

## 4.7 Message Size

The theoretical analysis of parallel algorithms considers messages as having unit cost. Therefore, the cost of communication becomes linear in the number of messages. As we have seen (Figure 8), real computers exhibit a different situation. Messages have a latency cost and a transmission cost and since the latency cost is significant, it pays to transfer, when possible, large sets of data instead of small ones, thus reducing the total number of messages.

We tested this hypothesis on the funnel polygon triangulation using the merging algorithm described in Section 3. We executed the algorithm with different message sizes (Figure 16 ). The performance significantly improves (by a factor of 3) when the message size increases from 3 to 50 points per bucket. As a consequence the contribution of phase 4 to the total execution time also decreases (to about 21%) and therefore the overall execution time reduces.

The same results apply for the size of the message used in sample sorting. An optimal size corresponds to a perfect balance between communication and computation.

Figure 17 gives a complete picture of the execution time as a function of a "worst case" input size for all four phases and total execution time on 4 processors using a message size of 50 points.

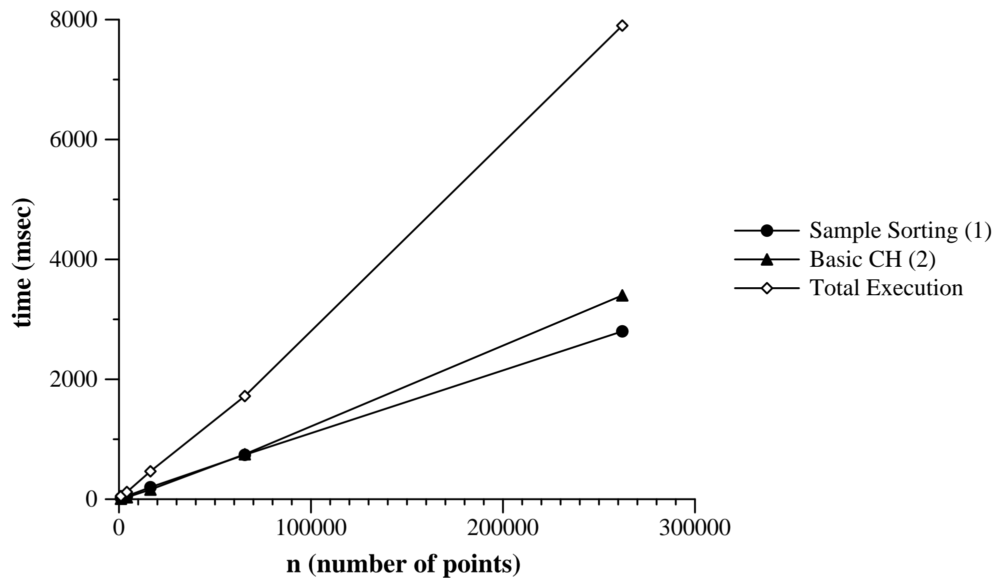


Figure 14: Execution time - expected average case

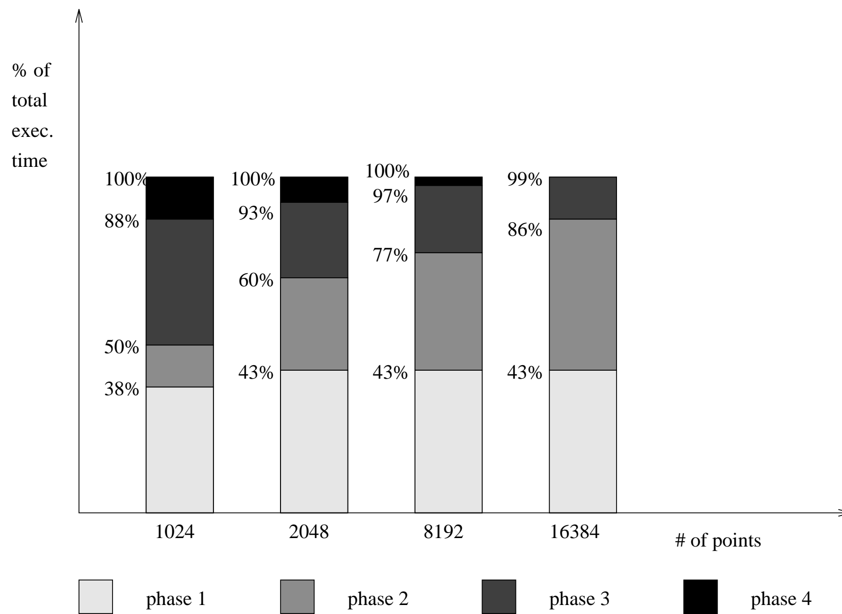


Figure 15: Distribution (II) of execution time among phases in the expected average case

The dependence is almost linear as the input size increases. The speedup is between linear and nlogn as n increases. The main contribution as the input size increases is given by the first two phases which is similar to the result we got for expected average case input. The only difference is an increased contribution of the funnel polygon triangulation, from insignificant in the expected average case (less than 0.05%) to approximately 21% in the worst case considered.

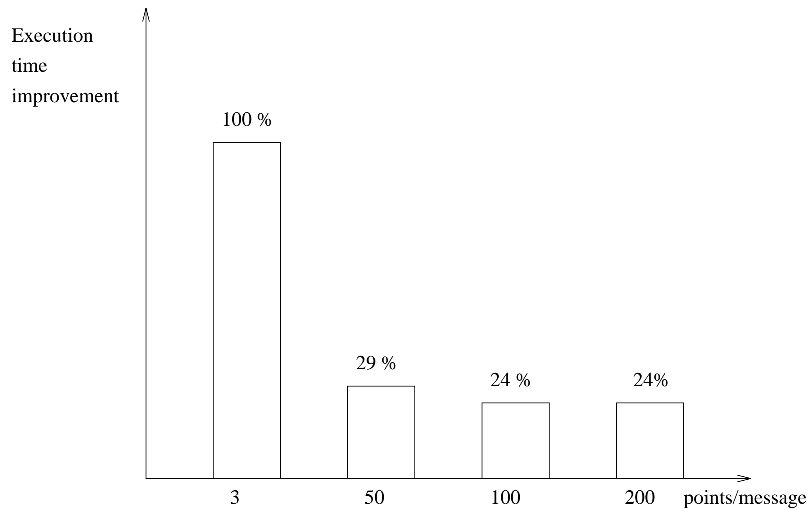


Figure 16: Cost of Funnel Polyon Triangulation for different message sizes

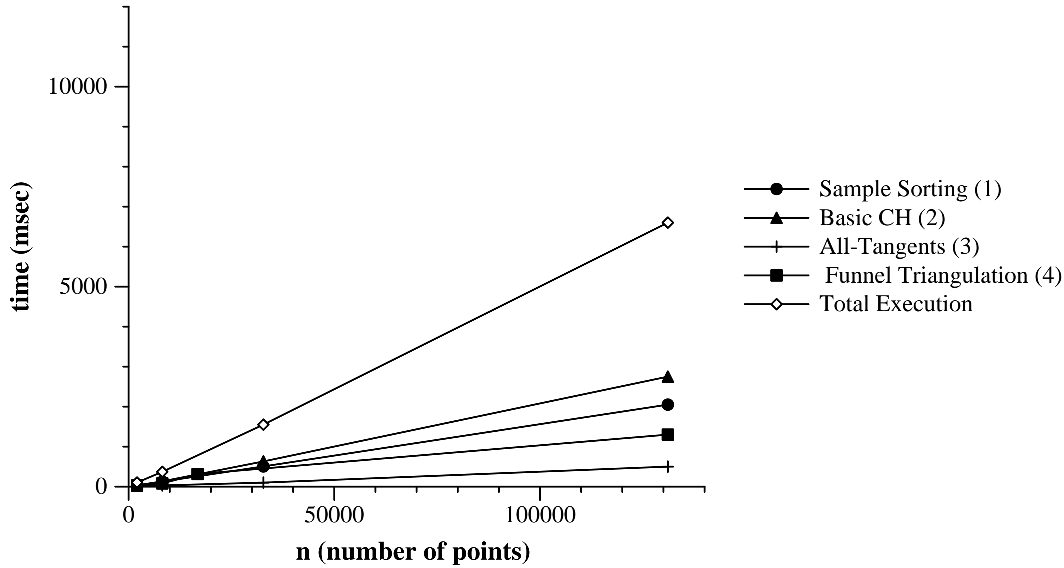


Figure 17: Execution time - worst case

## 4.8 Global Sorting applied to Funnel Polygon Triangulation

As we have mentioned before, the funnel polygon triangulation using the merging algorithm, may degenerate in an O(n) running time for a very "unlucky" distribution of points. A safe solution corresponds to a global sort as we described in Section 3.4.2.

We have also implemented the global sorting based algorithm for funnel polygon triangulation

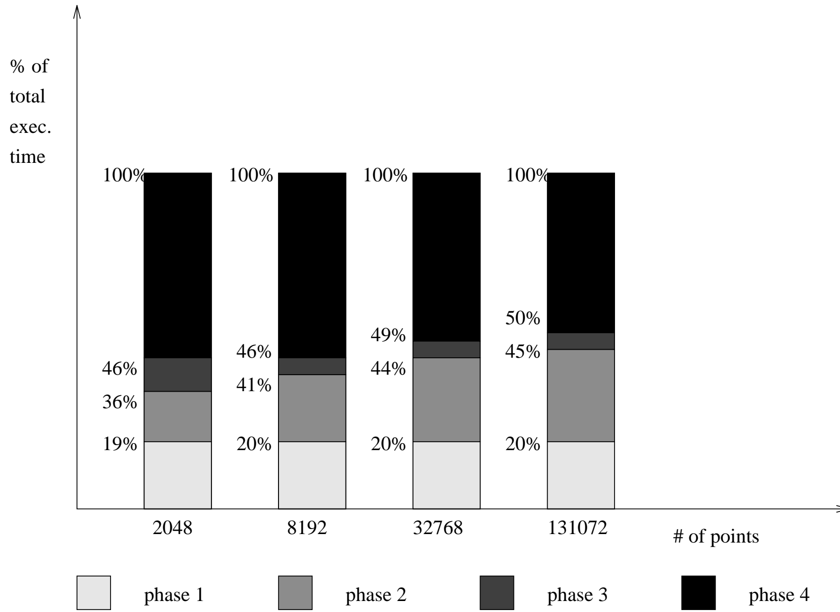


Figure 18: Distribution of execution time among phases in the worst case

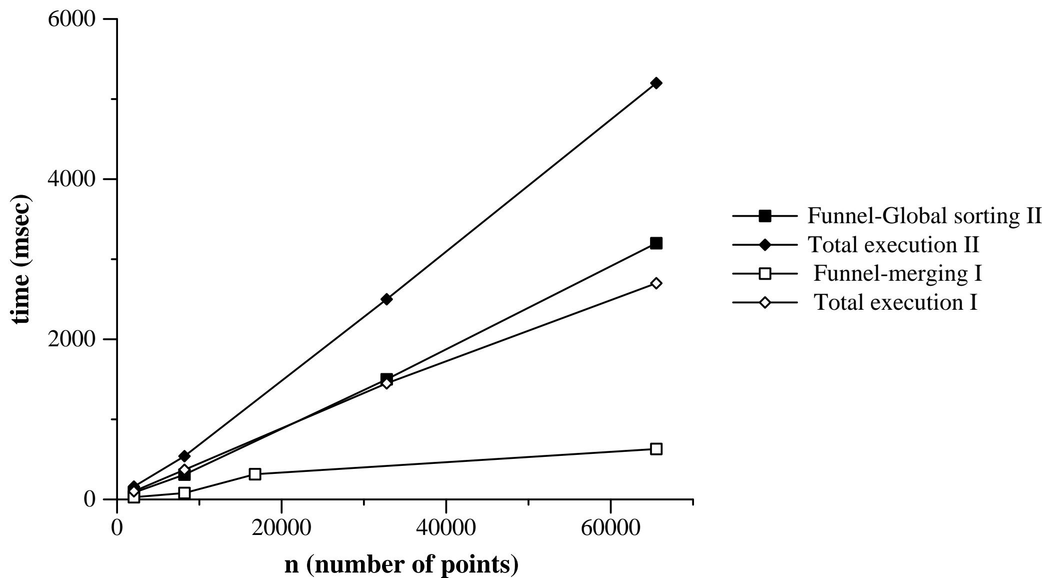


Figure 19: Cost of Funnel Polygon Triangulation using Global Sorting and Merging

to compare it with the first one (which uses the merging technique). We used the same distribution of points on circles, but this doesn't generate a worst case for this phase. In Figure 19 we show the execution time for the funnel polygon triangulation and the total execution time using the two methods. The global sorting method (II) has also an almost linear speedup but with larger constant factors than the merging method (I) with 50 points/message. Therefore, the contribution of phase 4 is less significant in the total execution time when using method I than when using method II. This happens because, when points are reasonably well distributed among funnel polygons, assigning one polygon to each processor leads to less communication compared with the one required for completely redistributing the points using the global sort. An analogy with the quicksort algorithm performance vs. heapsort performance can easily be made : a safe implementation for the funnel polygon triangulation, which must have a guaranteed performance for all input data must definitely use the global sorting algorithm; nevertheless, as long as the "pathological" cases (all points belonging to the same funnel polygon) are not expected or very rare, using the merging method for funnel polygon triangulation leads to the best algorithm for triangulation of a set of points.

We also noticed that global sorting used in phase 4 takes about 3 times more than the same method applied in phase 1. This is explained by the larger communication time required at phase 4, when a point carries about 3 times more information than the same point in phase 1.

# 5 Conclusions

In this study, we address the performance issues of triangulating a finite set of points in the plane, demonstrating the practical relevance of the CGM model for real implementations of parallel algorithms on multicomputers. A fully implementable algorithm for parallel triangulation is described in detail and the corresponding implementation on an iPSC/860 Hypercube was extensively tested under both expected average and worst case input data distributions.

To our knowledge, this is one of the few studies covering the gap between the design of parallel algorithms for solving a computational geometry problem, and their theoretical complexity analysis, and a real implementation and its performance evaluation on a parallel machine.

Our tests confirmed the conclusions of the theoretical analysis; for small n/p ratios, the communication cost dominates, reducing the speedup as p increases; for large n/p ratios, the total execution time is dominated by the computation cost, which mainly consists of a number of sorting-based phases. Therefore, for large n, the total execution time has an nlogn sorting-like asymptotic behavior.

Our reported measurements covered relatively small data sets, in the absence of virtual memory support, which anyway would have perturbed the timings significantly. It is obvious that for larger data sets we would have achieved better speedup.

The message size is a crucial implementation parameter, which can be tunned to reduce the impact of communication an thus improve the performance. We proved both theoretically and experimentally the positive effects of reasonably large messages over small ones.

We have also drawn experimental conclusions for two different methods of funnel polygon triangulation showing a less costly solution for expected average cases and a more expensive solution with guaranteed asymptotic performance in all cases.

# 6 Acknowledgements

I would like to thank my advisor, Diane Souvaine. She encouraged me to study Computational Geometry and start research work during the Summer and Fall '93 Independent Study courses at Rutgers University, when most of this work was accomplished.

I am also very grateful to Andrew Rau-Chaplin for the helpful discussions and advice. He dedicated much of his time directing my research with new ideas and helping me with many comments on this essay.

I thank Bernard Chazelle for giving me the freedom to extend this research through the final project for Spring' 94 Computational Geometry course at Princeton University.

# References


[1] Selim G. Akl and Kelly A. Lyons, *Parallel Computational Geometry, Prentice Hall, 1993.*

[2] F.P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction, Springer Verlag 1985.*

[3] C. A. Wang and Y. H. Tsin, *An O(log n) time parallel algorithm for triangulating a set of points in the plane, Information Processing Letters, Vol 25, 1987, 55-60.*

[4] E. Merks, *An optimal parallel algorithm for triangulating a set of points in the plane, International Journal of Parallel Programming, Vol 15, No. 5, 1986, 399-411.*

[5] P.D. MacKenzie and Q.F. Stout, *Asymptotically efficient hypercube algorithms for computational geometry, Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, College Park, Maryland, October 1990, 8-11.*

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

[7] A. Rau-Chaplin *On Parallel Data Structures and Parallel Geometric Applications for Multicomputers, Ph.D. Thesis, Carleton Univ., Ottawa, Canada, November 1992.*

[8] G.E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton S. J. Smith and M. Zhanga, *A Comparison of Sorting Algorithms for the Connection Machine CM-2.*

[9] K. Mehlhorn, *Data structures and algorithms 3: multidimensional searching and computational geometry, in EATCS Monographs in Theoretical Computer Science, W. Brauer, G. Rozenberg and A. Salomaa (Editors), Springer-Verlag, Berlin, 1984.*

[10] R. L. Graham, *An efficient algorithm for determining the convex hull of a finite planar set, Information Processing Letters, Vol 1,1972, 132-133.*

[11] R. Miller and Q. F. Stout, *Efficient parallel convex hull algorithms, IEEE Transactions on Computers, Vol. C-37, No. 12, December 1988, 1605-1618.*

[12] R. Miller and Q. F. Stout, *Mesh computer algorithms for computational geometry, IEEE Transactions on Computers, Vol C-38, No. 3, March 1989, 321-340.*

[13] J. O'Rourke, *Computational Geometry in C, Cambridge University Press, draft, 1993.*

[14] A. Aggarwal, B. Chazelle, L. J. Guibas, C. Ó'Dúnlaing and C. K. Yap, *Parallel computational geometry, Algorithmica, Vol 3, 1988, 293-327.*

[15] R. Cypher and C. G. Plaxton, *Deterministic sorting in nearly logarithmic time on the hypercube and related computers, Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computation, Baltimore, May 1990, 193-203.*

[16] X. Guan and M. A. Langston, *Time-Space optimal parallel merging and sorting, International Conference on Parallel Processing, 1989.*

[17] M. J. Atallah and M. T. Goodrich, *Efficient parallel solutions to some geometric problems, Journal of Parallel and Distributed Computing, Vol 3, 1986, 492-507.*

[18] R. Cole and M. T. Goodrich, *Optimal Parallel Algorithms for Point-Set and Polygon Problems, Algorithmica, Springer-Verlag, Vol. 7, 1992, 3-23.*

[19] R. Cypher and J. L. S. Sanz, *Optimal Sorting on reduced architectures.*

[20] M. Goodrich and M. Atallah, *Efficient Plane Sweep in Parallel, Proc. of the Second Symp. on Computational Geometry, York Heights,(June 1985).*

[21] H. ElGindy, *An Optimal Parallel Algorithm for Triangulating Simplicial Points Sets in Space, Internal Report, University of Pennsylvania (April 1986).*

[22] A. Yao, *A lower bound to finding convex hulls, J. Assoc. Comput. Mach. 28 (1981), 780-787.*