1995

Parallel Algorithms for Constructing Convex Hulls.

Jigang Liu
Louisiana State University and Agricultural & Mechanical College

PARALLEL ALGORITHMS FOR CONSTRUCTING CONVEX HULLS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Jigang Liu
B.S., Beijing University of Aeronautics and Astronautics, 1982
August, 1995

Acknowledgements

I would like to express my heartfelt appreciation to Professor Doris L. Carver for her invaluable encouragement, support, guidance, and insight throughout the course of my studies and research at Louisiana State University.

I would also like to express my gratitude to my advisory committee members, Professors S. Sitharama Iyengar, Donald H. Kraft, J. Bush Jones, David C. Blouin and Jimmie D. Lawson for their persistent interest and constructive advice.

I am very grateful to Professor S. Q. Zheng for his generous advice and support of this research.

I am indebted to my parents, Mr. Yi Liu and Mrs. Jing Zhang, and my parent-in-laws, Mr. Yisan Wu and Mrs. Baicong Liu for their devoted support, endless love and selfless sacrifice.

I, especially, would like to extend my sincere gratitude and constant love to my wife, Xiaofei Wu, and daughters, Shuang and Yang. Without their love, support, sacrifice, and understanding, I would not have been able to complete this work.

Table of Contents

List of Tables

List of Figures

Abstract

For a given set of planar points S, the *convex hull* of S, $CH(S)$, is defined to be a list of ordered points which represents the smallest convex polygon that contains all of the points. The convex hull problem, one of the most important problems in computational geometry, has many applications in areas such as computer graphics, simulation and pattern recognition.

There are two strategies used in designing parallel convex hull algorithms. One strategy is the divide-and-conquer paradigm. The disadvantage to this strategy is that the recursive merge step is complicated and difficult to implement on current parallel machines. The second strategy is to parallelize sequential convex hull algorithms. The algorithms designed using the second strategy are often iterative algorithms which can be more easily implemented on the current parallel machines.

This research focuses on designing parallel convex hull algorithms using the second strategy because we intend to facilitate the implementation of the newly designed algorithms on massively parallel machines. We first design a sequential algorithm for constructing a convex hull of a simple polygon, which is a special case of a set of planar points. This optimal algorithm is extended to handle a set of planar points without increasing the time complexity. Next, the sequential algorithm is converted for linear array and two or more dimensional mesh-array architectures. The algorithms for the case where the number of points is greater than the number of processors is also addressed. Each of the algorithms developed is optimal. To analyze the performance of the algorithms compared to previous algorithms, a system called the *Parallel Convex Hull Simulation System* was developed. The results of the analysis indicate that the new algorithms exhibit better performance than previous algorithms.

Chapter 1

Introduction

Computational geometry is a branch of computer science which is devoted to the design and analysis of algorithms for solving geometric problems. It is a recent field[S78] of theoretical computer science that has developed rapidly.

The field of geometric algorithms, with its rich historical context and numerous new fundamental algorithms, is important for large-scale applications, such as computer graphics, pattern recognition, robotics, statistics, database searching, and the design of very large scale intergrated circuits.

Interest in parallel algorithms for geometric problems has grown in recent years because parallelism seems to hold the greatest promise for major reductions in computation time. The idea is to use several processors which cooperate to solve a given problem simultaneously in a fraction of the time taken by a single processor. Parallel machines are classified as either processor networks machines where an interconnected set of processors cooperate to solve a problem by performing local computations and exchanging message or parallel random access machines where a common memory is used as a bulletin board and any pair of processors can communicate through this shared memory[AL93]. The modern approach to parallel computational geometry was pioneered by A. Chow[C80].

The goal of this research was to develop improved sequential and parallel algorithms for solving the convex hull problem. After we give an introduction to computational geometry in section 1, we define the convex hull problem in section 2. In section 3, previous sequential and parallel algorithms are reviewed. Finally, the outline of this dissertation is presented in section 4.

1.1 Research in Computational Geometry

Geometry is a major branch of mathematics with important theory and applications. The treatises of ancient mathematicians form the basis of efficient algorithms on geometry. Computational geometry is a field of computer science since it concerns how to solve geometric problems with computers. In general, the main problems studied in computational geometry are [AL93]:

Convex Hull Given a finite set of points in the plane, find their *convex hull* (i.e., the convex hull polygon with smallest area that includes all the points, either as its vertices or as interior points).

Segment Intersection Given a finite set of line segments in a plane, find and report all pairwise *intersections* among line segments, if any exist.

Geometric Search Given a *convex planar subdivision* (i.e., a convex polygon itself partitioned into convex polygons) and a finite set of data points, determine the polygon of the subdivision occupied by each data point.

Visibility and Separability Given a simple polygon P and a point p inside P, determine that region of P that is *visible* from p (i.e., the region occupied by points q's such that the line segment with endpoints p and q does not intersect an edge of P).

Nearest Neighbors Given a finite set of points in the plane, determine which two points are *closest* to one another.

Voronoi Diagram Given a finite set S of data points in the plane, for each point p of S, find the *region* of the plane formed by points that are closer to p than to any other point of S.

Geometric Optimization Given $2n$ points in the plane, match each point with exactly one other point so that the sum of the Euclidean distances between matched points is as small as possible.

Polygon Triangulation Given a simple polygon P, *triangulate* P (i.e., connect the vertices of P with a set of chords such that every resulting polygonal region is a triangle).

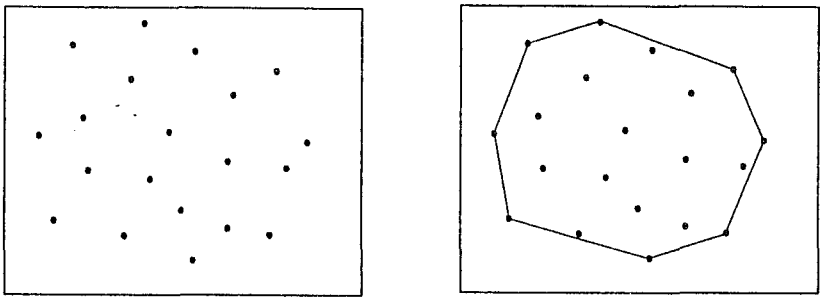
The computation of the convex hull is a central problem in computational geometry. It has been vastly studied, not only because of its practical applications, such as computer graphics and statistics, but also because other computational geometry problems start with the computation of a convex hull. The *farthest-pair problem* is one of the examples. For a given set of n points in the plane, the farthest-pair problem is determining the two points with the maximum distance from each other. Given a fully connected graph of n points, there are $n(n-1)/2$ edges. Thus, to determine which edge has the maximum length costs $O(n^2)$ time at most. This time complexity can be improved if the convex hull of the n points is constructed first because the farthest pair of vertices of an n-vertex convex hull can be found in $O(n)$ time (It is easy to prove that these two points must be vertices of the convex hull). Therefore, since the time complexity for constructing the convex hull of a set of n planar points is $O(nlogn)$, the time complexity for finding the farthest pair of points in a given set of n points is $O(nlogn)$.

1.2 Conventional Convex Hull Algorithms

The first convex algorithm was proposed by Graham in 1972 [G72]. Variations of the algorithms have been presented in [J73, K77, E77, B78, F79]. The convex hull algorithm was summarized and Graham's algorithm was refined by Andrew in [A79]. The convex hull problem is formally defined in [P88] as

> *Convex Hull Problem*: Given a set S of N points in E^d, construct its convex hull (that is, the complete description of the boundary $CH(S)$), where E^d represents the *Euclidean* space of d dimension and $CH(S)$ indicates the convex hull of the set S.

Given a set of n points S in a plane, the *two – dimensional* or *planar convex hull problem* is to find the smallest subset P of S such that the points in P are the vertices of a convex polygon and every point in S is contained in the convex polygon defined by P. The *convex hull* itself is the boundary of the polygon defined by P, and the points in P are the extreme points of the convex hull of S. Some points can lie on the convex hull polygon, but they are considered as interior points and not extreme points. A given set of planar points and its corresponding convex hull is shown in Fig. 1.1.



(a) A set of planar points (b) The convex hull of (a)

Fig. 1.1 A set of planar points and its corresponding convex hull

A convex hull algorithm can be used to sort a list of n real numbers x_i. If one projects them onto the parabola $y = x^2$, then all the points (x_i, x_i^2) will be corners of their convex hull and appear in sorted order. Thus the lower bound for the planar convex hull problem is stated as the following theorem[PS88].

> **Theorem** : Sorting is linear-time transformable to the convex hull problem; therefore, finding the ordered convex hull of N points in the plane requires O ($NlogN$) time.

Traditionally, two steps are required to construct the convex hull of a finite set: 1) identify the extreme points; 2) order these points so that they form a convex polygon. An extreme point is referred to as a point on the boundary of a convex hull. In other words, an extreme point is a vertex of a convex hull. An early convex hull algorithm was designed based on the following theorem [G72]:

Theorem: A point p fails to be an extreme point of a plane convex set S only if it lies in some triangle whose vertices are in S but is not itself a vertex of the triangle.

This theorem tells us how a non-extreme point can be identified. If we can identify all of the non-extreme points from a set and then eliminate such points from the set, only extreme points remain. For a given set of n planar points, there are $\binom{n}{3}$ possible triangles. Therefore determining whether a point is inside one of $\binom{n}{3}$ triangles costs $O(n^3)$ time. Since there are n points in S, the complexity of the algorithm based on the above theorem is $O(n^4)$. This algorithm is conceptually simple but it is extremely inefficient. By using $O(n^4)$ time, we only determine the extreme points. Those extreme points must then be sorted in order to obtain the convex hull.

In [J73], an algorithm to compute the convex hull which starts with the lowest point p (the point with the smallest y-coordinate) is given. The point p is then used as the origin to measure the polar angles with the remaining $n-1$ points. Suppose that points p and q form the smallest polar angle among the measured $n-1$ polar angles. Then point q determines as an extreme point of the hull. Next, point q is used as the origin to measure the polar angles with the remaining $n-1$ points. The point which forms the smallest polar angle with q is the next extreme point on the hull. The algorithm is terminated when p is picked again as an extreme point. Since it costs n time to determine an extreme point on the hull and the maximum number of the extreme points in a set of n planar points is n, the time complexity of the algorithm is $O(n^2)$. This algorithm is referred as Jarvis' March.

In [G72], the first convex hull algorithm which uses a sorting step is proposed. Based on a set of sorted points, extreme vertices of the convex hull can be constructed in linear time. The algorithm, often referred to as Graham's scan, is presented as follows:

1) find an internal point q of S, where S is a set of planar points;

2) order the remaining $n-1$ points of S according to their polar angles;

3) trace the ordered points generated in step 2) to eliminate non-extreme points.

The first step of the Graham scan costs $O(n)$ time at most. Since the complexity of sorting is $O(nlogn)$, the second step uses $O(nlogn)$ time. The last step costs $O(n)$. Therefore, the time complexity of Graham's algorithm is $O(nlogn)$, which is optimal.

Although Graham's scan algorithm is an optimal algorithm, there are avenues for improvement. In [A79], the first step of the Graham's scan is removed by ordering the points according to their x-coordinates rather than the polar angles. The construction of the convex hull is also partitioned into a problem of building two sub-hulls, the upper hull and the lower hull, separated by the left most and the right most points in the set. The algorithm [A79] is described as follows:

Given a set of n points in the plane, first determine its left and right extreme points, u_{small} and u_{large}, respectively. Then draw the line passing through extreme points u_{small} and u_{large}. Next, partition the remaining points into two subsets depending upon whether they lie below or above this line. The lower subset will give rise to a polygonal chain (lower-hull or L-hull) which is monotone with respect to the x-axis; similarly, the upper subset gives rise to an analogous chain (upper-hull or U-hull). After the two sets are determined, order the points in a set by increasing the x-coordinates. Then apply step 3) of Graham's scan to construct the corresponding chain. Finally, concatenate these two chains to form the convex hull. This algorithm

avoids the trigonometric operations which are used in Graham's algorithm and it does not require finding an internal point to start the algorithm.

Graham used the order relationship of polar angles of points and Andrew used the order relationship of x-coordinates of points. The order relationship of slopes of points is first used in [LZ92]. If a set of planar points is given as a simple polygon, the ordering relationship of the slopes of the points can be used to simplify the calculation of the corresponding convex hull.

A simple polygon is defined as a polygon in which no two consecutive edges share a vertex. In other words, a simple polygon partitions the plane into two disjoint regions, the interior (bounded) and the exterior (unbounded) that are separated by the polygon. In order to use the ordering relationship of the slopes of points, the construction of the convex hull is partitioned into four subproblems. In other words, a given simple polygon is partitioned into four chains by its left most point, right most point, highest point and lowest point. The use of four partitions rather than two partitions of the problem makes the algorithm simpler and more efficient than the algorithms presented in [GY83] [L83]. Although the algorithm is designed for the simple polygon problem, it has strong impact on the issue of parallelism of the algorithm for a general case. A summary of the sequential convex hull algorithms is shown in Table 1.1.

1.3 Parallel Convex Hull Algorithms

Parallel solutions of the convex hull problem were introduced in the early 80's. A summary of the parallel work is given in [AL93].

The design of parallel algorithms mainly depends on the model type, that is the shared memory model or the network model. With the shared memory model, CRCW(Concurrent Read and Concurrent Write), CREW(Concurrent Read and Exclusive Write) and EREW(Exclusive Read and Exclusive Write) are different architectures that can be used. In the network model, architectures are roughly classified as

Table 1.1 A summary of the sequential convex hull algorithms

Year	Author		Time Complexity	Characteristics
1972	R. L. Graham	[G72]	*O(nlogn)*	Polar Angels
1972	J. Sklansky	[S72]	*O(Cn)*	For a Simple Polygon
1973	R. A. Jarvis	[J72]	*O(nm)*	Polar Angles
1977	W. F. Eddy	[E77a,b]	$O(n^2)$*worse case*	Idea of Quick Sort
1977	F. P. Preparata S. J. Hong	[PH78]	*O(nlogn)*	Divide-and-Conquer
1978	K. R. Anderson	[A77]	*O(nlogn)*	sines and cosines
1978	J.Koplowitz D. Jouppi	[KJ77]	*O(nlogn)*	$2N - M + 1$ iterations
1978	J. L. Bentley M. I. Shamos	[BS78]	*O(nlogn)* *worse case*	For a Large N, Divide and Conquer
1978	A. Bykat	[B78]	$O(n^2)$*worse case*	Divide-and-Conquer, turn
1979	A. M. Andrew	[A79]	*O(nlogn)*	x-coordinates
1983	R. L. Graham F. F. Yao	[GY83]	*O(n)*	For a Simple Polygon (Half partition)
1983	D. T. Lee	[L83]	*O(n)*	For a Simple Polygon (Half partition)
1992	J. Liu S. Q. Zheng	[LZ92]	*O(n)*	For a Simple Polygon (Quarter partition)

linear array, mesh connected array, and hypercube. Due to the different models and architectures within models, it is difficult to compare parallel algorithms in a straightforward manner. But if attention is focused on the design approaches, the algorithms for the convex hull problem can be classified into two groups: 1) recursive algorithms and 2) iterative algorithms.

The recursive algorithms are based on the divide-and-conquer paradigm. Basically, the design approach consists of three steps. First, the problem is partitioned into subproblems. Second, the subproblems are solved recursively. Finally, the subsolutions resulting from the subproblems are merged into the solution for the whole problem.

In order to write parallel algorithms based on the divide-and-conquer approach, a model of computing must first be selected. The first question that needs to be answered is how to assign the problem or subproblems to the computer. Many authors have developed modified strategies for the partition required in the first step. Other authors have emphasized the last step by improving the method for merging the subsolutions.

In [C81], two partition strategies are proposed. The first strategy is a half-size partition which means that the whole problem is divided into two subproblems that are equal in size. For the CCC(Cube Connected Cycle) model, the algorithm runs in $O(\log^2 n)$ time and uses O(n) processors. The second strategy is to partition the problem into $n^{1-1/k}$ subproblems, where $1 \leq k \leq \log n$. Then the algorithm costs O(klogn) on time with $O(n^{1+1/k})$ processors for the same computing model.

A different partition strategy is taken in [AG86], and independently in [ACGOY88]. In these two algorithms, the problem is divided into $n^{1/2}$ subproblems and O(logn) running time and O(n) processors are reached based on the CREW PRAM model for both algorithms.

In [MS89], the problem is subdivided into $n^{1/4}$ subproblems so that each subproblem has $n^{3/4}$ points. The performance of the algorithm is O(logn) in running time with O(n) processors for a weaker EREW PRAM model.

The main approach for the merge step is the calculation of the common tangents. The differences on the merge step among algorithms are the methods for determining the tangents as well as the methods for finding the smallest tangents. For example, in [AG86] the common tangents are calculated by applying the sequential algorithm of [O81]. In [MS89] the "slope records" are used to compute the common tangents. After determining the convex vertices in each of subhulls, the algorithms use parallel prefix computation to compress those subsolutions into one solution.

The iterative algorithms are based on the approach of parallelization of existing sequential algorithms. A typical example, found in [A82], [A89], uses the approach of Jarvis' March presented in [J73]. Akl's algorithm [A89] uses the CRCW PRAM model with $O(n^2)$ processors in constant time.

Besides Akl's algorithm, Chazelle proposed an algorithm [C84] based on the observation similar to the Jarvis' march [J73]. Holey and Ibarra's algorithms [HI92] are the latest iterative parallel convex hull algorithms based on the idea of the Graham scan [G72]. Chazelle's algorithm is considered as the first linear array convex hull algorithm and it is designed for a dynamic situation in which the complete set of points is not available when the algorithm is started. A point v is pumped from the leftmost processor of the linear array. It travels from left to right and determines whether the smallest convex wedge, centered at v, contains all the points encountered so far. If it does, v stops at the first vacant processor on the right-hand side. Otherwise, it is deleted. Since the above operation leaves those extreme points in an arbitrary order, a second linear array is needed to order those extreme points in either a clockwise or counterclockwise order. This two-linear-array algorithm was converted

into a one-linear-array algorithm by Chen [CCL87] for a static situation in which all points of S are available before the execution of the algorithm. But the algorithm needs to be run twice. Both algorithms use $O(n)$ time with $O(n)$ processors, which are optimal.

In [HI92], Holey and Ibarra presented convex hull algorithms for mesh-connected arrays under different input/output and communication assumptions. In particular, these mesh-connected arrays are one-way iterative linear arrays, one-way cellular linear arrays, and two-way d-dimensional cellular arrays. They showed that the time complexities of all their algorithms are optimal within a constant factor. For all the architecture models discussed, they considered the static version of the convex hull problem. The dynamic version of the problem is also considered, which supports maintaining the convex hull of a dynamic point set that is defined by a sequence of point insertion and deletion operations, processing queries of whether or not a given point is inside the current convex hull, and reporting all hull points of the current convex hull. All their algorithms assume that $n = O(p)$, where p is the number of processors, and n is the number of points. In Table 1.2, the parallel convex hull algorithms are listed according to their architectures.

1.4 Outline of Dissertation

This research is based on the iteration strategy of designing parallel convex hull algorithms. In Chapter 2, we propose a new method for constructing the convex hull of a simple polygon. We then extend the algorithm for the simple polygon problem to a general case where a set of planar points is given.

In Chapter 3, an Odd-Even parallel convex hull algorithm based on the algorithm given in Chapter 2 is presented. When the ordering relationship of the slopes of points is used with the well known Odd-Even transposition sort method, an Odd-Even convex hull algorithm is established. The new algorithm offers advantages over the

Table 1.2 A summary of parallel convex hull algorithms

EREW PRAM Model

Year	Authors	Processors	Running time
1980	D. Nath , S. N. Maheshwari, P. C. P. Bhatt	$O(n^{1+1/k})$	$O(K\log n)$
1984	S. G. Akl	$O(n^{1-\varepsilon})$	$O(n^{\varepsilon}\log n)$
1988	R. Miller & Q. F. Stout	$O(n)$	$O(\log n)$

CREW PRAM Model

Year	Authors	Processors	Running time
1980	D. Nath , S. N. Maheshwari, P. C. P. Bhatt	$O(n^{1+1/k})$	$O(K\log n)$
1981	A. L. Chow	$O(n)$	$O(\log^2 n)$
1981	A. L. Chow	$O(n^{1+1/k})$	$O(K\log n)$
1986	M. J. Atallah & M. T. Goodrich	$O(n)$	$O(\log n)$
1988	A. Aggarwal, B. Chazelle, L. J. Guibas, C. O'Dunlaing, C. K. Yap	$O(n)$	$O(\log n)$
1988	R. Cole & M. T. Goodrich	$O(n)$	$O(\log n)$

CRCW PRAM Model

Year	Authors	Method	Processors	Running time
1982	S. G. Akl	AND	$O(n^3)$	$O(1)$
1988	Q. F. Stout	COLLISION	$O(n)$	$O(1)$ *expected*
1989	S. G. Akl	AND, Smallest	$O(n^2)$	$O(1)$

LINEAR ARRAY

Year	Authors	Structures	Processors	Running time
1984	B. Chazelle	Two Arrays	$O(n)$	$O(n)$
1987	G. H. Chen, M. S. Chern, & R. C. T. Lee	Array Ring	$O(n)$	$O(n)$
1992	J. A. Holey & O. H. Ibarra	Two-way Array	$O(n)$	$O(n)$

MESH

Year	Authors	Structures	Processors	Running time
1984	R. Miller & Q. F. Stout	Mesh	$O(n)$	$O(n^{1/2})$
1989	R. Miller & Q. F. Stout	Mesh	$O(n)$	$O(n^{1/2})$
1989	S. G. Akl	Mesh-of-trees	$O(n^2)$	$O(\log n)$

CUBE

Year	Authors	Structures	Processors	Running time
1981	A. L. Chow	CCC	$O(n)$	$O(\log^2 n)$
1981	A. L. Chow	CCC	$O(n^{1+1/k})$	$O(K\log n)$
1988	R. Miller & Q. F. Stout	Hypercube	$O(n)$	$O(\log n \log\log n)$
1988	I. Stojmenovic	Hypercube	$O(n)$	$O(\log^2 n)$
1991	F. T. Leighton	Hypercube	$O(n)$	$O(\log n \log\log n)$
1992	J. A. Holey & O. H. Ibarra	Hypercube	$O(n)$	$O(\log^2 n)$

previous algorithms. To illustrate how the algorithm works, an example is included in this chapter.

To generalize the Odd-Even convex hull algorithm proposed in Chapter 3, we discuss two general cases in Chapter 4. The first case is the case where the number of the points is greater than the number of processors. The second case is the case where a two or more dimensional mesh-array architecture is given. For both cases, the algorithms are designed and the complexity of the algorithms are analyzed.

This research focuses on both the design and the implementation of algorithms. In Chapter 5, the performance analysis of both the Odd-Even parallel convex hull algorithm and Holey's algorithm is conducted. Two data collecting strategies, random data collection and designed data collection, are developed for analyzing the performance of the speed-up of the algorithms. In order to perform the analysis of the performance of the parallel algorithms, four testing data collecting methods are introduced. These methods are called "time"-related, "point"-related, "size"-related and "communication"-related. The performance analysis is discussed from a statistical point of view.

The system used for the performance analysis of the algorithms is called *Parallel Convex Hull Simulation System.* There are three major components in the system. One component is the point collection and hull display sub-system which includes the user interface, the input methods and output methods of the system. The second component is the hull calculation sub-system, which is located on a remote parallel machine. We describe the characteristics and communication mechanisms of the parallel machine to illustrate how to implement the parallel convex hull algorithms on the parallel machine. The third component is the communication sub-system. Since the point collection and hull display sub-system are not located physically on a single machine with the hull calculation sub-system, the communication between the two sub-systems

is crucial. The system is based on the client/server paradigm and is written in MPL(Maspar Programming Language), Motif/X-window and TCP/IP on the Maspar/Ultrix for the server and IBM RS/6000/AIX for the client.

In Chapter 6, we present conclusions of the research, a summary of the significance of this research, and a discussion about future research areas.

Chapter 2

A New Sequential Convex Hull Algorithm

We introduce a new method to construct the convex hull for a set of planar points. In Section 2.1, a brief introduction to the problem is given. Previous algorithms are discussed in Section 2.2. In Section 2.3, we propose a new approach to design a convex hull algorithm for a simple polygon. The analysis and comparison are given in section 2.4. Finally, a summary are presented in Section 2.5.

2.1 Introduction

A simple polygon is defined as a polygon in which no two nonconsecutive edges share a point [PS88]. In other words, a simple polygon partitions a plane into two disjoint regions, the *interior* (bounded) and the *exterior* (unbounded) that are separated by the polygon (*Jordan curve theorem*).

Given a simple polygon P in the plane represented by a list of vertices in the order that they appear on P, finding the convex hull of P requires at least $\Omega(N)$ time, where N is the number of vertices in P. This problem is referred to as *CHSP* (convex hull for simple polygon). Several $O(N)$-time algorithms for *CHSP* have been proposed [S72, S78, MA79, GY83, L83]. Bykat [B78] showed that the algorithm given by Sklansky in [S72] does not always work, and the algorithm proposed by Shamos in [S78] also fails in some special cases. The algorithm given by McCallum and Avis in [MA79] uses two stacks and is quite complicated. Graham & Yao [GY83] and Lee [L83] independently developed simpler algorithms using one stack. Their algorithms are similarly based on *problem decomposition*. In their algorithms, the *CHSP* is split into two subproblems where each subproblem constructs a halfhull by a linear scan of a chain of vertices of P. During the scanning process, reference points and lines are introduced to assist in *decision-making*, namely, what vertices of P should be kept for

possible inclusion in $CH(P)$ (convex hull of P) and what vertices should be rejected for further contention. We present an algorithm, inspirited by the algorithms given in [GY83] and [L83], which constructs the convex hull of a simple polygon by constructing four component convex chains, called *quarterhull*. We show that our algorithm is not only much simpler conceptually than any of the previous algorithms, but also more efficient in terms of number of operations needed in comparison with the algorithms given in [GY83] and [L83].

2.2 Previous Algorithms

In this section we briefly describe the algorithms given in [GY83] and [L83], the simplest among all known algorithms for *CHSP*. Since these two algorithms are almost the same, we refer to them as one generic algorithm, the *original single-stack* (*OSS_HULL*) algorithm.

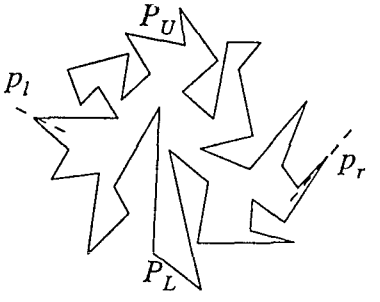


Fig. 2.1 Two chains P_U and P_L of a simple polygon.

Given a simple polygon P represented by a list of its vertices in clockwise order, let p_l be a vertex with the smallest x-coordinate value and p_r be a vertex with the largest x-coordinate value. Then, we can obtain two chains P_U and P_L of vertices of P, as shown in Figure 2.1 (note: p_l and p_r belong to both chains and P_U contains a vertex with the largest y-coordinate value among all vertices of P). These two chains may be represented by queues. The original single-stack algorithm, *OSS_HULL*, finds the convex hull $CH(P)$ of P by finding two halfhulls, *upper-hull* and *lower-hull* of P,

which are denoted respectively by $UH(P)$ and $LH(P)$. The $UH(P)$ and $LH(P)$ are constructed from P_U and P_L respectively.

The OSS_HULL algorithm is:

Algorithm *OSS_HULL*(P)

Find the extreme points p_l and p_r

Find two chains P_U and P_L of P;

Call *HALFHULL*(P_U, p_l, p_r) to obtain $UH(P)$;

Call *HALFHULL*(P_L, p_r, p_l) to obtain $LH(P)$;

Concatenate $UH(P)$ and $LH(P)$ to obtain $CH(P)$;

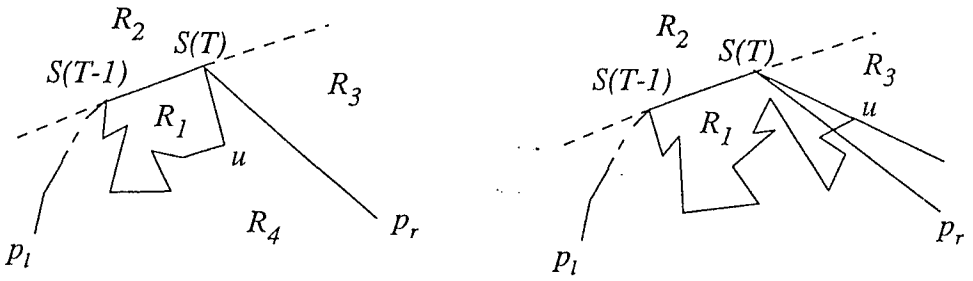
end *OSS_HULL*

Procedure *HALFHULL*(Q , p)

$u := p$;
PUSH(Q, p);
$q :=$ *DEQUEUE*(Q) ;
PUSH(S, q);
while Q is not empty **do**
begin
$q :=$ *DEQUEUE*(Q) ;
if $S(T-1)S(T)q$ is right turn **then** /* q is in the region 1 or 3 or 4 */
if $uS(T)q$ is right turn **then** /* q is in the region 3 or 4 */
if $p_M\ S(T)q$ is right turn **then** /* q is in the region 3 */
PUSH(S, q);
else /* q is in the region 4 */
while $(S(T)\ p_M$*FRONT*$(Q))$ is right turn **do**
$u := q$;
$q :=$ *DEQUEUE*(Q);
else /* q is in region 1 */
while $(S(T-1)S(T)$*FRONT*$(Q))$ is right turn **do** /* *FRONT*(Q)

```
            is still in region 1 */
              u := q;
              q := DEQUEUE(Q);
          else /* q is in region 2 */
            begin
              while (qS(T)S(T − 1)) is right turn do
                POP(S);
              endwhile
              PUSH(S, q);
            end
        end /* q is discarded if it is in region 1 or 4 */
      endwhile
      return(S); /* Return the convex hull */
    end HALFHULL
```

$LH(P)$ is constructed by calling procedure $HALFHULL(P_L, p_r, p_l)$. Similarly $UH(P)$ is constructed by calling procedure $HALFHULL(P_U, p_l, p_r)$. Since the procedures are similar, we only discuss how $UH(P)$ is constructed. The procedure *HALFHULL* eliminates all vertices of P_U that are not on $CH(P)$. Let P_U be represented by a queue Q in which the vertices are in clockwise order as they appear in P_U. The procedures $FRONT(Q)$ and $DEQUEUE(Q)$ both find the head (current front element) of the queue Q. The difference is that *DEQUEUE* deletes the returned element from Q, but *FRONT* does not. A stack S is needed in *HALFHULL* to record all vertices of P_U that have been recognized as vertices of the U-hull of P_U. We use variable T as the stack pointer that points to the top of stack S. Standard stack operations *PUSH* and *POP* are assumed. We use u to represent the vertex of P_U preceding vertex $S(T)$ on P_U. The procedure iterates until all vertices in P_U are examined. The upper hull of P is available in stack S once all vertices in queue P_U have been exhausted.



(a) (b)

Fig. 2.2 Regions where vertex q following $S(T)$ maybe.

Vertices $S(T-1)$, $S(T)$, u, and p_r are used to determine whether $FRONT(Q)$ should be included in S and whether some vertices already in S should be kept for constructing $UH(P)$. Three or four regions of the vertical strip area determined by p_l and p_r can be identified, depending on the relative locations of vertices $S(T-1)$, $S(T)$, u, and p_r, as shown in Figure 2.2. If u is on or to the right of the directed line from p_r to $S(T)$, then *three* regions, R_1, R_2 and R_3, are identified (Figure 2.2(a)); otherwise, *four* regions, R_1, R_2, R_3 and R_4, are identified (Figure 2.2(b)). Three lines, the first passing through $S(T)$ and p_r, the second passing through $S(T)$ and u, and the third passing through $S(T-1)$ and $S(T)$, are used to bound the three regions in the situation shown in Figure 2.2(b). For the situation depicted in Figure 2.2(a), two of these three lines are used to bound the regions. We call the ordered triple $(p_i p_j p_k)$ a right turn (left turn) if and only if p_k is to the right (left) of the straight line passing by p_i and p_j and directed from p_i to p_j. Given three points $p_i = (x_i, y_j)$, $p_j(x_j, y_j)$ and $p_k(x_k, y_k)$, the position of point p_k relative to the line through points p_i and p_j can be conveniently established using homogeneous coordinates [M57, PS88] as follows: let

$$D(i,j,k) = \begin{bmatrix} x_i & y_i & 1 \\ x_j & y_j & 1 \\ x_k & y_k & 1 \end{bmatrix}$$

If $D(i, j, k)$ is 0, then p_j is on the line passing through points p_i and p_j. If $D(i, j, k)$ is greater than 0, p_k is above the line (left turn). Otherwise, p_k is below the line (right turn). Assuming that a multiplication operation can be done in $O(1)$ time, then testing whether $(p_i p_j p_k)$ is a right turn or left turn takes $O(1)$ time. The procedure *HALFHULL* described given below is taken from [PS88], an original variant of Lee's algorithm [L83], with a slight modification as in Graham and Yao [GY83].

The procedure *HALFHULL* finds the upper-hull $UH(P)$ and stores it in stack S in clockwise order. The time complexity analysis of *OSS_HULL* is straightforward. Its correctness is shown in [GY83] and [L83].

2.3 A Simplified Algorithm

We propose a modified algorithm which we refer to as the *refined single-stack* algorithm (RSS_HULL). Let p_0 and p_2 be the points of P that have the minimum and maximum x-coordinate values, respectively, and let p_1 and p_3 be the points of P that have the minimum and maximum y-coordinate values, respectively. P can be partitioned into four chains $C_{i,(i+1)mod4}(P)$, $0 \le i \le 3$, as shown in Figure 2.3(a). $C_{i,(i+1)mod4}(P)$ contains all vertices of P on the path from p_i to $p_{(i+1)mod4}$ in clockwise order, including vertices p_i and $p_{(i+1)mod4}$. Clearly, p_0, p_1, p_2 and p_3 are on $CH(P)$. Then, $CH(P)$ can also be partitioned into four chains: $CH_{i,(i+1)mod4}(P)$, $0 \le i \le 3$, as shown in Figure 2.3(b). We refer to each portion $CH_{i,(i+1)mod4}(P)$, $0 \le i \le 3$, of the convex hull $CH(P)$ as a *quarter hull* of P. By the definition of simple polygons and the Jordan curve theorem , we know that all vertices of $CH_{i,(i+1)mod4}(P)$ are vertices of $C_{i,(i+1)mod4}(P)$. The algorithm RSS_HULL(P) follows:

Algorithm *RSS_HULL(P)*

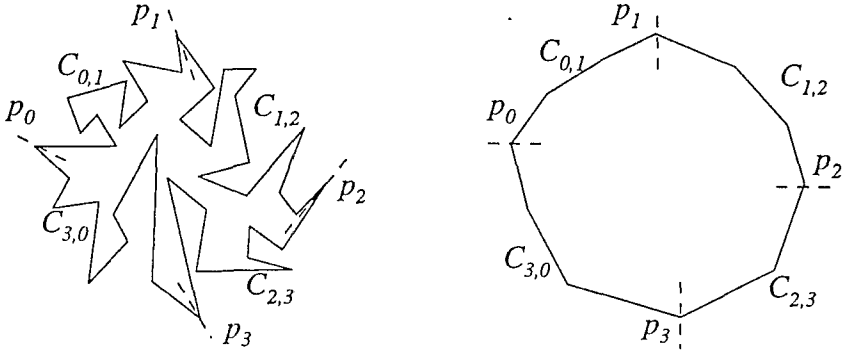
Find the extreme points p_i, $0 \le i \le 3$;
Find $C_{i,(i+1)mod4}(P)$, $0 \le i \le 3$;
for $i = 0$ **to** 3 **do**

$CH_{i,(i+1)mod4}(P)$:= $QUADHULL(C_{i,(i+1)mod4}(P), p_{(i+1)mod4})$;

endfor

Concatenate $CH_{0,1}$(P), $CH_{1,2}$(P), $CH_{2,3}$ (P) and $CH_{3,0}$(P).

end *RSS_HULL*



(a) Four chains (b) Four quarter hulls

Fig. 2.3 Quarter partition of a simple polygon

The location of the extreme points, p_is, and chains, $C_{i,(i+1)mod4}(P)$s, is easily found in a single pass by scanning the vertices of P. Then, four quarter hulls of $CH(P)$ are constructed from the four chains of P using procedure *QUARTERHULL*. Hence, the correctness and the efficiency of the algorithm *FIND_HULL2* depend on procedure *QUARTERHULL*. Without loss of generality, we base the discussion on the procedure *QUARTERHULL* on the construction of $CH_{0,1}(P)$, the upper-left quadrant of $CH(P)$.

Similar to the procedure *HALFHULL* described in the previous section, the procedure *QUARTERHULL* employs a queue Q that contains a list of vertices of $C_{0,1}(P)$ in clockwise order, and a stack S. The variable T is used as the stack pointer. When *QUARTERHULL* is applied to find $CH_{0,1}(P)$, only *three* vertices, $S(T-1)$, $S(T)$ and p_1, are used for decision making (note: *four* reference points are used in procedure *HALFHULL*). Three regions of the vertical strip area determined by p_0 and p_1 can be identified using two lines, the first passing through $S(T-1)$ and $S(T)$, and the second

passing through $S(T)$ and p_1. These regions are shown in Figure 2.4. If we compare Figure 2.4 with Figure 2.2, we can see that vertex u is not used in region partitions of Figure 2.4. Furthermore, Figure 2.4 is similar to Figure 2.2(b). In Figure 2.2(b), the area below the two lines, the first passing through vertices $S(T-1)$ and $S(T)$, and the second passing through vertices $S(T)$ and p_1, is divided into *two regions*, region 1 and region 4. In Figure 2.4, the area below the two lines: the first passing through vertices $S(T-1)$ and $S(T)$ and the second passing through vertices $S(T)$ and p_1 is identified as *one* region: region 1. Procedure *QUARTERHULL* is given below:

```
Procedure QUARTERHULL( Q , p )
    PUSH(S, p);
    q := DEQUEUE(Q);
    PUSH(S, q);
    while Q is not empty do
      begin
        q := DEQUEUE(Q);
        if (S(T − 1)S(T)q) is right turn then /* q is in region 1 or 3 */
          if (pS(T)q) is right turn then /* q is in region 3 */
            PUSH(S, q); /* otherwise, q is in region 1, do nothing */
        else /* q is in region 2 */
          begin
            while (qS(T)S(T − 1)) is right turn do
              POP(S);
            endwhile
            PUSH(S, q);
          end
      end
    endwhile
    return(S);
end QUARTERHULL
```

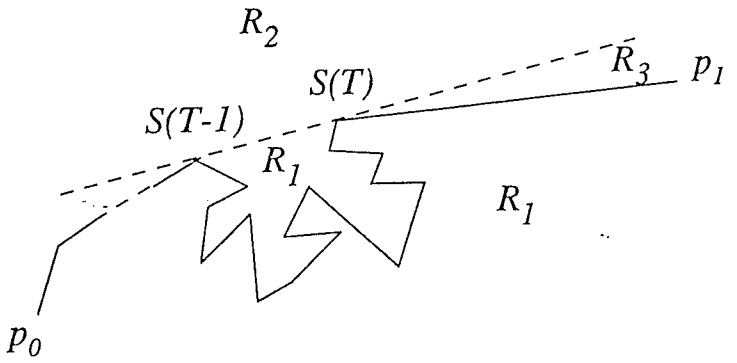


Fig. 2.4 Three regions where vertex q following $S(T)$ maybe

Instead of repeating the arguments given in [GY83] and [L83] to show that procedure *QUARTERHULL* is correct, we informally derive the correctness of *QUARTERHULL* from the correctness of procedure *HALFHULL*. Consider the construction of $CH_{0,1}$. In Figure 2.4, the area below the two lines, one passing through vertices $S(T-1)$ and $S(T)$ and another passing through vertices $S(T)$ and p_1, is identified as a single region called region 1. During the construction of $UH(P)$ by *HALFHULL*, if q (which is the vertex currently being considered) is in region 1 or region 4, then it is rejected for further consideration since it cannot be in $UH(P)$. During the construction of $CH_{0,1}(P)$ by *QUARTERHULL*, if q is in region 1, then it is rejected for further consideration since it cannot be in $UH(P)$. We compare the remaining cases of *HALFHULL* and *QUARTERHULL* in parallel. If q is in region 3, then q is pushed on the stack S in both *HALFHULL* and *QUARTERHULL*. This is because q will be a vertex on the hull of all previously considered vertices (including q). Similarly, if q is in region 2, then q should be a vertex on the hull of all previously considered vertices (including q) in both *HALFHULL* and *QUARTERHULL*. The vertices in the stack S are checked in reverse order to see whether or not they should be on the hull with respect to the inclusion of q. This procedure is exactly what is done in both *HALFHULL* and *QUARTERHULL*. The correctness of *HALFHULL* implies the correctness of

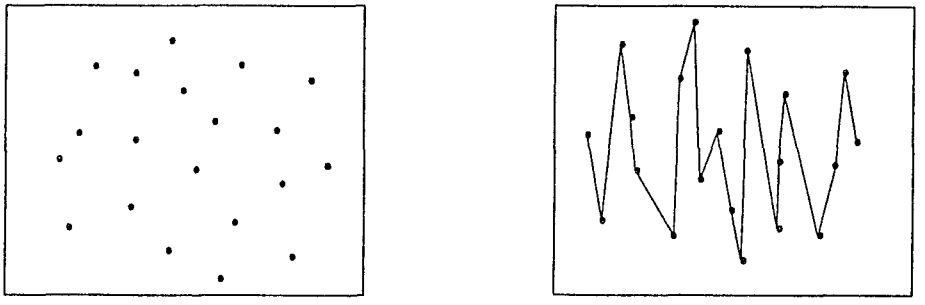
QUARTERHULL. The time complexity of our refined single-stack algorithm *RSS_HULL* is $O(N)$, simply because it performs a linear scan on the given ordered list of vertices of P, and each vertex is pushed onto the stack at most once.

2.4 Analysis and Comparison

The refined single-stack algorithm presented in this chapter is conceptually simpler than the original single-stack algorithm. The algorithm constructs four quarter hulls separately instead of constructing two half-hulls. Since the edges connecting two adjacent vertices in a quarter hull must be monotonically increasing or decreasing, the decision of whether or not a vertex should be included into the partial quarter hull currently under construction is easier than that for the partial half-hull, as can be observed in procedures *QUARTERHULL* and *HALFHULL*. *HALFHULL* needs four reference vertices and three reference lines, whereas *QUARTERHULL* uses only three reference vertices and two reference lines. Consequently, our algorithm *RSS_HULL* is more efficient than the original single-stack algorithm. It is obvious that if a vertex q is pushed onto the stack S during the execution of one of the four procedure calls to *QUARTERHULL*, it must also be pushed onto the stack S during the execution one of the two procedure calls to *HALFHULL*, but the reverse is not true. Thus for any input simple polygon P, the total number of $(p_i p_j p_k)$-type tests performed by our refined single-stack algorithm is less than or equal to the total number of such tests performed by the original single-stack algorithm given in [GY83] and [L83]. In fact, in the worst case, the number of $(p_i p_j p_k)$-type tests performed by the original single-stack algorithm can be at least two times the number of such tests by our algorithm. Considering that a $(p_i p_j p_k)$-type test involves several multiplication (or division) operations, the refined algorithm proposed here is faster than any previously known algorithms for the construction of the convex hull of a simple polygon, when the constant factor is taken into account.

2.5 Summary

Using the relationship among the slopes of points, the construction of the convex hull for a simple polygon can be divided into four sub-problems rather than two sub-problems used in the previous algorithms. Since the four partition strategy requires less constraint for removing non-extreme points than the two partition does, the computation time is reduced. Therefore the algorithm for constructing the convex hull of a simple polygon is simplified. In addition, given a set of planar points, if we sort the set according to the x-coordinates of the points, the set of the sorted points can be defined as a upper (lower) chain of a simple polygon as shown in Figure 2.5.



(a) A set of planar points (b) Points sorted by x-coordinates

Fig. 2.5 Transfer a general case to a simple polygon case

Thus the algorithm proposed in this chapter can be applied to this set of sorted points to construct the convex hull of the set of planar points. Since the lower bound of the convex hull algorithm for a general case is $O(nlogn)$ and sorting against the x-coordinates of points costs $O(nlogn)$, the algorithm extended from the algorithm for a simple polygon is optimal. In the next chapter, we show how this design idea can be used to explore parallel convex hull algorithms.

Chapter 3

An Odd-Even Parallel Convex Hull Algorithm

In this chapter, we first discuss existing linear array convex hull algorithms with their architectures and design approaches. In section 3.2, a new approach for designing parallel convex hull algorithms is presented. An Odd-Even parallel convex hull algorithm is proposed in section 3.3. In section 3.4, an example is provided to demonstrate the correctness of the algorithm. Finally, a summary is given in section 3.5.

3.1 The Previous Parallel Convex Hull Algorithms on Linear Array

According to the survey in [AL93], three linear array convex hull algorithms have been proposed before. Chazelle's algorithm [C84] is considered as the first linear array convex hull algorithm for a dynamic situation in which all the points in a set are not available when the algorithm starts. Two systolic arrays are required by Chazelle's algorithm, one for reporting convex hull vertices and the other for forming the ordered list of convex hull vertices. To determine whether a points v is a convex vertex, we need to check if there exists a minimum convex wedge centered at the point v that contains all the other points. The algorithm is executed as follows: A point v is pumped from the leftmost processor of the linear array. It travels from left to right and determines whether the smallest convex wedge, centered at v, contains all the points encountered so far. If it does, v stops at the first vacant processor on the right-hand side. Otherwise, it is deleted. After all the convex hull vertices are identified, a second array is used to sort the convex hull vertices in either clockwise or counterclockwise. The algorithm is running in a one point per processor fashion and its complexity is $O(n)$ in time with $O(n)$ processors, which is optimal.

Chen *et al* [CCL87] proposed a linear array convex hull algorithm using a single systolic array for a static situation in which all the input points are available before the

algorithm starts. The single systolic array composed of m processors and a special processor called *selector* which are connected in a ring. Interaction with the outside world takes place solely at the selector. Two registers R_u and R_d are required in each processor, and enough memory must be provided in the selector for storing n objects $a_1, a_2, \ldots, a_n$. The basic operation is a cyclic right shift of the objects held in all the R_u-registers or all the R_d-registers. The selector takes part in the shifting operation. When the selector receives an object a_x from processor m, it sends some other object a_y to processor 1 where $y = (x - m - 1) mod \ \ n + 1$. The kernel of the algorithm is to examine all pairs out of n objects. It uses the basic shift operation in such a way that every object pair $a_i a_j$ sooner or later meet the R_u, R_d-registers of some processor.

This special systolic array is used twice for the convex hull problem. First, it is used to identify the convex hull vertices using the same approach introduced in Chazelle's algorithm [C84]. Secondly, it is used to order all the convex hull vertices determined using the approach proposed by Javis [J73]. In other words, by designing a different systolic array, Chen *et al* converted Chazelle's two-systolic-array algorithm to a one-systolic-array algorithm for a static situation. The algorithm runs in (n^2/m) with $O(m)$ processors. The product of the time complexity and the processor complexity is $O(n^2)$ which is the same as the one in Chazelle's algorithm.

Not like the linear algorithms designed by Chazelle and Chen *et al* which are based on the design idea of Javis March, Holey and Ibarra [HI92] proposed their linear convex hull algorithm (referred as Holey's algorithm hereafter) based on Graham Scan.

Holey's algorithm uses $n/3$ processors. Each processor stores three points at a time, sorted lexicographically. The points are sorted across the array by swapping elements. An optimization function is used to determine whether a points is a non-extreme point by checking the relationship among the three points. If the middle point

of the three is below the line connected by other two points, it is a non-extreme point and eliminated. As points are eliminated, they are replaced by the value **nil** which is pushed to the end of the array holding points of greatest x-coordinate. The algorithm is worked as follows: Each processor has five memory cells for storing five variables, *left, small, middle, large* and *right*, respectively. Initially, three points are stored in the memory cells of *small, middle,* and *large* on each of processors and immediatedly sorted lexicographically. The middle points is tested by the optimization function against the other two, and if eliminated, it is replaced by **nil** which is considered to have an x-coordinate of the positive infinite. At each cycle, the processor sends its greatest point (or **nil** if fewer than three real points remain) to its greater numbered neighbor and its least point to its lesser numbered, receiving and storing the greatest point from its lesser numbered processor to its *left*, the least point from its greater numbered processor to its *right*. If the points(*small*) it sent to and received from its lesser neighbor(*left*) were not in sorted order, it keeps the new point(*left*) and discards the one it sent(*small*); if they were in sorted order, it uses the point it received(*left*) and its middle point to test if the point it sent(*small*) can be eliminated. The points sent to(*large* and received from the greater neighbor(*right*) are treated analogously. Then the remaining points are resorted and if there are three non-**nil** points, the middle is again checked for elimination. This loop is repeated until all points are sorted and all non-hull points are eliminated. Holey's algorithm is written as follows, where *elim_p* is the optimization function and n is the number of the points:

```
HI_CONVEX_HULL(small, middle, large, n)
begin
 for i := 1 to (4*n + 2)/3 do
  sort(small, middle, large)
  if elim_p(small, middle, large) then
```

```
      middle := large
      large := nil
    end if
    send large to right_neighbor
    send small to left_neighbor
    receive right from right_neighbor
    receive left from left_neighbor
    if not eod (left_neighbor) then
     if left = nil  or left > small then
       small := left
     else if elim_p (left, small, middle) then
       small := nil
     end if
    end if
    if not eod (right_neighbor) and right ≠ nil then
     if large = nil  or right < large  then
       large := right
     else if elim_p (middle, large, right) then
       large := nil
     end if
    end if
  end for
  sort(small, middle, large)
end
```

The upper bound on the number of the iteration in Holey's algorithm is $4n/3$ because if the points are entered in unsorted order, the points become sorted in at most

$n/3 + 2$ iterations of the loop. While it requires $n-2$ cycles if exactly one point is eliminated at each repetition and there are $n-2$ non-extreme points for a sorted list of points. Therefore, $4n/3$ iterations are required. As a theorem, Holey's algorithm runs in $O(n)$ in time and uses $O(n)$ processors where n is the number of points in a set.

3.2 A New Approach for Designing Linear Array Convex Hull Algorithm

As mentioned above, Chazelle and Chen *et al* adopted the approach proposed in the Jarvis March, and Holey's algorithm absorbed the method introduced in the Graham Scan. In this section, we present a different design approach derived from the sequential algorithm given in Chapter 2.

Basically, Chazelle and Chen *et al*'s algorithms use the order of polar angles of edges implied in Jarvis March. Holey's algorithm applies the order of x-coordinates of points provided in Graham Scan. The approach we propose employs the ordering relationship of the slopes of consecutive points introduced in Chapter 2. In order to present the approach, an observation is given as follow:

Observation

If v_i , v_{i+1} and v_{i+2} are three consecutive points with increasing x-coordinates on the upper hull, where $i = 0..m-3$ and m is the number of the sorted points in $CH(S)$, the following condition holds:

$$slope(v_i, v_{i+1}) \quad \geq \quad slope(v_{i+1}, v_{i+2}).$$

This observation indicates that the slopes of a convex halfhull edges are in order, that is, the slopes of the edges on the upperhull are in descending order and the slopes of the edges on the lowerhull are in ascending order as the x-coordinates of the points are increasing. Obviously, for n sorted points $v_1, v_2, \ldots, v_n$, there are $n-1$ slopes between the consecutive points, namely, v_1v_2, $v_2v_3, \ldots, v_{n-1}v_n$. Therefore, sorting against slopes takes at most $O(nlog(n))$ time sequentially.

Based on the observation above, we present the following algorithm for constructing a convex hull of a set of planar points.

Given a set of planar points, $S = \{t_1, t_2, ..., t_n\}$.

Begin

1. Sort points by x-coordinates;

   $\{t_1, t_2, ..., t_n\} \rightarrow \{v_1, v_2, ..., v_n\}$ where

   $v_1, v_2, ..., v_n$ are ordered according to x-coordinates;

2. Let $S_1 \leftarrow \{v_1, v_2, ..., v_n\}$; $S_2 \leftarrow \{v_n, v_{n-1}, ..., v_1\}$;

   call subroutine HALFHULL,

   $Q_1 \leftarrow$ HALFHULL(S_1); $Q_2 \leftarrow$ HALFHULL(S_2);

3. Merge the two subhulls:

   $Q \leftarrow Q_1 \cup Q_2$;

End

The first step takes $O(nlog(n))$ time because it requires sorting. For the subroutine HALFHULL, a number of existing sorting algorithms can be used to sort all slopes of the points in descending order. This second step takes $O(nlog(n))$time. The third step, which is trivial, takes $O(n)$ time at most. So the total time complexity of the algorithm is $O(nlog(n))$ time, where n is the number of points. Although the Graham scan[G72], which requires $O(n)$ time, can be used in the second step, the ordering relationship of slopes can be easily implemented in a parallel fashion as discussed later in this chapter.

Although the convex hull problem and sorting problem have very close relationship, they differ from the two aspects. First, for a sorting problem, the number of inputs data is equal to the number of output data; however, for a convex hull problem, the number of inputs is usually more than the number of the outputs when some non-extreme points are eliminated from the input point set. For instance, if the input is a set

of 10 whole numbers, a sorting algorithm will produce a sorted list of these 10 whole numbers. On the other hand, 10 planar points might not generate a convex hull with all of 10 points being its extreme vertices. Secondly, in a sorting problem, the comparison is made between two numbers. But in a convex hull problem, at least three points are needed to compare two slopes. These problems prevent us from directly applying an existing parallel sorting algorithm to the convex hull problem.

In previous parallel convex hull algorithms, the non-extreme points are removed by assigning the corresponding variable a null value[C84][HI92]. This arrangement cannot be used in a sorting situation because it violates the ordering relationship among the points. In order to keep the ordering relationship among the points, a duplication strategy is chosen. If $Slope(v_{i-1}, v_i)$ is less than $Slope(v_i, v_{i+1})$, v_i is eliminated by duplicating v_{i-1} on the corresponding processor for v_i. Thus, for n input points the convex hull algorithm generates n output points with m distinct points which represents the convex hull, where $m \leq n$ with $n - m$ left most point.

The second problem is solved based on the data communication strategy used in the parallel odd-even sort algorithm on a linear array. In the odd-even transposition sort algorithm, an odd(even) processor obtains a point from an even(odd) processor adjacent to it. In our algorithm, an odd(even) processor is arranged to access its two neighboring processors to get two points. Since in a linear array an odd(even) processor has two neighboring processors which are even(odd) processors, any operation on the odd(even) processors does not affect the data in the even(odd) processors. This strategy keeps data consistency.

3.3 An Odd-Even Parallel Convex Hull Algorithm

The sorting algorithm adopted in the algorithm is the odd-even transposition sort introduced in [Q87], with some necessary modification for accommodating the two problems mentioned in the previous section. Suppose that n points are distributed

among n processors in an one-point-per-processor pattern. The points are sorted according to their x-coordinates. The sorted points are then sorted according to their slopes. Since sorting against x-coordinates of points is a regular sorting problem, the main focus of this research is the parallelization of sorting against slopes of points.

Similar to the approach for the odd-even transposition sort in[Q87], the algorithm proposed here is designed for the machine with a SIMD-MC1(Single Instruction and Multiple Data stream - Mesh Connected with one dimension) architecture. The processing elements are organized into a one-dimensional array. The data structures are arranged as follows:

(1) $A(a_1, a_2, \ldots, a_n)$: array used to store the points. For each point a_i, $1 \le i \le n$: a_i(x) represents the x-coordinate and a_i(y) represents the y-coordinate of the point sorted in a_i.

(2) $B(b_1, b_2, \ldots, b_n)$ and $T(t_1, t_2, \ldots, t_n)$: arrays used to represent temporary memory for the values in the array A.

(3)P_i, $1 \le i \le n$, represents processor i with array elements a_i, b_i and t_i as local memory cells.

The parallel algorithm is as follows:

P-HALFHULL(S):

begin

for i $\leftarrow$ 1 to n **do**

for all P_j , $1 \le$ j $\le$ n **do**

/* ODD-EVEN EXCHANGE */

if j < n **and** ODD(j) **then**

$b_j \Leftarrow a_{j-1}$ /* obtain a point from the preceded processor */

$t_j \Leftarrow a_{j+1}$ /* obtain a point from the succeeded processor */

if $NOT(SLOPE(b_j,a_j) > SLOPE(a_j, t_j)$ **and** $(a_j \neq b_j))$ **then**

$a_j \leftarrow b_j$ /* replace the value of a_j with the value of a_{j-1} */

endif

endif

/* EVEN-ODD EXCHANGE */

if j > 1 **and** EVEN(j) **then**

$b_j \Leftarrow a_{j-1}$ /* obtain a point from the preceded processor */

$t_j \Leftarrow a_{j+1}$ /* obtain a point from the succeeded processor */

if $NOT(SLOPE(b_j,a_j) > SLOPE(a_j, t_j)$ **and** $(a_j \neq b_j))$ **then**

$a_j \leftarrow b_j$ /* replace the value of a_j with the value of a_{j-1} */

endif

endif

endfor

endfor

end

To establish the correctness of the algorithm, we assume, as the previous linear array array algorithms did, no three points in the given set of planar points are co-linear.

Lemma 1

When a point is eliminated, the duplication operation for the point moves to the left-hand side one processor per odd or even comparison step until the duplication of the leftmost point is made.

Proof: Actually, a point is eliminated by duplicating the point in its preceding processor. In the following step, this preceding processor conducts the comparison and a duplication of the point from its preceding processor is obtained. This type of the

duplication does not stop until a duplication of the leftmost point occurs. For instance, suppose that the current step is an odd step, i is an odd number, and processors P_{i-1}, P_i and P_{i+1} hold points v_{i-1}, v_i and v_{i+1} respectively, if slope(v_{i-1}, v_i) is smaller than slope(v_i, v_{i+1}), point v_{i-1} is duplicated on P_i. In the next step, an even step, P_{i-1} is the processor which makes the comparison. Since processors P_{i-1} and P_i have the same point, a duplication of point P_{i-2} is made on P_{i-1}. This type of duplication will move to the left-hand side until a duplication of point v_1 is made.

Lemma 2

If the point v is eliminated in the jth step, $i - e - 1$ odd-even steps are required to generate a duplication of the leftmost point for v, where i is the position of processor containing point v and e is the number of points eliminated before v.

Proof: If v on processor P_i is the first point eliminated, by lemma 1 it takes $i - 1$ odd-even comparisons to generate a duplication of v_1 on the left end of the array. If e points are eliminated and if these e points reach v_1 before v does, these points generate e duplications of v_1 from processor P_2 to P_e. Then the point v will generate a duplication of v_1 on processor P_{e+1}. That is, after the elimination of v on processor P_i, $i - e - 1$ odd-even comparisons are needed to produce a duplication of v_1 on P_{e+1}, by lemma 1.

Lemma 3

If v is the only non-extreme point in S, a maximum of $n - 1$ odd-even comparison steps is needed to generate a duplication of v_1 on the left side, where n is the number of total points in S.

Proof: If v_2 is the only non-extreme point in S, it is eliminated in the second step of the first iteration and a duplicated v_1 is generated to replace v_2 on the processor p_2.

If v_3 is the only non-extreme point in S, it is eliminated in the first step of the first iteration and a duplicated v_1 is generated on p_2 in the second step of the first iteration. Thus, no matter v_2 or v_3 is the only non-extreme point, two odd-even comparison steps are sufficient. Assume that v_m or v_{m+1} needs m odd-even comparison steps to generate a duplication of v_1 on the left, where m is even and less than $n-3$. If v_{m+2} is the only non-extreme point in S, it is eliminated in the second step of the first iteration and a duplicated v_{m+1} is generated on p_{m+2}. If v_{m+3} is the only non-extreme point in S, it is eliminated in the first step of the first iteration and a duplicated v_{m+1} is generated on p_{m+2} in the second step of the first iteration. In other words, after two odd-even comparison steps, the only non-extreme point (no matter it is v_{m+2} or v_{m+3} initially) is shifted to the processor p_{m+1}. By induction, it takes $m+2$ odd-even comparison steps to generate a duplication of v_1 when v_{m+2} or v_{m+3} is the only non-extreme point in S. Therefore, if v is the only non-extreme point in S, a maximum of $n-1$ ($n-2$ if n is even) odd-even comparison steps is sufficient to generate a duplication of v_1 on the left side. Lemma 3 can also be proven by using lemma 2. Since v_n is an extreme point, v_{n-1} is a non-extreme point in S which takes the most steps to generate the left-most point v_1 at the right side. When n is an even number, v_{n-1} is eliminated in the first odd-even comparison step. When n is an odd number, v_{n-1} is eliminated in the second odd-even comparison step. By lemma 2, since $i = n-1$ and $e = 0$, $n-2$ odd-even comparison steps are required if n is an even number, or $n-1$ steps if n is an odd number, to generate a duplication of v_1 for the non-extreme point v_{n-1}.

Based on the lemmas given above, the complexity of the algorithm can be given as follows.

Theorem

A sorted set of n planar points can be constructed into a upperhull or a lowerhull within n pairs of odd-even comparisons on a linear array.

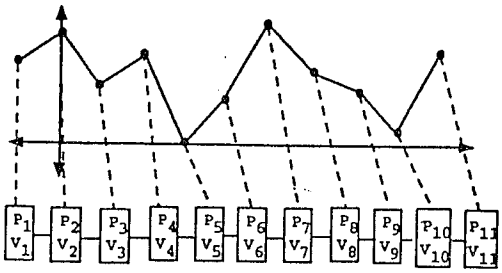
Proof : For a given set of n planar points, at least two of them, the leftmost and the rightmost points, are on the upper hull or the lower hull of the set. In other words, there are at most $n-2$ non-extreme points in a given set of planar points. In a linear array, non-extreme points could be eliminated anywhere between the leftmost processor and the rightmost processor which hold the leftmost and the rightmost points of a set of points, respectively. According to lemma 3, the worst case is that all of the $n-2$ non-extreme points are eliminated on the right side. Suppose that the first point eliminated is v_{n-1}. Then the next non-extreme point v_{n-2} is eliminated two comparison steps later on the right side. The remaining $n-4$ non-extreme points are eliminated in the same pattern. Next, $n-1$ comparison steps are required to make a duplication of v_1 for v_{n-1} by lemma 3. The non-extreme point right after v_{n-1} needs only one more comparison step to duplicate a v_1 for itself because the distance between the two points is narrowed by the duplication of v_1 made by the former point. In other words, $n-3$ more odd or even comparison steps are needed to eliminate the remaining $n-3$ non-extreme points after v_{n-1}. Therefore, the total number of odd or even comparison steps is $n-1+n-3$, which is $2(n-2)$. Thus, a total of $(n-2)$ pairs of odd-even comparison steps is needed for constructing the upper hull or lower hull of a set of n planar points in the worst case. The construction of the the halfhull is completed if no duplication or elimination takes place in a pair of odd-even comparisons.

3.4 An Example of the Odd-Even Convex Hull Algorithm

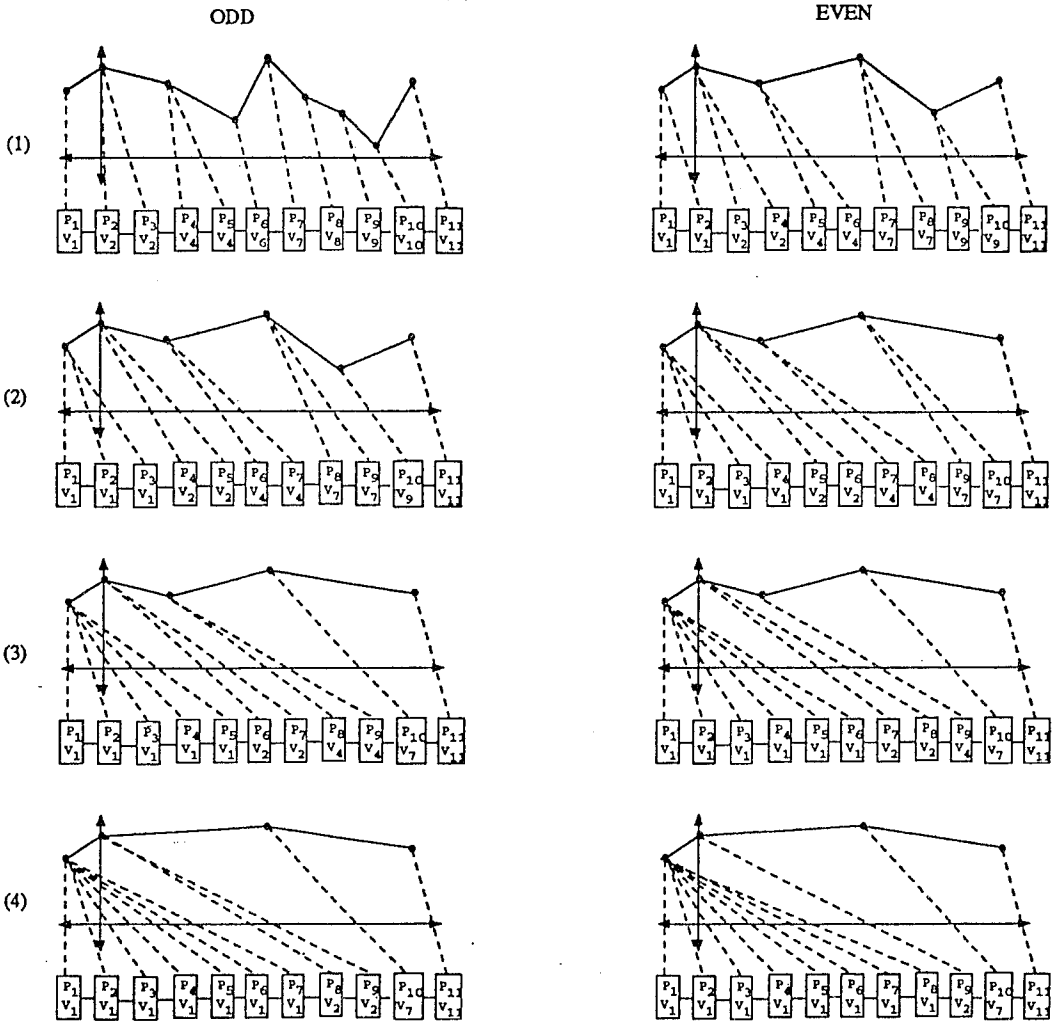
Suppose that the given points are already sorted in the first step of the algorithm and initially located in a linear array shown in Figure 3.1(a). Each small rectangular represents a processing element of a linear array. P_i denotes the *ith* processor and v_i denotes the *ith* point. The diagram in the (i) row and the *ODD* column shows the result after the *ODD* comparison in the ith iteration. Similarly, the diagram in the (i) row and the EVEN column shows the result after the EVEN comparison in the ith

iteration. For instance, during the first ODD comparison, the processor P_3 obtains v_2 from P_2 and v_4 from P_4. Since *slope*(v_2, v_3) is less than the *slope*(v_3, v_4), the point v_3 on P_3 is replaced by the point v_2. This action is shown in (1) row and *ODD* column in Figure 3.1 (b), where the rectangle containing P_3 and v_2 is connected to v_2 by a dotted line. The same action is taken on processor 5 because the *slope*(v_4, v_5) is less than the *slope*(v_5, v_6). For processor 7, since the *slope*(v_6, *vsub*7) is greater than the *slope*(v_7, v_8), no action is taken on processor 7. The point v_9 is replaced by the point v_8 since the *slope*(v_8, v_9) is less than the *slope*(v_9, v_{10}). No action is taken on either processor 1 or processor 11 because they the boundary processors and the points on those processors are extreme points already. On next even step, processor 2 obtains v_1 from processor 1 and v_2 from processor 3. Since the *slope*(v_2, v_2) cannot be computed and is considered as positive infinite, *slope*(v_1, v_2) is less than the "*slope*"(v_2, v_2). Thus, the point v_2 on processor 2 is replaced by the point v_1 from processor 1 as shown in row (1) and column EVEN in Figure 3.1 (b) in which there is a dotted line connecting processor 2 to point v_1. The duplication operation are taken on the rest of even processors because the slope relationship is not satisfied on either processor. The same operations are performed for the rest of iterations. When the algorithm is terminated, the convex hull is shown as in row (4) and column EVEN in Fig. 3.1 (b) with the duplicated v_1 on processors 2 to 8 and processor 9 holds v_2, processor 10 has v_7 and v_{11} remains on processor 11. As we can see, the convex hull computed by the Odd-Even convex hull algorithm is right aligned. If we change the rule for the duplication, such as to duplicate the point from the processor with the higher index, the computed convex hull is left aligned.

In Figure 3.1 (b), only four iterations are shown because after the fourth iteration, the convex hull is computed. But according to the algorithm, seven more iterations are required since there are eleven points involved and eleven iterations are needed based



(a) Initialization



(b) The trace of the algorithm

Fig. 3.1 An example for the odd-even convex hull algorithm

on the algorithm. In other words, if we want to reduce the number of the iterations of the Odd-Even convex hull algorithm, we need to monitor the algorithm. For each of odd (even) step, we check if a duplication operation is taken place. If there is no duplication action occurred for two consecutive Odd-Even steps, the algorithm can be terminated because a pair of Odd-Even comparisons covers all of possibility of having a non-extreme point.

3.5 Summary

The significance of the algorithm presented in this paper is its simplicity and extensibility. The algorithms provided in [C84] and [CCL87] need special chips to implement the algorithms. On the other hand, our algorithm is designed for a general linear array and very easy to be implemented. Although the algorithms presented in [HI92] can be implemented on a general linear array, more comparisons of points are required. In each iteration of Holey's algorithm, five comparisons of points are needed for the computation in a single processor. Since the number of the iterations is $4n/3$, the total number of the comparisons of points is $20n/3$ (about $6n$ to $7n$) for Holey's algorithm. On the other hand, in the Odd-Even parallel convex hull algorithm, $n/2$ comparisons of points are required for the combination on either an odd or even processor. Therefore, a total number of n point comparisons are needed for the Odd-Even parallel convex hull algorithm. In addition, for an SIMD machine, such as the Maspar, a user obtains either all the processors in the machine or none of them. Thus, if the number of a given set of planar points is less than the number of the processors in an SIMD machine, Holey's algorithm has no advantage by using less number of processors. The performance analysis of both the Odd-Even parallel convex hull algorithm and Holey's algorithm is discussed in Chapter 5.

Since the ordering relationship of slopes is used in this paper, the convex hull problem can be solved by adopting a sorting algorithm with minor modifications,

making the control scheme of the algorithm relatively simple and straightforward. While the linear array model is the fundamental structure of parallel computing, the algorithm can be implemented on other structures, such as mesh-connected array. Also, this algorithm can be easily extended to handle the case of $n > p$, where n is the number of points and p is the number of processors, because the corresponding sorting algorithm on a linear array has already been provided[Q87]. In contrast, the algorithms in [HI92] are not easily extended for the case of $n > 3p$ because three points, as opposed to a single point, are allocated on a single processor. The extended and generalized Odd-Even parallel convex hull algorithms are discussed in the next chapter.

Chapter 4

Generalized Odd-Even Convex Hull Algorithms

In this chapter, we generalize the Odd-Even convex hull algorithm presented in the previous chapter to the general cases in which $n \leq p$, $n > p$ and the mesh-array architecture are considered, where n represents the number of points and p indicates the number of processors. After the discussion on the relationship between the linear array architecture and the mesh-array architecture in section 4.1, the algorithm for the case where $n \leq p$ is proposed in section 4.2. The algorithm for the case where $n > p$ is presented in section 4.3. In section 4.4, the algorithm for the d-dimensional mesh-connected arrays is discussed. Finally, in section 4.5, a summary is provided.

4.1 The Mesh-Array Architecture and Convex Hull Algorithms

Linear arrays are a subclass of a class of more general parallel architectures referred as multi-dimensional mesh-connected arrays. A d-dimensional N-ary mesh-connected array (or simply d-dimensional mesh-connected array) has $p = N^d$ processors and $dN^d - dN^{d-1}$ edges. Each processor corresponds to an N-ary d-vector $(i_1, i_2, \ldots, i_d)$, where $0 \leq i_j \leq N-1$ for $1 \leq j \leq d$. Two processors are connected by a communication link if their d-vectors differ in precisely one coordinate and if the absolute value of the difference in that coordinate is 1. It is well known that the d-dimensional N-ary mesh-connected array can also be defined as the cross product of d linear arrays of size N. A direct consequence of this recursive definition is that for those problems that require global communications, the communications between arbitrary pairs of processors, linear array algorithms are usually considered as a base for the design of algorithms for high-dimensional mesh-connected arrays. For example, almost all optimal sorting algorithms on d-dimensional mesh-connected arrays are recursively reduced to sorting on linear arrays.

Several convex hull algorithms designed on mesh-connected arrays have been proposed before, but most of them are based on the divide-and-conquer paradigm [A89, MS89, OSZ93]. The algorithm designed by Holey and Ibarra [HI92] is the only algorithm based on the iterative design strategy.

In [HI92], the mesh-array algorithm is a generalization of Holey's linear array algorithm to higher dimension arrays. Essentially, the mesh-array algorithm cycles through its dimensions performing the linear array algorithm in a different dimension for each repetition of the loop. The processors are numbered in snake-like row-major order, thus embedding a Hamiltonian path into the mesh. A processor sends its greatest point to its greater numbered neighbor in the current dimension and its least point to its lesser numbered neighbor. Within the main loop, a *for-loop* is embedded which cycles the dimension of communication from 0 to $d-1$, so that each repetition of the loop body is in the next dimension of the mesh mod d. The outer loop runs $2d\lceil n^{1/d} \rceil$ times for a total of $2d^2 \lceil n^{1/d} \rceil$ iterations of the inner loop body. Each processor's greater neighbor in a given dimension is that neighbor processor with a higher number in the snake-like row-major numbering system, and its lesser neighbor is the one with a lower number. Therefore, the convex hull of a set of n points can be computed by this algorithm with $O(n)$ processors in $O(n^{1/d})$ time.

4.2 A Parallel Convex Hull Algorithm for The Case Where $n \leq p$

In the previous chapter, we prove that the number of the iterations of the Odd-Even convex hull algorithm is $n-2$. For the generalized cases introduced in this chapter, we use the concept of *turn* to show the relationship between the number of the iterations and the number of the extreme points. Denote the x-coordinate and y-coordinate of a point u by $x(u)$ and $y(u)$, respectively. For any two points u and v, we say that $u < v$ if $x(u) < x(v)$ or $x(u) = x(v)$ and $y(u) < y(v)$. Given any sequence of three distinct points (u_i, u_j, u_k), where $x(u_i) < x(u_j) < x(u_k)$, we say it forms a right

(left) turn if u_k is on the right (left) side of the line passing through u_i and u_j, viewed in the direction from u_i to u_j. The sequence (u_i, u_j, u_k) does not form a turn if the three points are co-linear. As discussed in the previous section, the slopes are decreasing when the upperhull is traced from u_{small} to u_{large}. It can also be said that every three vertices form a right turn when the upperhull is traced from u_{small} to u_{large}. If the direction of the turn is used as the quantity for sorting, the algorithm proposed in the previous section can be immediately modified for using the concept of turns. The algorithm HALFHULL proposed here is used for constructing the upperhull, denoted by $CH_U(S)$.

Assume each processor P_i has a memory cell, A_i, that is used to store the current point associated to P_i. We also use A_i to denote the content of A_i. Initially, A_i contains u_i for $0 \leq i \leq n-1$. For any two points u_i and u_k in $CH_U(S)$ and any point u_j, the sequence (u_i, u_j, u_k) forms a right turn if $i < j < k$. The procedure HALFHULL performs a number of iterations, each consisting of two steps, the *odd step* and the *even step*. During the odd (even) step, each processor P_i, such that i is an odd (even) number greater than 0 and less than $n-1$, compares its A_i with A_{i-1} and A_{i+1}. If A_{i-1}, A_i and A_{i+1} are three distinct points and A_{i-1}, A_i, A_{i+1} forms a left turn, then set $A_i = A_{i-1}$. We call the operation performed by P_i in this case a *point elimination operation.* If A_{i-1} and A_i are different points but A_i and A_{i+1} are the same point, we also set $A_i = A_{i-1}$. We call the operation performed by P_i in this case a *point-move operation.* Note that point-move operations in a step cause point movement to the right in a pipelined fashion. Other than these two cases, processor P_i does not change A_i. We call an iteration an idling iteration if in each of its two steps, none of the A_i's change its content. We say that the $CH_U(S)$ is correctly computed by HALFHULL if when HALFHULL terminates, the following conditions are satisfied: (1) only points in $CH_U(S)$ are in A_i's and these points are in sorted order; and (2) each of the A_j's such that $0 \leq j \leq n-k$,

where $k = |CH_U(S)|$, contains u_0. These two conditions imply that the points of $CH_U(S)$ are right-aligned in A_i's. When an idling iteration occurs, no more points can be eliminated. The details of HALFHULL are as follows.

Procedure HALFHULL
begin
  for k = 1 **to** p **do**
    for all P_i, $0 < i < p-1$, and i is odd **do in parallel**
      if $A_i = A_{i+1}$ **then** $A_i := A_{i-1}$
      else if (A_{i-1}, A_i, A_{i+1}) forms a left turn **then** $A_i := A_{i-1}$
    endfor
    for all P_i, $0 < i < p-1$, and i is even **do in parallel**
      if $A_i = A_{i+1}$ **then** $A_i := A_{i-1}$
      else if (A_{i-1}, A_i, A_{i+1}) forms a left turn **then** $A_i := A_{i-1}$
    endfor
  endfor
end

In order to prove that HALFHULL correctly computes $CH_U(S)$, we derive some facts which are stated in the following four lemmas.

Lemma 1 *After an odd (even) step of HALFHULL, at most two A's contain the same point that is not u_0, and if so, the point is in A_{i-1} and A_i such that i is odd (even).*

Proof: The proof is by a simple induction on the steps. In the first step, a duplicated point can be generated in A_i only when i is odd and P_i performs a point elimination operation. In the later steps, if the step number is even (odd), only processors P_i, where i must be an even (odd) number, either a point elimination operation or point

move operation is performed, can copy the point in A_{i-1} into A_i. This ensures that after an even(odd) step, at most two elements of A contain the same point other than u_0, and if so, the point is in A_{i-1} and A_i such that i is even (odd). □

Lemma 2 *Let $S' = S - \{u_0\}$. If $CH_U(S') = S'$, i.e. all points in S' are on the upperhull of S', then HALFHULL correctly computes $CH_U(S)$ in at most $(n-2)/2$ iterations if all points but the largest one in S' are not on the convex hull of $CH_U(S)$.*

Proof: Obviously, the leftmost point, u_0, of S is in $CH_U(S)$. Every Odd(or Even) step of an iteration of HALFHULL eliminates the currently leftmost point u_i of the remaining points of S' if (u_0, u_i, u_{i+1}) forms a left turn. This process continues until the currently leftmost point in S' that can not be eliminated is found. Clearly, there are at most $n-2$ points in S' that can be eliminated and each iteration of HALFHULL has a pair of Odd-Even steps. Thus, the maximum number of the iterations of HALFHULL for finding the $CH_U(S)$ is $(n-2)/2$. □

Lemma 3 *Let $S' = S - \{u_0, u_1\}$. If $CH_U(S') = S'$, i.e. all points in S' are on the upperhull of S', then HALFHULL correctly computes $CH_U(S)$ in no more than $(n - |CH_U(S)|)/2$ iterations.*

Proof: We prove this lemma by double induction. It is easy to verify the lemma is true for $n \leq 6$. Suppose that for $m - 1 \geq n \geq 6$ the claim is true. We consider this as the first level hypothesis. Consider the case of $n = m$.

If no points are eliminated during the first iteration, then $CH_U(S) = S$. If in the first step of the first iteration, P_1 performs a point elimination operation, then A_1 contains u_0 after the operation. Lemma 2 ensures that $CH_U(S)$ can be computed in $(n-2)/2$ iterations.

The only case left is that point u_2 is eliminated in the second step of the first iteration, but no point is eliminated in the first step. Then, after the first iteration, A_2 contains u_1, and the contents of all other A_i's remain unchanged. We only need to prove the following claim:

Claim: Under the condition that $CH_U(S') = S'$, if A_0 contains u_0, A_1 and A_2 contain u_1, and A_i contains u_i for $2 < i \leq n-1$, then $n - |CH_U(S)| - 1$ iterations are sufficient to compute $CH_U(S)$.

Proof: The proof is by induction on n. It is easy to verify that the claim is true for $n \leq 6$. Suppose that for $n \leq m-1$ the claim is true. We consider this as the second level hypothesis. Consider the case of $n = m$ and a pair of adjacent odd and even steps. There are two possible cases.

Case 1: P_3 performs a point elimination operation in the odd step.

There are two subcases, depending on whether or not P_4 performs a point elimination operation in the even step. If P_4 does not perform a point elimination, then after this step A_0, A_1 and A_2 contain u_0, A_3 contains u_1, and the contents of the other A_i's remain unchanged. Using the first level hypothesis, we know that $(n-3)/2$ additional iterations are sufficient for computing $CH_U(S)$. If P_4 performs a point elimination, then after this step, A_0, A_1 and A_2 contain u_0, A_3 and A_4 contain u_1, and the contents of other A_i's remain unchanged. Using the second level hypothesis, we know that $(n-3)/2$ additional steps are sufficient for computing $CH_U(S)$.

Case 2: P_3 *does not perform a point elimination operation* in the odd step.

In this case, we know that (u_1, u_3, u_4) does not form a left turn. There are two subcases, depending on whether or not P_2 performs a point elimination operation in the even step. If P_2 does not perform a point elimination, then (u_0, u_1, u_3) does

not form a left turn. By the convexity of S', the fact that both (u_1, u_3, u_4) and (u_0, u_1, u_3) do not form left turns (note that u_2 has already been eliminated) implies that the computation of $CH_U(S)$ is complete. If P_2 performs a point elimination, then u_1 is eliminated, and A_i, $0 \le i \le 2$, contains u_0. By Lemma 2, no more than $\lceil (n - |CH_U(S)|)/2 \rceil \le \lceil (n-2)/2 \rceil$, for $n \ge 6$, additional iterations are sufficient for computing $CH_U(S)$.

This completes the induction proof of the claim and the induction proof of the lemma. □

Lemma 4 *Let S be a sorted list of n planar points and $S' = S - \{u_{n-1}\}$. If $CH_U(S') = S'$, i.e. all points in S' are on the upperhull of S', then HALFHULL correctly computes $CH_U(S)$ in no more than $(n-2)$. iterations.*

Proof: As we mention in the proof for lemma 3, if no point is eliminated during the first iteration of HALFHULL, then $CH_U(S) = S$. Otherwise, an elimination operation occurs on P_{n-2}. Assume that $CH_U(S') = (u_{i_1} = u_0, u_{i_2}, ..., u_{i_k} = u_{n-2})$, P_{n-2} and P_{n-3} hold the same point $u_{i_{k-1}}$ after P_{n-2} performs an elimination operation. On the next step of the iteration, P_{n-3} executes a point-move operation and sets itself with the point of $u_{i_{k-2}}$ from P_{n-4} because P_{n-2} and P_3 hold the same point of $u_{i_{k-1}}$. Suppose j points, $u_{i_{k-j+1}}, u_{i_{k-j+2}}, ..., u_{i_k}$, have been eliminated by j iterations of HALFHULL, we need to prove if there are $j+1$ non-extreme points, one more iteration of HALFHULL is required. When the first point u_{i_k} is eliminated, P_{n-2} replaces the u_{i_k} with $u_{i_{k-1}}$ and P_{n-3} substitutes $u_{i_{k-2}}$ for its previous point $u_{i_{k-1}}$ in two consecutive Odd-Even(or Even-Odd) steps. If $u_{i_{k-1}}$ is a non-extreme point, it is eliminated in the next consecutive Odd-Even(or Even-Odd) steps with P_{n-2} holds the point $u_{i_{k-2}}$ and P_{n-3} holds the point $u_{i_{k-3}}$. Thus, if j points are eliminated, P_{n-2} has the point $u_{i_{k-j}}$ and P_{n-3} has the point of $u_{i_{k-(j+1)}}$. Obviously, an Odd(Even) step is required for eliminating $u_{i_{k-j}}$ and the

following Even(Odd) step is needed for moving $u_{i_{k-(j+1)}}$ to P_{n-2}. By induction, $j+1$ iterations are required for eliminating $j+1$ non-extreme points. Since there are $n-2$ non-extreme points in S' at most, the construction of $CH_U(S)$ needs no more than $n-2$ iterations of HALFHULL. □

Theorem 1 *Given a sorted list of S with n planar points, the procedure HALFHULL correctly computes $CH_U(S)$ in no more than $n-2$ iterations.*

Proof: The proof is by induction on n. It can be easily verified that the theorem is true for $n \leq 6$. Suppose that the theorem is true for $n \leq m$, where $m \geq 6$. We want to show that for $n = m+2$ the theorem also holds.

Let $S' = S - \{ u_0, u_1 \}$, Suppose $|CH_U(S')| = k$, and let $CH_U(S') = (u_{i_1} = u_2, u_{i_2}, u_{i_3}, \ldots, \ldots, u_{i_{k-1}}, u_{i_k} = u_{m+1})$. We compare the execution of HALFHULL on P_is for S', where $2 \leq i \leq m+1$, and the execution of HALFHULL on all P_is for S, where $0 \leq i \leq m+1$. We denote these two executions by $E(S')$ and $E(S)$, respectively. We divide the execution $E(S)$ of HALFHULL into two phases. The first phase consists of exactly $2m-k$ iterations, and the second phase consists of additional iterations required to compute $CH_U(S)$.

Let R be the set of all distinct points in A_i's right after the first phase $E(S)$, and let $r = |R|$. An important fact is that $R \subseteq \{u_0, u_1\} \cup CH_U(S')$. By the hypothesis, we know that $E(S')$ takes at most $(m-2$ iterations to compute $CH_U(S')$. Note that in $E(S')$ all points between u_{i_j} and $u_{i_{j+1}}$ are eliminated by comparing the points in the interval defined by the two points. Clearly, all points in R are in sorted order. Suppose that after the first phase, points u_{i_1} through $u_{i_{j-1}}$ are eliminated, and points u_{i_j} through and u_{i_k} are not eliminated. Suppose that $u_{i_1} = u_2$ is eliminated during the l-th iteration because (u_1, u_2, u_q) is detected forming a left turn (Note: if u_1 is eliminated before u_2 is eliminated, u_0 is used instead of u_1; for brevity, we assume that this is not the case),

then u_1 takes the role of u_2. That is, after the j-th iteration of $E(S')$ and $E(S)$, any point a that is eliminated in $E(S')$ by comparing with u_2 and u_b can be eliminated in $E(S)$ by comparing with u_1 and u_b. This is because that if (u_2, u_a, u_b), where $2 < a < b$, forms a left turn, then (u_1, u_a, u_b) also forms a left turn. By a simple inductive argument, the fact that $R \subseteq \{u_0, u_1\} \cup CH_U(S')$ can be derived.

Since S contains two more points, u_0 and u_1, compared with S', the influence of these two points is limited. Let $(u_{t_0}, u_{t_1}, \ldots, u_{t_{r-1}})$ be the sorted sequence of distinct points in all A_i's after the first phase of $E(S)$. Clearly, $u_{t_0} = u_0$, and $u_{t_{r-1}} = u_{i_k} = u_{m+1}$. The two additional points, can only affect the locations of the leftmost three points (i.e. u_{t_0}, u_{t_1}, u_{t_2}) of R. By a simple inductive argument, it is easy to verify that all points of u_{t_3} through $u_{t_{r-1}}$ in R have a single copy in A_i's, and they are right aligned. Since point $u_{t_0} = u_0$ can have more than one copy and all its copies are left aligned, we have to consider four possible cases and apply the induction on each of these cases to show that the theorem is true. For all cases, we assume that the rightmost copy of u_{t_0}, which is u_0, is in A_s. We use I to denote the total number of iterations, which includes the iterations in both of the two phases of $E(S)$, and use A_i to denote the memory cell of P_i for storing its current point and the point itself.

Case 1: Both u_{t_1} and u_{t_2} have a single copy.

In this case, $(A_s, A_{s+1}, A_{s+2}, A_{s+3}) = (u_{t_0}, u_{t_1}, u_{t_2}, u_{t_3})$. If $R = CH_U(S)$, the computation of $CH_U(S)$ is completed. Otherwise, additional iterations are required. There are two subcases:

Subcase 1.1: s is even.

Let $n' = (m+1) - s + 1$. By Lemma 2, no more than $n' - |CH_U(R)|$ additional iterations suffice. Then, the total number of iterations is $I \le 2m - k + n' - |CH_U(R)| \le 2m + 2 - |CH_U(S)| < 2n - |CH_U(S)|$ since $n' \le k + 2$ and $CH_U(R) = CH_U(S)$.

Subcase 1.2: s is odd.

Consider the effect of one more iteration, the $(2m-k+1)$-th iteration. All points to the right of u_{t_1} cannot be eliminated since they are in $CH_U(S')$. If u_{t_2}, in the odd step, is eliminated, then after the even step, $(A_s, A_{s+1}, A_{s+2}, A_{s+3}) = (u_{t_0}, u_{t_0}, u_{t_2}, u_{t_3})$, which is an instance of Subcase 1.1. In this case, let $n' = (m+1)-(s+1)+1$. Otherwise, u_{t_1} must be eliminated in the even step. Then, $(A_s, A_{s+1}, A_{s+2}, A_{s+3}) = (u_{t_0}, u_{t_1}, u_{t_1}, u_{t_3})$, and we encounter an instance of Case 2. In this case, let $n' = (m+1)-s+1$. The proofs of Subcase 1.1 and Case 2 ensure that $I \le 2m-k+1+n'-|CH_U(R)| \le 2m+3-|CH_U(S)| < 2(m+2)-|CH_U(S)| = 2n-|CH_U(S)|$, since $n' \le k+2$.

Case 2: u_{t_1} has two copies and u_{t_2} has a single copy.

In this case, $(A_s, A_{s+1}, A_{s+2}, A_{s+3}) = (u_{t_0}, u_{t_1}, u_{t_1}, u_{t_2})$. Point u_{t_1} can either be u_1 or a point in $CH_U(S')$. Each of A_{s+3} through A_{m+1} contains a unique point of $CH_U(S')$. By Lemma 1, s must be even. Let $n' = (m+1)-s+1$. By the proof of Lemma 3 (more specifically, the Claim in the proof), we know that $n'-|CH_U(R)|-1$ additional iterations are required to complete the computation of $CH_U(S)$. Then, $I \le 2m-k+n'-|CH_U(R)| \le 2m+2-|CH_U(S)| < 2n-|CH_U(S)|$, since $n' \le k+2$.

Case 3: u_{t_1} has a single copy and u_{t_2} has two copies.

In this case, $(A_s, A_{s+1}, A_{s+2}, A_{s+3}, A_{s+4}) = (u_{t_0}, u_{t_1}, u_{t_2}, u_{t_2}, u_{t_3})$. By Lemma 1, s is odd. Clearly, u_{t_2} must be a point in $CH_U(S')$. Consider the effect of one additional iteration, the $(2m-k+1)$-th iteration. After the odd step of an iteration, A_{s+2} replaces its point u_{t_2} by u_{t_1} but no point is eliminated. This is the only change made in this step. In the even step, the only point that may be eliminated is u_{t_2}. We have two subcases.

Subcase 3.1: No point is eliminated in the even step of the $(2m-k+1)$*-th iteration.*

If $r = k+2$, then $CH_U(S) = \{u_0, u_1\} \cup CH_U(S')$, which implies that $u_{t_1} = u_1$ and $u_{t_2} = u_2$. The execution $E(S)$ can be easily derived from the execution $E(S')$ as follows. When u_2 is moved to the right in the j-th step in $E(S')$, u_1 is moved one position to the right in the $(j+1)$-th step, and u_0 is moved one position to the right in the $(j+2)$-th step. Thus, $E(S)$ has at most $2m-k+2$ iterations. Since $2m-k+2 = 2(m+2)-(k+2) = 2n-(k+2)$ and $|CH_U(S)| = k+2$, we know that $I = 2n - |CH_U(S)|$. If $r < k+2$, then after this even step, we have $(A_s, A_{s+1}, A_{s+2}, A_{s+3}, A_{s+4}) = (u_{t_0}, u_{t_0}, u_{t_1}, u_{t_2}, u_{t_3})$, which is an instance of Case 1.1. Using the argument of Case 1.1, we know that $I \le 2(m+1) - |CH_U(S)| = 2n - |CH_U(S)|$.

Subcase 3.2: Point u_{t_2} *is eliminated in the even step of the* $(2m-k+1)$*-th iteration.*

Then, we have $(A_s, A_{s+1}, A_{s+2}, A_{s+3}, A_{s+4}) = (u_{t_0}, u_{t_0}, u_{t_1}, u_{t_1}, u_{t_3})$. This is an instance of Case 2, with one point of $CH_U(S')$ eliminated. Let $n' = (m+1)-(s+1)+1$. By the proof of Case 2, we know that $I \le (2m-k)+1+(n'-|CH_U(R)|-1) \le 2m+3-|CH_U(S)| < 2n-|CH_U(S)|$, since $n' \le k+2$.

Case 4: Both u_{t_1} *and* u_{t_2} *have two copies.*

We have $(A_s, A_{s+1}, A_{s+2}, A_{s+3}, A_{s+4}, A_{s+5}) = (u_{t_0}, u_{t_1}, u_{t_1}, u_{t_2}, u_{t_2}, u_{t_3})$. By Lemma 1, s is even. Clearly, points u_{t_2} and u_{t_3} must be in $CH_U(S')$. Consider the effect of the $(2m-k+1)$-th iteration. After the odd step, A_{s+1} contains $u_{t_0} = u_0$, A_{s+2} and A_{s+3} contain u_{t_1}. These are the only changes made in this step. In the even step, A_{s+2} is changed to contain u_0. There are two possibilities for the content of A_{s+4} after this even step, depending on whether or not u_{t_2} is eliminated in this

step. If the point u_{t_2} in A_{s+4} is not eliminated, then $(A_s, A_{s+1}, A_{s+2}, A_{s+3}, A_{s+4}, A_{s+5}) = (u_{t_0}, u_{t_0}, u_{t_0}, u_{t_1}, u_{t_2}, u_{t_3})$, and we encounter an instance of Subcase 3.1, which leads to $I \leq 2n - |CH_U(S)|$. If the point u_{t_2} in A_{s+4} is eliminated in this even step, then $(A_s, A_{s+1}, A_{s+2}, A_{s+3}, A_{s+4}, A_{s+5}) = (u_{t_0}, u_{t_0}, u_{t_0}, u_{t_1}, u_{t_1}, u_{t_3})$, and we encounter an instance of Case 2, which leads to $I \leq 2n - |CH_U(S)|$.

This completes the induction and the proof of the theorem. □

Corollary 1 *Using odd-even transposition sort and procedure HALFHULL, the convex hull of a set S of n points can be computed on a linear array of n processors, each having $O(1)$ memory, in $O(n)$ time.*

Remarks: It is important to point out that the maximum number of iterations proved in Theorem 1 may not indicate the worst case performance of procedure HALFHULL. In fact, the worst case example we have been able to find requires $n - 1$ iterations, which is exactly one half the maximum number of iterations we have been able to prove. We conjecture that for any set S of n points, $n - 1$ iterations of HALFHULL are sufficient to correctly compute $CH_U(S)$.

4.3 A Parallel Convex Hull Algorithm for The Case Where $n > p$

We generalize the algorithm presented in the previous section to the case that $n > p$, and each processor P_i holds a subsequence of S of $\lceil n/p \rceil$ points (dummy points are added, if necessary, to ensure this assumption). A generalization of the odd-even transposition sort, called the odd-even merge-split sort [BS78b], can be used to sort the points to satisfy the following condition: if point u is in P_i and point v is in P_j, and $i < j$, then $u < v$. This sorting takes $O((n/p)\log(n/p)) + O(n)$ time. In parallel, each processor can apply the Andrew's scan method[A79] to construct the upperhull of the sorted points associated with it. This takes $O(n/p)$ time. Let A_i be the upperhull

for the points in P_i. Then, we can apply the following procedure to merge these sub-hulls into the upperhull of S.

Procedure HALFHULL_G

begin

for k = 1 **to** 2p-2 **do**

for all P_i, $0 < i < p-1$, and i is odd **do in parallel**

if $A_i = A_{i+1}$ **then** $A_i := A_{i-1}$

else

begin

find the portion of A_i that is the subset of the upperhull of $A_{i-1} \cup A_i \cup A_{i+1}$, and let A_i' be this subset of A_i;

if $A_i' = \varnothing$ **then** $A_i := A_{i-1}$ **else** $A_i := A_i'$

end

endfor

for all P_i, $0 < i < p-1$, and i is even **do in parallel**

if $A_i = A_{i+1}$ **then** $A_i := A_{i-1}$

else

begin

find the portion of A_i that is the subset of the upperhull of $A_{i-1} \cup A_i \cup A_{i+1}$, and let A_i' be this subset of A_i;

if $A_i' = \varnothing$ **then** $A_i := A_{i-1}$ **else** $A_i := A_i'$

end

endfor

endfor

end

We say that procedure HALFHULL_G correctly computes the upperhull of S if HALFHULL_G terminates and there exists a value k, $0 \le k \le p-1$, such that (1) $A_i = A_j$ for $i \le k$ and $j \le k$, (2) $A_i \neq A_j$ for $i \ge k$, $j \ge k$ and $i \neq j$, and (3) the concatenation of all the subsequences of A_i's, where $i \ge k$, is the upperhull of S.

Theorem 2 *Procedure HALFHULL_G correctly computes $CH_U(S)$ in no more $2p-k$ iterations, where k is the number of A_i's whose initial points are not all eliminated during execution.*

Proof: Similar to the proof of Theorem 1. Note that if the upper bound of the number of required iterations in Theorem 1 can be proved less than $2n-k$, then the upper bound of this theorem can be reduced accordingly. □

A simple implementation of HALFHULL_G requires the points of current A_{i-1} and A_{i+1} to be transmitted to processor P_i in each step of the two steps of an iteration in order to construct the upperhull of points in $A_{i-1} \cup A_i \cup A_{i+1}$. By Theorem 2, this implementation of HALFHULL_G requires $O((n/p) * (p-k))$ time which is about the same or even less than the $O((n/p)\log(n/p)) + O(n)$ time required by the sort preprocessing time.

We can improve the actual performance of this straightforward implementation by reducing the computation of $CH_U(A_{i-1} \cup A_i \cup A_{i+1})$ to the computation of $CH_U(A_{i-1} \cup A_i)$, and the computation of $CH_U(A_i \cup A_{i+1})$. To compute $CH_U(A_{i-1} \cup A_i)$, we need to find the supporting line of the hulls A_{i-1} and A_i. The supporting line l of hulls is the line that passes through a point u in A_{i-1} and a point v in A_i such that all points in A_{i-1} and A_i are on the same side of it. The line segment with u and v as endpoints is called the *bridge* of hulls A_{i-1} and A_i. Once the bridge of A_{i-1} and A_i is known, then by concatenating the subsequence of points of A_{i-1} that are not to the right of u and the subsequence of points of A_{i-1} that are not to the left of v, we have

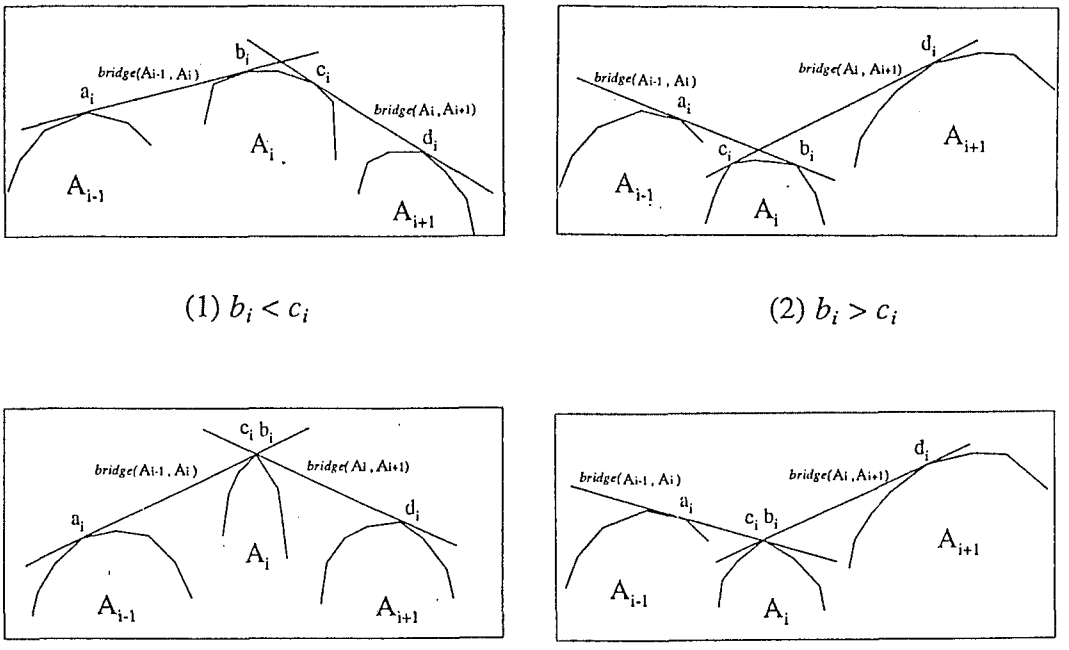
$CH_U(A_{i-1} \cup A_i)$. Sequentially, using the search method described in [OL81], the bridge of A_{i-1} and A_i can be computed in $O(\log (\min \{|A_{i-1}|, |A_i|\}))$ time. In the parallel processing context, the task of finding the bridge of A_{i-1} and A_i can be carried out as follows. P_i selects a point $v_{i,1}$ in A_i and sends it to P_{i-1}, then P_{i-1} finds a point $v_{i-1,1}$ in A_{i-1} such that all points in A_{i-1} are below the line passing through $v_{i,1}$ and $v_{i-1,1}$. P_{i-1} then sends point $v_{i-1,1}$ to P_i, and P_i finds a point $v_{i,2}$ in A_i such that all points in A_i are below the line passing through $v_{i,2}$ and $v_{i-1,1}$. Then, P_i sends point $v_{i,2}$ to P_{i-1}, and P_{i-1} finds a point $v_{i-1,2}$ in A_{i-1} such that all points in A_{i-1} are below the line passing through $v_{i,2}$ and $v_{i-1,2}$. This zigzag process continues until the bridge of A_i and A_{i-1} is found. If A_i and A_{i-1} are represented by a height balanced trees, $O(\log (\min \{|A_{i-1}|, |A_i|\}))$ steps are sufficient for computing $CH_U(A_{i-1} \cup A_i)$. Thus, the number of points transmitted during this process is $CH_U(A_{i-1} \cup A_i)$. By the analysis of [OL81], the total computation time required in this process is also $O(\log \min \{|A_{i-1}|, |A_i|\})$. This process is abstracted as a procedure $bridge(A_{i-1}, A_i)$, which returns two points a_i and b_i, which define the bridge of A_{i-1} and A_i, to both P_{i-1} and P_i. By applying $bridge(A_i, A_{i+1})$, P_i obtains another two points, c_i and d_i that define the bridge of A_i and A_{i+1}. Then, the new A_i can be constructed as follows:

if $b_i > c_i$ or $b_i = c_i$ and (a_i, b_i, d_i) forms a left turn

then $A_i := A_{i-1}$ **else** $A_i := \{u | u \in A_i$ and in not to the left of b_i and not to the right of $c_i\}$

There are four situations with the two bridges, which are shown in Figure 4.1.

In one step, all odd (even) numbered processors perform the computations described above. Clearly, data transmissions among processors are reduced. Also, this technique makes the computing process in-place, i.e. only a small constant number of additional memory cells other than those for A_i are required in each P_i.



(1) $b_i < c_i$ (2) $b_i > c_i$

(3) $b_i = c_i$ and (a_i, b_i, d_i) is a right turn (4) $b_i = c_i$ and (a_i, b_i, d_i) is a left turn

Fig. 4.1 Four situations of the bridges

Since the odd-even merge-split sort can be implemented with $O(n)$ time performance, and $\lceil n/p \rceil + O(1)$ space for each processor, we have the following result.

Corollary 2 *Using odd-even merge-split sort and procedure HALFHULL_G, the convex hull of a set S of n points can be computed on a linear array of p processors, each having* $\lceil n/p \rceil + O(1)$ *memory, in* $O(n)$ *time.*

If the points in S are randomly distributed, procedure HALFHULL_G may only require one iteration. This is because that each A_i may contain at least one point in $CH_U(S)$. According to procedure HALFHULL_G, if A_i becomes empty during a step, the points of A_{i-1} are copied into A_i. In the next step, the points of A_{i-2} are copied into A_{i-1}. This cascade effect is the main factor of the communication overhead. If we can delay, and hopefully prevent such a cascade effect, the performance of

HALFHULL_G can be improved. For this reason, we may incorporate a dynamic balancing feature into HALFHULL_G. We can modify the above statement as follows:

if $b_i > c_i$ or $b_i = c_i$ and (a_i, b_i, d_i) forms a left turn

then A_i := {right half of the points in A_{i-1} and notify P_{i-1} to eliminate this portion from A_{i-1}

else A_i := {$u | u \in A_i$ and in not to the left of b_i and not to the right of c_i}

By this balancing method, the creation of a "hole" can be at least delayed, and the total number of iterations can be reduced. Let each processor P_i have a Boolean variable $flag_i$. Its value is dynamically assigned as follows: before an iteration, set $flag_i$ to true. If values of A_i are different before and after the iteration, then $flag_i$ is set to false at the end of the iteration. Then, after every c iterations, we perform a logic *AND* of all $flag_i$'s. If this logic operation is true, then we know that we have encountered an idling iteration, and no more iterations are needed. Combining this census operation with the balancing method, the actual performance of HALFHULL_G can be expected to be very good. However, in the worst case, all A_i's except A_0 and A_{p-1} do not contain any point in $CH_U(S)$. In this case, data transmission is significant because the hull points of A_0 have to move to A_{p-2} when the procedure terminates in order for $CH_U(S)$ to be correctly computed. For this case, we have no way to avoid cascade data transmission.

4.4 A New Mesh-array Convex Hull Algorithm

In this section, we intend to extend our linear array convex hull algorithm to a d-dimensional mesh-array convex hull algorithm. Since sorting on a d-dimensional mesh-array requires $O(d^2 n^{1/d})$ time [KH83, NS79, TK77], our algorithm is bounded by this time.

Let $n = 3p$. We claim that our HALFHULL_G procedure, combined with sorting, is equivalent to Holey's algorithm. In our algorithm, when a point is eliminated we can

modify its x-coordinate to be $-\infty$. With minor modification to our procedure HALFHULL_G, eliminated points are left-aligned, hull points are right-aligned, and all points are in the increasing order of their x-coordinates when the procedure terminates. In Holey's algorithm, the point u_j is eliminated when (u_i, u_j, u_k) is found forming a left turn. The eliminated points are assigned a new x-coordinate $+\infty$, and shifted step by step to the left. When Holey's algorithm terminates, eliminated points are left-aligned, hull points are right-aligned, and all points are in the decreasing order of their x-coordinates. By slightly modifying our procedure HALFHULL_G and reversing the point order, each step of Holey's algorithm can be simulated by two steps, an odd step followed by an even step, of our algorithm. Using the techniques of [HI92], our modified algorithm can be generalized to compute the convex hull of n points by a d-dimensional mesh-connected array with $O(n)$ processors, each having $O(1)$ words of memory, in $O(d^2 n^{1/d})$ time, which is optimal. Furthermore, our modified procedure HALFHULL_G, combined with odd-even merge-split sort, can be generalized for the d-dimensional mesh-connected array with no restriction of $p = \lceil n/3 \rceil$. For this case, the algorithm obtained by combining our generalized algorithm and the techniques of [HI92] can compute the convex hull of n points by a d-dimensional mesh-connected array with p processors, each having $n/p + O(1)$ words of memory, in $O((n/p) \log (n/p) + np^{1/d-1})$ time, assuming that $n \gg p$, and $p^{1/d} \gg 2$, which is optimal.

4.5 Summary

We compare our algorithm that uses the HALFHULL procedure with the algorithm in [HI92] that uses two-way cellular arrays. For our algorithm, $p = n$, whereas $p = \lceil n/3 \rceil$ for Holey's algorithm. Each processor uses 3 memory cells in our algorithm, whereas 5 memory cells are required by Holey's algorithm. Our algorithm requires an explicit sorting phase, whereas in Holey's algorithm sorting and convex hull construction occur simultaneously. In each step of our algorithm, each processor (but the end

processor) either receives or sends two points from/to its neighbors, whereas in each step of Holey's algorithm algorithm, each processor (but the end processor) receives and sends two points from/to its neighbors. One of the advantages of our algorithm over Holey's algorithm is that our algorithm has a much simpler control scheme.

It is important to note that converting an efficient algorithm that is designed for smaller number of processors to an algorithm using more processors with better or the same efficiency is much more difficult than the reverse - coverting an algorithm designed for larger number of processors to an algorithm for smaller number of processors. It is not clear how to convert Holey's algorithm to the cases that $n = p$ and $n \gg p$. Our algorithms should be considered more general than Holey's algorithm. Indeed, following any $O(p^{1/d})$ time recursive sorting algorithm for the d-dimensional mesh-connected array of p processors, we can generalize procedure HALFHULL to an $O(p^{1/d})$ time procedure that computes the upperhull of a set of n, where $n = p$.

To demonstrate the advantage of the Odd-Even parallel convex hull algorithm, a system was developed to analyze the performance of the algorithms. The system and the performance analysis of the algorithms are discussed in Chapter 5.

Chapter 5

Algorithm Performance Analysis

In order to compare the performance of the various algorithms, we implemented the algorithms on the Maspar. In this chapter, we give a brief introduction to the Maspar, describe the system that implements the algorithms, and discuss the analysis of the speed-up and performance.

5.1. Maspar Machine

The Maspar machine is a general purpose massively parallel computer with SIMD (Single Instruction Multiple Datastream) architecture. By operating in a single instruction multiple data stream fashion, thousands of PEs (Processing Elements) can work on a single problem simultaneously. A stream of instructions is sent out to all PEs from an ACU (Array Control Unit). For each incoming instruction, a PE can either execute the instruction or be passive. This architecture supports the data-parallel style of programming, which is suitable for applications with operations on collections of many similar data elements, such as the convex hull problem. The model of the machine used in this research is the Maspar model 1208B and its architecture is shown in Fig. 5.1.

A Maspar machine system has four components [M91]. First, it has a Data Parallel Unit which has two sub-components. One sub-component is the Array Control Unit (ACU) which is used to control the operation of the PE Array as well as communications among the PEs and communications between the PEs and the rest of the Maspar system. As a dedicated programmable control processor, the ACU is the only place where instructions are issued and decoded. This division of labor allows for maximum efficiency. The ACU contains its own separate data and instruction memories and has 4 Gigabytes of demand-paged virtual instruction memory, allowing it to operate

independently from the UNIX front-end system. Another sub-component is the Processing Element Array. Each PE in the array is a Maspar designed, register-based, load/store CMOS RISC processor with 40 registers, in addition to its 64kbytes of local RAM and an execution logic. All of the PEs in the array work synchronously to execute instructions from the ACU.

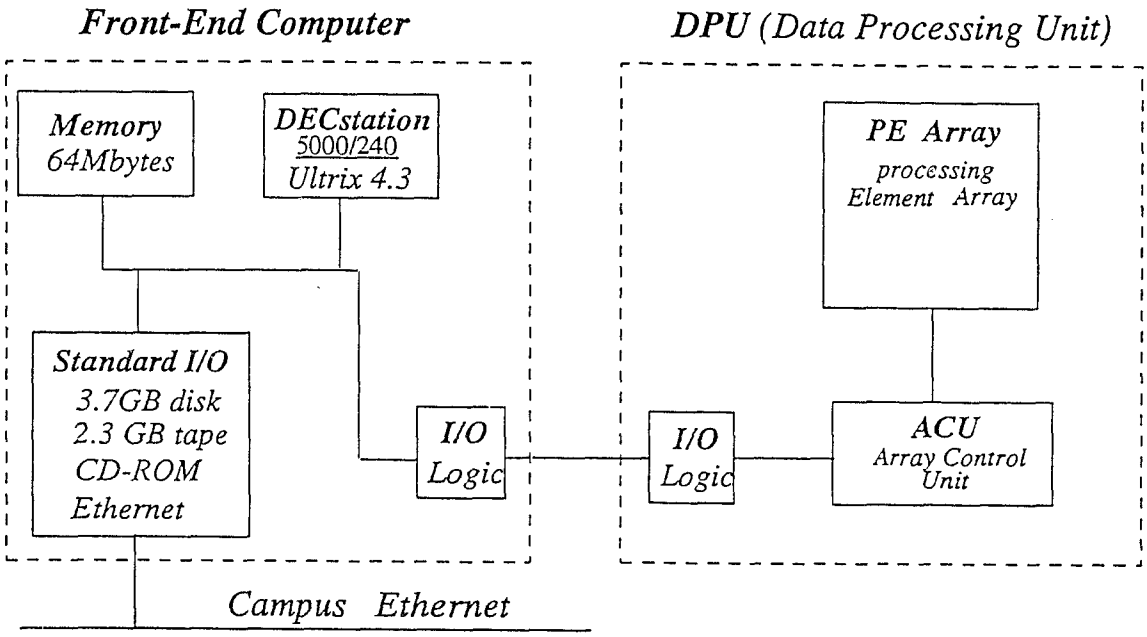


Fig. 5.1 The architecture of the Maspar machine

The second component is the interprocessor communication within the machine. A Maspar provides three highly efficient mechanisms for communicating with elements in the PE Array. The communication between ACU and PEs is performed by a special bus system. The ACU broadcasts values to all processors in the array simultaneously and performs global reductions on parallel data to recover scalar values from the array. The second mechanism, X-Net, supports high-speed data movement to and from the eight nearest neighbors of each PE. The method is very useful for moving data arranged in a uniform array. Its peak bandwidth of communication is about 12

Gigabytes per second. The Global Router mechanism, the third type of the communication method, handles communications of arbitrary connections among processors within the PE array. By providing a multi-stage hierarchical crossbar switch, the Router concurrently establishes links, each of which will enable any two processors to send or fetch data. The connection mechanism supports up to 1024 simultaneous links and each link operates at an aggregate bandwidth of 750 Megabytes per second. Since Router is much slower than X-Net, X-Net is preferred.

The programming languages are the third component of the system. Besides the Assembly and Fortran 77 programming languages, Maspar provides MPL (Maspar Programming Language) which is an extension of ANSI C. MPL supports two data types. One is the regular C data type. Another is the extension for handling data parallel processing. This type is called *plural data type*. A plural data type can be used to specify variables or data that are located in the local memory of the array of PEs. A plural variable is found in every PE, and the operations with the plural variables are plural operations. Plural operations can be performed on each PE in a SIMD manner [A93]. Functions of the type of plural can be defined, stored and executed on PEs as well. In addition, the plural "for" and "while" allow users to implement SIMD control flow for the data that is distributed on PEs.

The last component is the front-end machine. The Maspar uses a DECstation 5000 to manage program execution, user interface, file storage and network communications. When there is a need for a massively parallel execution, it invokes the ACU and the PE Array. This scalar processor runs ULTRIX, Digital Equipment Corporation's UNIX operating system, and includes Ethernet hardware and TCP/IP, NFS, and DECwindows software, which are based on the X Window System, Version 11. This front-end computer also features 64 Megabytes of memory, 3.7 Gigabytes of disk storage and a 2.3 GByte 8mm tape.

5.2 PCHSS Implementation System

The *Parallel Convex Hull Simulation System(PCHSS),* was developed to implement both the algorithms designed in this research and the algorithms designed by previous researchers. Using PCHSS, we can analyze and compare the performance of the various algorithms. This system consists of three portions. An interface system is used to obtain data. The data is then sent to the Maspar through the communication system of PCHSS. The parallel computing system on the Maspar is in charge of computing the convex hull of data received from the interface system. As soon as the computation is finished, the result is sent back to the interface system through the communication system. The interface system takes the result and shows the convex hull on the screen. The system structure of PCHSS is shown in Fig. 5.2.

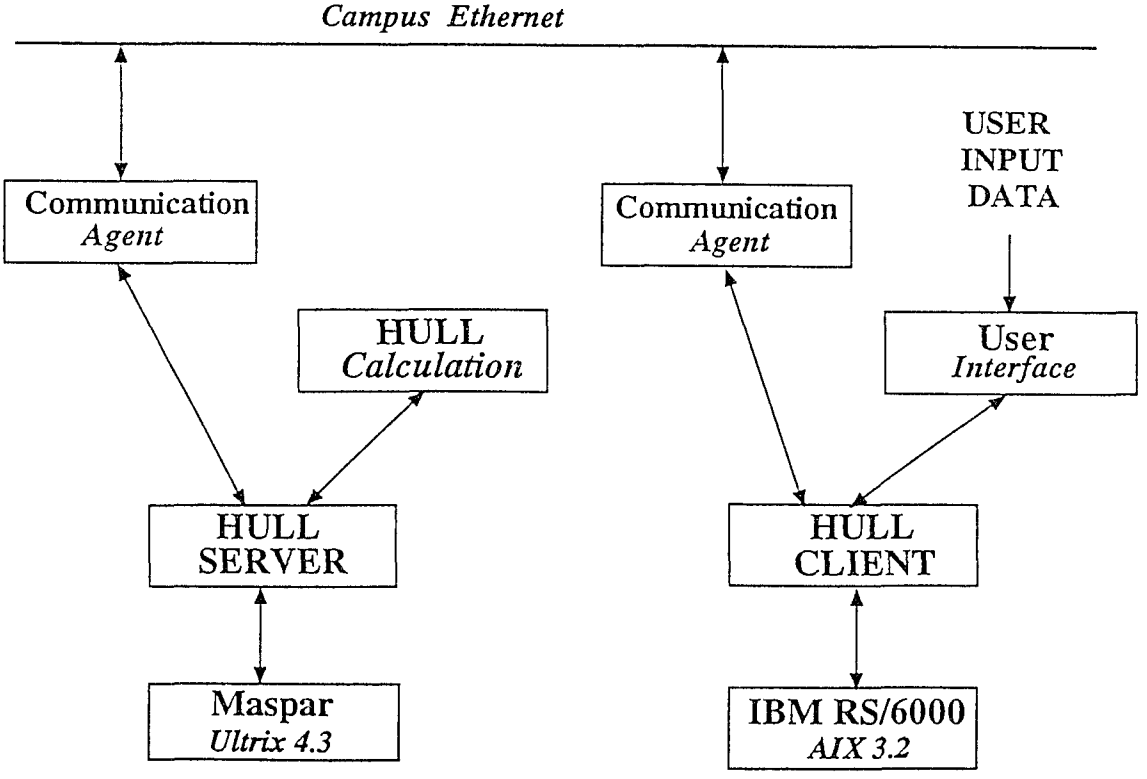


Fig. 5.2 PCHSS system structure

5.2.1 Interface System of PCHSS

The interface system resides on an IBM RS/6000 Model 590 which uses the AIX 3.2.5 operating system with 512 Megabytes main memory. The program is written in X-window R11 with OSF/Motif Release 1.2 [HF94]. The screen of the interface program, shown as Fig. 5.3, has two parts. One part is the drawing area with 500 by 600 pixels. The other part is the function button panel attached on the left hand side of the drawing area. There are seven function buttons and the functions of each button are explained as follows.

Draw: This button is used to allow users to draw points on the drawing area. The user can move the cursor to the place where a point is desired and click the left button of the mouse. The system displays a small cross in that place to indicate that a point has been obtained. There is a small window below the button panel, where the number of the points obtained so far is displayed. As the user clicks the mouse, the number in that small window increases accordingly. The advantage of this type of input is to allow users to define the worst case, the best case and extreme cases of the convex hull problem.

Auto: Besides using *Draw* to obtain points, a user can use the button *Auto* to obtain a set of points randomly generated by the system. When the button *Auto* is pressed, the system displays a small window and asks for a number. This number is the number of the points generated by the system. The advantage of this type of input is time and efficiency. The maximum number of input points for the current PCHSS system is 8192.

Hull: When a set of the points is obtained, the user can click the button *Hull* to send the set of data to the parallel machine. As soon as the calculation is done, the resulting set of data is passed back to the interface system over the network. The interface system uses this calculated data set to draw the convex hull on the drawing area.

There are three sub-buttons under the button *Hull*. Each subbutton corresponds to a particular parallel convex hull algorithm running on the parallel machine or a sequential convex hull algorithm running on a single PE. *M-1* indicates the Odd-Even convex hull algorithm proposed in chapter 3. Holey and Ibarra's algorithm is introduced by the button *H-1*. The sequential algorithm can be invoked by pressing the button *S-1*.

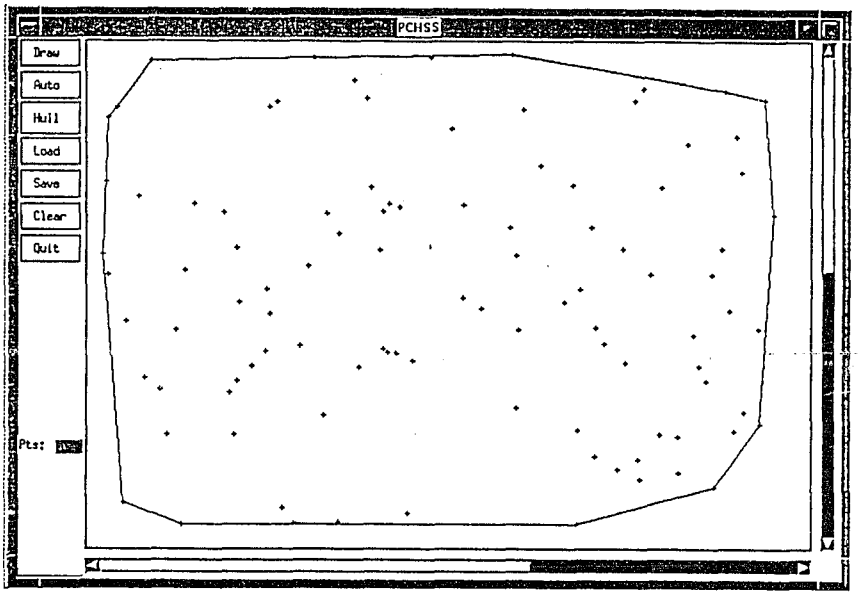


Fig. 5.3 Interface system of PCHSS

Save: To save the current data set to a file, the user clicks the *Save* button. After the *Save* button is pressed, the system pops up a window and asks for a file name. As soon as a file name is given and the button "OK" is clicked, the system saves the file and returns to the previous status.

Load: An existing data file can be loaded from the disk through the button *Load*. After the loading file window pops up, users can change the current directory as they wish, move the scrolling bar and highlight a file name they want. As soon as the *OK* button is clicked, the current highlighted file is loaded to the system and displayed on the drawing area.

Clear: This button is used to remove everything from the drawing area. It works as an initialization agency.

Quit: The only exit from the interface system is to click the *Quit* button.

The interface system flowchart is shown as in Fig. 5.4.

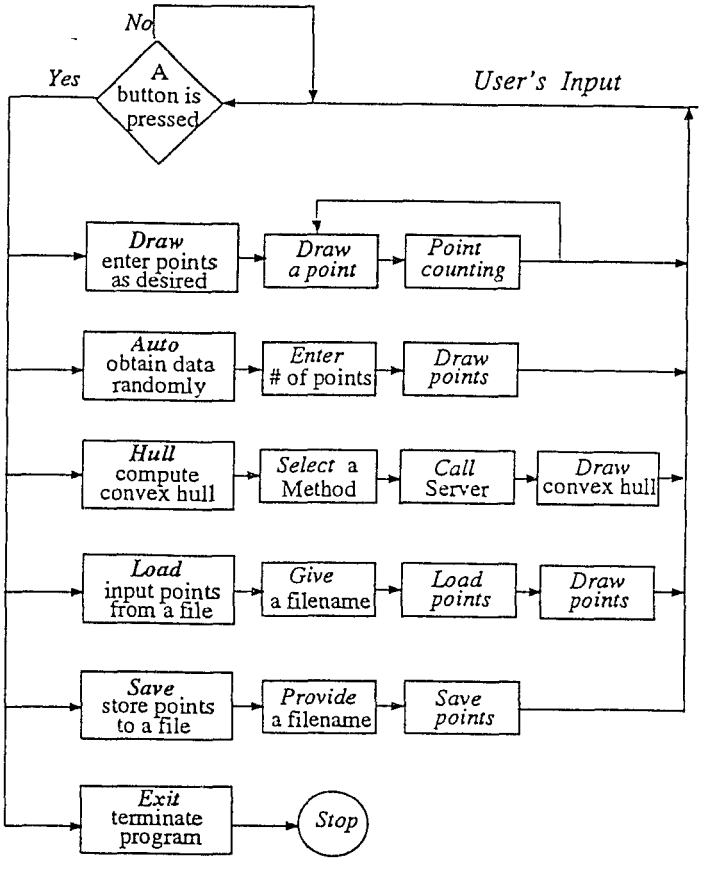


Fig. 5.4 Flowchart of Interface system of PCHSS

5.2.2 Communication system

The PCHSS system is a client/server system. The parallel computing system which is running on the Maspar works as a server system. The interface system acts as a client program. The communication between the client program (node) and the

server program (node) is established over the campus Ethernet using the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol with the Unix Socket mechanism[CS93] provided by the AIX 3.2.4 and the Ultrix 4.3.

When a method to compute the convex hull is selected through the *Hull* button in the interface system, the communication system is invoked. This system sends a message to the parallel computing system over the network and tells the server program which algorithm should be used to compute the convex hull for the data provided by the client program. After the acknowledgement from the server node, the client node sends the data to the server. When the server receives the data from the client, it invokes the corresponding program to calculate the convex hull. As soon as the computation is completed, the server notices the client and sends the result of the computation (the convex hull of the input points) back to the client. The interface system on the client node then reads the data and draws the corresponding convex hull in the drawing area.

5.2.3 Parallel Computing System of PCHSS

The programs running on the Maspar are written in the MPL. Since the processor array is arranged as a 128 (columns) by 64 (rows) array and the X-Net can only be associated with a constant or a non plural variable, we use the X-Net as a communication method between PEs when the number of the points is less than 128. In order to see the performance of the algorithms on numbers of points greater than 128, we use the Router mechanism as a communication method between PEs because the Router can be used with a plural variable. In addition to the control program of the parallel computing system, there are three sub-programs written in this system. The first program is written for Holey's algorithm and the second program is developed for the Odd-Even convex hull algorithm. The third program is written for the sequential algorithm proposed in Chapter 2. Since there is a limitation of the local memory (64K) on

a single processing element, this sequential program is run on a particular PE with at most 120 points. The parallel computing system layout is provided in Fig. 5.5.

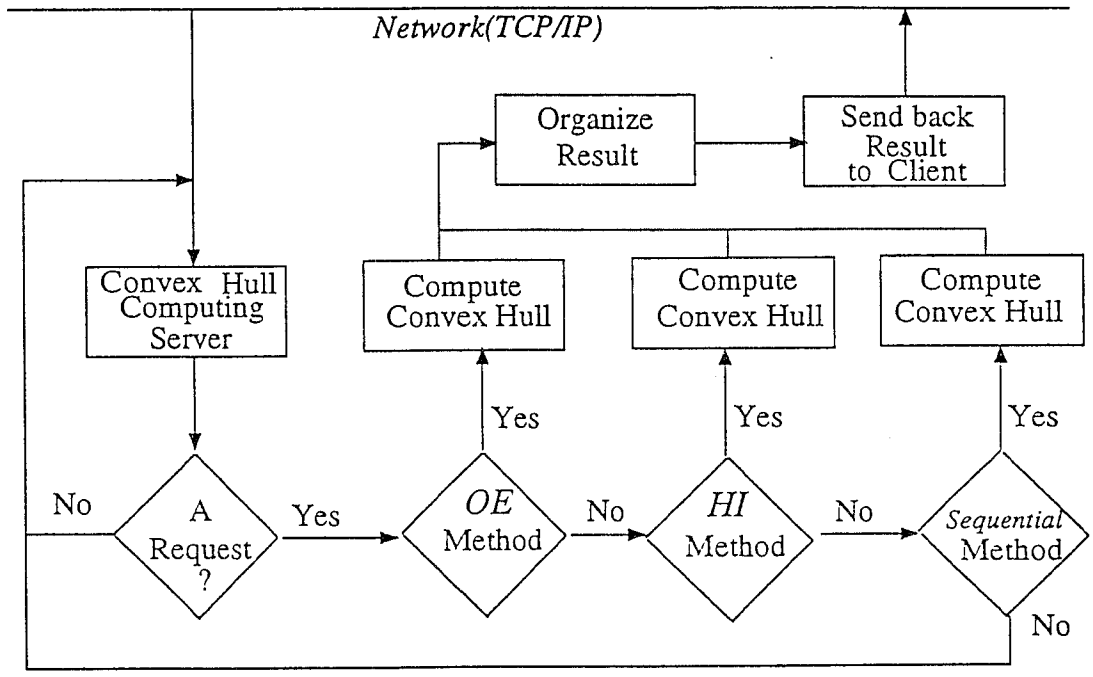


Fig. 5.5 Layout of the parallel computing system of PCHSS

Besides a communication program running on the client side, a communication program runs on the parallel machine. It is used to obtain requests from other machines (clients). As soon as a request is received, the communication program invokes the parallel programs accordingly. The communication program is in charge of noticing the requesting node, and passing the calculated data to the requesting node.

5.3 Analysis and Discussion

We first discuss the test cases and the strategies used to obtain the test data. We then analyze the speed-ups measured for the algorithms and the methods to balance

the results. Finally, the analysis of the performance of the Odd-Even convex hull and Holey's algorithm on the Maspar machine is presented.

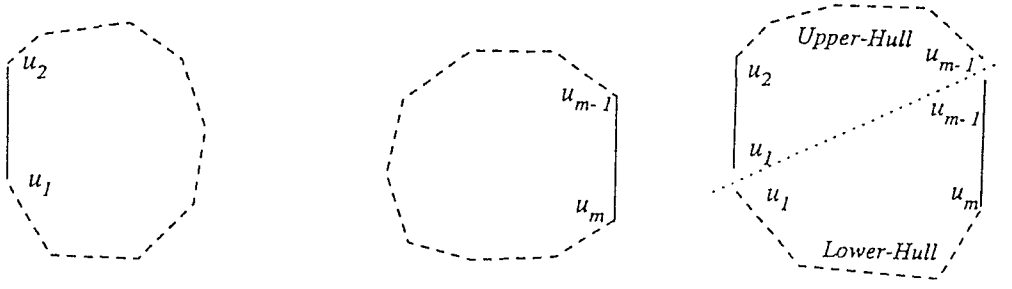
5.3.1 Test Strategies and Testing Cases

The test strategies adopted here are the random-data-collection and the designed-data-collection which are based on the data generating tools provided in the PCHSS. With the strategy of the random-data-collection, the points are generated unexpectly, such as two or more points with the same x-coordinates. This case is not allowed because it fails on the assumptions made for both algorithms, the Odd-Even convex hull and Holey's. In order to accommodate the random-data-collection, a modification is needed for both Odd-Even and Holey's algorithms.

If points which have the same x-coordinates are internal to a convex hull, they are removed as the convex hull is built; however, if those points are on the edges of the convex hull, the construction of the convex hull will be incorrect if the points are not ordered properly. To solve this problem, we add an extra constraint to the algorithms. If two points have the same x-coordinates, their y-coordinates join the comparison. The point with the smaller y-coordinate is considered to be the point which has the smaller x-coordinate.

To show that the constraint properly handles the problem, we need to consider the following two cases. If two points, say u_1 and u_2, have the same x-coordinate which is the smallest x-coordinate of a set of points as shown in Fig. 5.6(a), u_1 is considered to be smaller one according to the constraint because u_1 has a smaller y-coordinate than u_2 does. As a result, for any vertex, say u_i, on the upper hull, (u_1, u_2, u_i) forms a right turn. On the other hand, for any vertex, say u_j, on the lower hull, (u_1, u_2, u_j) does not form a left turn. Therefore the point u_1 is the break point for the upper and lower hulls on the left hand side. The same analysis applies to the second case in which two points have the same x-coordinate which is the largest

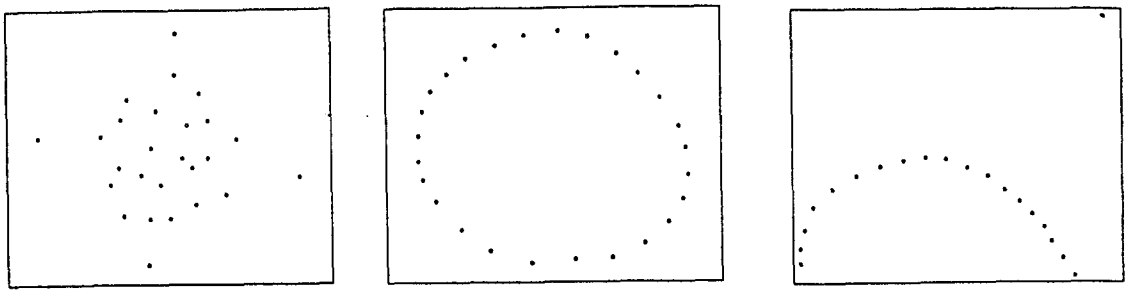
x-coordinate in a set of points as shown in Fig. 5.6(b). Since (u_i, u_m, u_{m-1}) does not form a right turn and (u_j, u_m, u_{m-1}) forms a left turn, only u_{m-1} is on the upper hull and both u_m and u_{m-1} are on the lower hull. Therefore, the break point for the upper and lower hull on the right hand side is u_{m-1}. The corresponding upper and lower hulls is illustrated in Fig. 5.6(c).



(a) The same x-smallest (b) The same x-largest (c) Two halfhulls

Fig. 5.6 Two or more points with the x-smallest or x-largest

We use the designed data collection method to obtain extreme cases. Three cases are considered. In the first case, the majority of points, say more than 90% of the points, are not on the convex hull. In other words, we are expecting less computing time for this case. As the opposite of the first case, the second case is designed for the situation in which the majority of points, say more than 90% of points in a given set, are on the convex hull. In this case, the extreme points are stored in stack and larger stack is required for this case. Thus, more stack operation in the sequential algorithm is expected than the first case. The third case is an extreme case for the construction of the upper hull of a given set of points. In this case, the non-extreme points are eliminated using the point which has the largest x-coordinate. The three cases are shown in Fig. 5.7.



(a) Case 1 (b) Case 2 (c) Case 3

Fig. 5.7 Designed cases for the performance analysis

5.3.2 Speed Up Analysis

The speed up of a parallel algorithm is the ratio of the time complexity of the fastest sequential algorithm to the time complexity of the parallel algorithm. We know that the lower bound for the sequential convex hull algorithms is $O(nlogn)$ and the time complexity for the parallel convex hull algorithms on a linear array is $O(n)$. Therefore, theoretically the speed up of the parallel convex hull algorithm on a linear array should be $O(logn)$.

From a practical point of view, we used two methods to obtain the data points. First, we randomly generated 30 sets of points with the size of 100 each and measured the time consumed by the sequential convex hull algorithm, the Odd-Even convex hull and Holey's algorithm as shown in Table 5.1 where "Odd-Even" stands for Odd-Even convex hull algorithm and "Holey" stands for Holey's algorithm. The time recorded in the table is the time used to compute the convex hull. In other words, the time used for loading points to each of PEs and the time used for collecting points from PEs are not included. The speed ups for the both Odd-Even and Holey's algorithms are listed in Table 5.1 and the diagram of the speed up is shown in Fig. 5.8.

The second data collection method is used to generate extreme cases. As we discuss in the previous section, three cases can be considered as extreme cases. With case 1, shown in Figure 5.7(a), the size of the stack maintained in the sequential algorithm is very small. As a result, we expect that the time consumed by the sequential algorithm will be small. Thus, if we use this case to measure the speed up, the speed up will be relatively small. With the second case, shown in Figure 5.7(b), the size of the stack used in the sequential algorithm is much larger than the size of the stack used in the first case. Therefore, the time consumed by the sequential algorithm should be greater than the time cost for the first case. In the third, case shown in Figure 5.7(c), the operation on the stack reaches its maximum because all points are pushed into the stack before the last point is encountered. The pop operations are then performed on the stack until the last point in the stack is met. The time required in this case is greater than the other two cases.

We used thirty data sets. The first 10 sets of points are drawn from the first case with the sizes varying from 4 to 40 with a distance of 4. The second 10 sets of points are taken from the second case with the sizes varying from 44 to 80 with a distance of 4. The third 10 sets of points are based on the third case with the sizes varying from 84 to 120 with a distance of 4. The points were taken as 30 different sizes points They are listed in Table 5.2. The averages of the time consumed by each of algorithms are calculated and the speed ups of the algorithms are shown in Fig. 5.9.

Although we are expecting that the operation on stack has a significant influence on the sequential algorithm, the result shown in Table 2 indicates that the effect from the operation on stack is not significant. For the case 3, the stack is extensively used for constructing the upper hull, but for the lower hull, the operation on the stack is minimum. Therefore, there is not a significant time consuming on average.

Table 5.1 Time cost on randomly generated data

No	Sequential	Odd-Even	Holey	Speed-Up Odd-Even	Speed-Up Holey
1	3.230072	0.210534	0.507454	15.34228	6.365251
2	3.236288	0.210774	0.519562	15.35430	6.228877
3	3.195300	0.210583	0.519562	15.17359	6.149988
4	3.097771	0.210504	0.514066	14.71597	6.026018
5	3.210988	0.214410	0.523404	14.97592	6.134817
6	3.285202	0.210729	0.511555	15.58970	6.421992
7	3.316137	0.214894	0.478998	15.43150	6.923071
8	3.208301	0.210519	0.530811	15.23996	6.044149
9	3.164052	0.210988	0.515592	14.99636	6.136736
10	3.354436	0.210924	0.511686	15.90353	6.555653
11	3.132740	0.210639	0.511401	14.87255	6.125800
12	3.328720	0.210549	0.502584	15.80972	6.623211
13	3.342884	0.214830	0.507780	15.56060	6.583331
14	3.329081	0.210924	0.506910	15.78332	6.567401
15	3.198719	0.210519	0.511750	15.19444	6.250550
16	3.354057	0.206613	0.515656	16.23352	6.504447
17	3.362148	0.206613	0.529630	16.27268	6.348107
18	3.432871	0.214590	0.537277	15.99735	6.389388
19	3.418998	0.210519	0.581863	16.24080	5.875950
20	3.309619	0.206613	0.511435	16.01845	6.471241
21	3.989698	0.214410	0.522984	18.60780	7.628719
22	3.286305	0.214830	0.507780	15.29724	6.471907
23	3.198282	0.210924	0.511686	15.16320	6.250478
24	3.241696	0.210924	0.511686	15.36902	6.335323
25	3.263424	0.210519	0.507439	15.50180	6.431165
26	3.230132	0.210924	0.515247	15.31420	6.269094
27	3.329582	0.210924	0.531280	15.78570	6.267095
28	3.338268	0.210924	0.511686	15.82688	6.524056
29	3.238561	0.218316	0.527250	14.83428	6.142363
30	3.261232	0.218316	0.521533	14.93813	6.253165

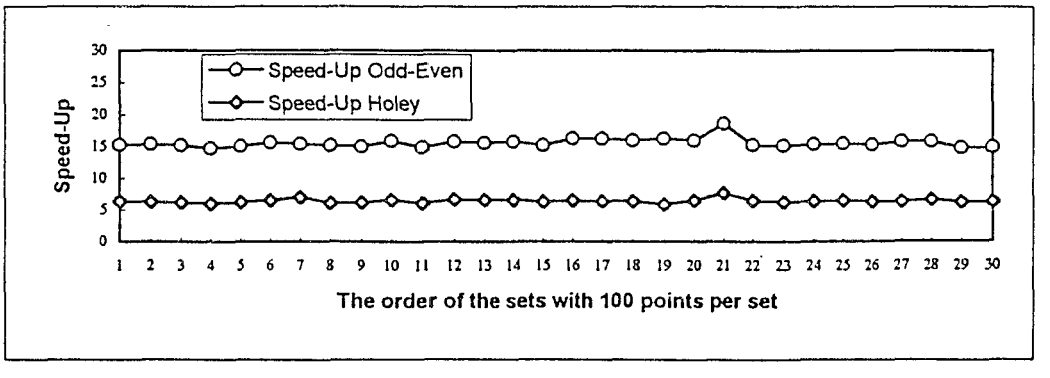


Fig. 5.8 Speed ups on randomly generated data

Table 5.2 Time cost on various sizes of designed data

No. of Points	Sequential	Odd-Even	Holey	Speed-Up Odd-Even	Speed-Up Holey
4	0.011718	0.007812	0.031248	1.50	0.38
8	0.031188	0.015624	0.062376	2.00	0.50
12	0.062361	0.027342	0.097470	2.28	0.64
16	0.105462	0.035154	0.124752	3.00	0.85
20	0.163797	0.042891	0.156304	3.82	1.05
24	0.210924	0.050673	0.179676	4.16	1.17
28	0.261702	0.058470	0.218166	4.48	1.20
32	0.355446	0.062376	0.253890	5.70	1.40
36	0.491226	0.074214	0.292380	6.62	1.68
40	0.553572	0.085767	0.319726	6.45	1.73
44	0.674497	0.093744	0.351540	7.20	1.92
48	0.765704	0.105252	0.394506	7.27	1.94
52	0.909952	0.105462	0.428820	8.63	2.12
56	1.049769	0.117244	0.452226	8.95	2.32
60	1.151468	0.124992	0.511686	9.21	2.25
64	1.348975	0.136710	0.526305	9.87	2.56
68	1.575137	0.144237	0.545854	10.92	2.89
72	1.718768	0.148143	0.592636	11.60	2.90
76	1.831978	0.159831	0.616083	11.46	2.97
80	2.112434	0.167628	0.631816	12.60	3.34
84	2.324219	0.171864	0.632026	13.52	3.68
88	2.525524	0.183237	0.721969	13.78	3.50
92	2.884742	0.195300	0.757828	14.77	3.81
96	3.354445	0.202812	0.959995	16.54	3.49
100	3.546505	0.218301	0.854413	16.25	4.15
104	3.589427	0.226203	0.866191	15.87	4.14
108	3.806911	0.230454	0.873887	16.52	4.36
112	4.180453	0.241707	0.875072	17.30	4.78
116	4.402318	0.245958	0.939580	17.90	4.69
120	4.624882	0.257796	0.978516	17.94	4.73

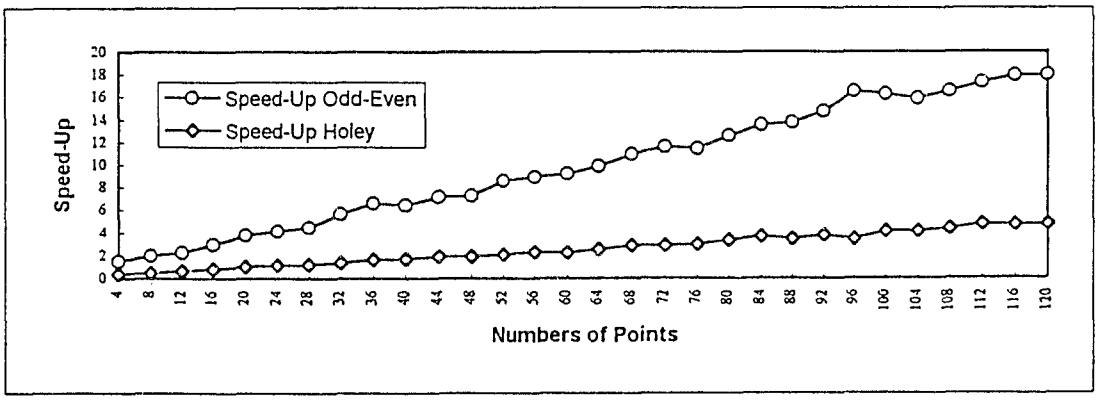


Fig. 5.9 Speed ups on various sizes of designed data

Based on the tables and figures provided above, the Odd-Even convex hull algorithm performs more efficiently than Holey's algorithm with respect to computation speed.

5.3.3 Parallel Performance Analysis

In this section, we compare the Odd-Even convex hull algorithm developed in this research with Holey's algorithm based on performance on the Maspar. To analyze the performance, four tests were designed. Based on those four test, statistical analysis was conducted. For all four tests, we assume that the data was collected from a normal distributed population. The null hypotheses is that there is no difference between the Odd-Even algorithm and Holey's algorithm. The alternative hypotheses is that the Odd-Even algorithm is faster than the Holey's algorithm. Thirty, rather than fifteen recommended in [MM89], observations were collected for each of tests because we wanted to increase the robustness. A statistical inference test is called robust if the probability calculations required are insensitive to violations of the assumptions made.

For a given set of points, both algorithms are used to compute the convex hull on the Maspar. Time required for each of algorithms for computing the convex hull of the set of points was recorded. Since both algorithms were running on the same machine and solving the same problem, the mean of the difference between the times required by the algorithms was used as the statistic. This statistic is called the paired t statistics and is defined as $(\overline{d} - \delta_0)/(S_{\overline{d}}^2/n)^{1/2}$, where $\overline{d}$ is the mean of the sample difference d_i, δ_0 is the population mean difference (usually zero). $S_{\overline{d}}^2$ is the estimated variance of the differences.

The Maspar allows four user programs to run on the machine simultaneously. We wanted to determine if the time required for each of the algorithms was related to the number of the programs running on the system. To that end, a set of 100 points was randomly generated. The algorithms were executed 30 times using the same data set.

The times required for each of the 30 executions were recored. Data generated by this method is referred as "time"-related data.

In addition to the time a program is called, we want to know if the time required for each of the algorithms is related to the locations of the given points. In order to test this case, 30 sets of points were randomly generated with each set having 100 points. This type of data is called "point"-related.

It is possible that the time required for each of the algorithms has a relationship to the size of a data set. Thus, the cardinality of 30 sets of points was defined to range from 4 to 120 in increments of 4. The times required for the algorithms on those data sets were recorded. We call the data obtained by this method "size"-related.

As discussed previously, there are two communication mechanisms for the Maspar, Xnet and Router. The first three test data sets described above are based on the Xnet because less than 128 points are used. In order to determine whether the time required for each of the algorithms was related to the communication mechanisms, we designed the fourth test strategy. Thirty data sets ranging from 250 to 8192 points in increments of 250 except the last two ranges (from 7000 to 7500 and 7500 to 8192) in order to observe what happens if the maximum number of points, 8192, was reached within 30 intervals. We call data obtained by this strategy "communication"-related.

Both the time required for each of the algorithms based on each of four data collection strategies and the corresponding calculation on the difference of time were recorded and are listed in Tables 5.3, 5.5, 5.7 and 5.9. The title "Odd-Even" indicates the Odd-Even convex hull algorithm and "Holey" represents the Holey's algorithm. The data recorded in the tables under these two titles are the times (in second) required for a processing element in the Maspar for computing the convex hull of a set of planar points. The difference between the time requited for the Odd-Even and Holey's algorithms is computed and recorded in the column of "Odd-Even - Holey". The

means, variances, T values and the significances of the T values of the difference between the times required by the algorithms were computed and shown in Table 5.4, 5.6, 5.8 and 5.10, respectively.

Based on the above analysis, neither "time" nor "location" affects the performance of the algorithms. Although the time required for computing a convex hull of a set of planar points increases as the number of points increases, it does not change the trend of the difference between the algorithms. The Router is slower than the Xnet, but it has the same effect on both algorithms. In other words, using the Router does not change the difference between the time required for each of the algorithms for computing the convex hull. As a result, since the one-tailed 0.001 rejection region for t distribution with 29 degrees of freedom is -3.3963, we reject the null hypothesis and conclude that the Odd-Even convex hull algorithm is faster than the Holey's algorithm on the Maspar based on the alternative hypothesis.

It is obvious that the Odd-Even convex hull algorithm uses n processors for n points and Holey's algorithm uses $n/3$ processors for n points. Although Holey's algorithm uses less number of processors, more calculation is required on each processor. On the other hand, there is no explicit step of sorting in Holey's algorithm. The sorting step is embedded in each of iterations of Holey's algorithm. These result in an extensive calculation on a single processor. This is the reason why Holey's algorithm is slower than the Odd-Even convex hull algorithm. Holey's algorithm is good when the number of points is larger than the number of processors. But the algorithm does not provide the solution for the case where the number of points is larger than three times of the number of processors. Since the Odd-Even sort is the foundation of the Odd-Even convex hull algorithm, the algorithm inherits the simplicity of the control scheme from the sorting algorithm and makes it easily executed, efficient, and faster than other algorithm.

Table 5.3 A set of 100 random planar points with 30 runs

Order of runs	Odd-Even	Holey	Odd-Even - Holey
1	0.210924	0.363258	-0.152334
2	0.210924	0.363258	-0.152334
3	0.218736	0.359352	-0.140616
4	0.210924	0.367164	-0.156240
5	0.214830	0.355446	-0.140616
6	0.218736	0.367164	-0.148428
7	0.210924	0.355446	-0.144522
8	0.210924	0.351540	-0.140616
9	0.218736	0.359352	-0.140616
10	0.207018	0.351540	-0.144522
11	0.207018	0.363258	-0.156240
12	0.214830	0.355446	-0.140616
13	0.207018	0.359352	-0.152334
14	0.214830	0.363258	-0.148428
15	0.210924	0.355446	-0.144522
16	0.218736	0.355446	-0.136710
17	0.214830	0.359352	-0.144522
18	0.203112	0.359352	-0.156240
19	0.218736	0.355446	-0.136710
20	0.210924	0.347634	-0.136710
21	0.210924	0.355446	-0.144522
22	0.210924	0.363258	-0.152334
23	0.214830	0.351540	-0.136710
24	0.218736	0.355446	-0.136710
25	0.214830	0.359352	-0.144522
26	0.218736	0.355446	-0.136710
27	0.210924	0.359352	-0.148428
28	0.218736	0.367164	-0.148428
29	0.207018	0.355446	-0.148428
30	0.207018	0.355446	-0.148428

Table 5.4 Test result for the "time"-related data

Number of Observations	Mean	Variance	T - Values	Probability > | T |
30	-0.1453032	0.000404	-125.2050818	0.0001

Table 5.5 30 sets of random planar points with the size of 100 each

Order of point sets	Odd - Even	Holey	Odd-Even - Holey
1	0.210924	0.355446	-0.144522
2	0.214830	0.355446	-0.140616
3	0.210924	0.359352	-0.148428
4	0.218736	0.359352	-0.140616
5	0.214830	0.355446	-0.140616
6	0.210924	0.347634	-0.136710
7	0.214830	0.359352	-0.144522
8	0.210924	0.363258	-0.152334
9	0.203112	0.363258	-0.160146
10	0.207018	0.355446	-0.148428
11	0.203112	0.363258	-0.160146
12	0.218736	0.359352	-0.140616
13	0.203112	0.355446	-0.152334
14	0.210924	0.363258	-0.152334
15	0.207018	0.359352	-0.152334
16	0.210924	0.351540	-0.140616
17	0.203112	0.355446	-0.152334
18	0.214830	0.359352	-0.144522
19	0.207018	0.351540	-0.144522
20	0.210924	0.359352	-0.148428
21	0.210924	0.367164	-0.156240
22	0.207018	0.363258	-0.156240
23	0.214830	0.355446	-0.140616
24	0.214830	0.355446	-0.140616
25	0.203112	0.359352	-0.156240
26	0.210924	0.355446	-0.144522
27	0.203112	0.359352	-0.156240
28	0.214830	0.355446	-0.140616
29	0.210924	0.363258	-0.152334
30	0.214830	0.355446	-0.140616

Table 5.6 Test result for the "point"-related data

Number of Observations	Mean	Variance	T - Values	Probability > | T |
30	-0.1476468	0.0000467	-118.316399	0.0001

Table 5.7 30 various sizes of sets of planar points (from 4 to 120 points)

Number of Points	Odd-Even	Holey	Odd-Even - Holey
4	0.007812	0.031248	-0.023436
8	0.015624	0.062376	-0.046752
12	0.027342	0.097470	-0.070128
16	0.035154	0.124752	-0.089598
20	0.042891	0.156304	-0.113413
24	0.050673	0.179676	-0.129003
28	0.058470	0.218166	-0.159696
32	0.062376	0.253890	-0.191514
36	0.074214	0.292380	-0.218166
40	0.085767	0.319726	-0.233959
44	0.093744	0.351540	-0.257796
48	0.105252	0.394506	-0.289254
52	0.105462	0.428820	-0.323358
56	0.117244	0.452226	-0.334982
60	0.124992	0.511686	-0.386694
64	0.136710	0.526305	-0.389595
68	0.144237	0.545854	-0.401617
72	0.148143	0.592636	-0.444493
76	0.159831	0.616083	-0.456252
80	0.167628	0.631816	-0.464188
84	0.171864	0.632026	-0.460162
88	0.183237	0.721969	-0.538732
92	0.195300	0.757828	-0.562528
96	0.202812	0.959995	-0.757183
100	0.218301	0.854413	-0.636112
104	0.226203	0.866191	-0.639988
108	0.230454	0.873887	-0.643433
112	0.241707	0.875072	-0.633365
116	0.245958	0.939580	-0.693622
120	0.257796	0.978516	-0.720720

Table 5.8 Test result for the "size"-related data

Number of Observations	Mean	Variance	T - Values	Probability > | T |
30	-0.3769913	0.0489111	-9.3365859	0.0001

Table 5.9 30 various sizes of sets of planar points (from 250 to 8192 points)

Number of Points	Odd-Even	Holey	Odd-Even - Holey
250	0.71445	1.63674	-0.92230
500	1.77665	3.50003	-1.72338
750	2.64942	5.31457	-2.66516
1000	3.55453	7.46885	-3.91433
1250	4.45265	9.71782	-5.26517
1500	5.33512	11.71406	-6.37895
1750	6.23524	13.80013	-7.56489
2000	7.12486	15.79218	-8.66732
2250	8.00740	17.76250	-9.75510
2500	9.17446	19.90393	-10.72946
2750	10.11810	21.90168	-11.78358
3000	11.05310	23.86644	-12.81334
3250	11.99470	25.84697	-13.85227
3500	12.90664	27.79177	-14.88513
3750	13.86632	29.85646	-15.99014
4000	14.80049	31.66725	-16.86677
4250	15.71790	33.97183	-18.25392
4500	16.63084	35.87042	-19.23959
4750	17.57557	38.05145	-20.47589
5000	18.51870	39.60892	-21.09022
5250	19.43128	41.70046	-22.26917
5500	21.24949	43.75873	-22.50924
5750	21.31518	45.77924	-24.46407
6000	22.22322	48.06804	-25.84482
6250	23.15106	49.09530	-25.94423
6500	24.09254	51.80428	-27.71175
6750	24.98586	53.74928	-28.76342
7000	25.90205	55.55965	-29.65761
7500	27.79735	60.66553	-32.86818
8192	30.39424	66.59324	-36.19900

Table 5.10 Test result for the "communication"-related data

Number of Observations	Mean	Variance	T - Values	Probability > | T |
30	-16.6356121	95.404624	-9.3285619	0.0001

5.4 Summary

In order to analyze the performance of various convex hull algorithms, we developed the *PCHSS* system. In addition to the random data collection and the designed data collection methods, four other data collection strategies were defined for analyzing the performance of the algorithms from a statistical point of view.

The "time"-related strategy is used to determine whether the time required for each of the algorithms for computing a convex hull is related to the clock time at which a program is submitted. Using the "point"-related strategy, we determine the effect of the positions of the points used in computing a convex hull. Whether the size of a set of points ("size"-related) affects the time required for each of the algorithms for constructing a convex hull is the third consideration. Finally, we want to determine if the communication mechanisms affects the time required for each of the algorithms ("communication"-related). Based on the statistical analysis, at a 99.9% confidence level, the Odd-Even convex hull algorithm is faster than Holey's algorithm.

The Odd-Even convex hull algorithm is faster than Holey's algorithm because it requires less comparison time. According to the Odd-Even Transposition Sort, each processor, odd or even, uses $n/2$ time to compare two values each time. Therefore the total time required by the sorting step is n because the operations on odd processors are followed by the operations on even processors. In contrast, Holey's algorithm as shown in Chapter 3 needs $(4 * n + 2)/3 * 5)$ comparisons, which ranges from $6n$ to $7n$ comparisons. According to Holey's algorithm, the variable *left* is compared with the variable *small* and the variable *right* is compared with the variable *large*. Then the points stored in the variables *small*, *middle* and *large* are sorted by using at least three comparisons. That is, for each step of the algorithm, at least 5 comparisons are made for the calculation. Since Holey's algorithm requires $(4 * n + 2)/3$ steps, the total comparisons needed are $(4 * n + 2)/3 * 5)$, which is $(20 * n + 10)/3$.

In addition, the Odd-Even convex hull algorithm costs less time than Holey's algorithm to make the function calls ("elimination" function for Holey's algorithm and "slope-comparison" function for the Odd-Even convex hull algorithm). In the Odd-Even convex hull algorithm, each odd or even processors makes at most n function calls for comparing slopes. Thus the total time for the function calls is $2n$. On the other hand, there are three possible optimization function calls on each processor in each of $(4 * n + 2)/3$ loops for Holey's algorithm. Therefore, the total $(4 * n + 2)/3 * 3$, which is about $4n$, time is required for the optimization calls in the Holey's algorithm. Thus, the Odd-Even convex hull algorithm uses one half of the number of the function calls required for Holey's algorithm. In other words, for a given set of planar points, the Odd-Even convex hull algorithm consumes less time for the function calls than Holey's algorithm.

In Holey's algorithm, each processor obtains three points at the initial step. Thus, Holey's algorithm needs $n/3$ processors to compute the convex hull of n planar points. But for a Maspar machine, a program is given either the entire PE array or none of the PEs. Therefore, if the number of the points is less than the number of PEs, there is no reason to use a slower algorithm to save the number of PEs. In other words, if the number of given points is less than the number of processors in the Maspar, it is better to use the Odd-Even convex hull algorithm because it is faster than Holey's algorithm.

Another reason to use the Odd-Even convex hull algorithm is that only one sorting step is actually required for constructing the convex hull of a given set of planar points. In both the Odd-Even convex hull and Holey's algorithms, the convex hull is constructed by building the upper hull and lower hull separately. In the Odd-Even convex hull algorithm, after the sorting step, the set of sorted points can be saved for computing the upper hull as well as the lower hull. Other the other hand, since there is no explicit sorting step in Holey's algorithm, the comparisons among points must be done

for the computation of both the upper hull and the lower hull, respectively. This causes that Holey's algorithm is slower than the Odd-Even convex hull algorithm.

In addition to the SIMD machine, if the Odd-Even convex hull algorithm is applied to an MIMD (Multiple Instruction and Multiple Datastream) machine, the Odd and Even processors can be used to compute the convex hull of a set of planar points simultaneously. While the Odd(Even) processors compute the upper hull, the Even(Odd) processors work on the lower hull.

Chapter 6

Conclusion

In this chapter, we first summarize the research. The contributions made by this research are presented in section 6.2. Finally, in section 6.3, we discuss future research.

6.1 Summary of the Dissertation

Computational geometry is a branch of computer science which addresses the design and analysis of algorithms for solving geometric problems. The convex hull problem is one of the fundamental problems in computational geometry with a wide range of applications, such as computer graphics, simulation, and pattern recognition.

The approaches used to design parallel convex hull algorithms can be classified into two categories: the divide-and-conquer paradigm and the iterative paradigm. Due to its recursive concept, the divide-and-conquer paradigm is commonly used in the design of parallel convex hull algorithms; however, such algorithms are difficult to implement on modern parallel machines because of the complexity of the recursive merge step involved in the algorithms.

The second approach is to refine the design typically found in sequential convex hull algorithms to convert them into interactive parallel convex hull algorithms. These algorithms have the advantage of simplicity and ease of implementation on modern parallel machines. The second approach was chosen for designing the parallel convex hull algorithms in this research.

We first presented a new sequential convex hull algorithm. We then converted that sequential algorithm to a parallel algorithm on a linear array architecture. The linear array convex hull algorithm was further generalized from two aspects, one aspect is the case where a two or more dimensional mesh-array is given and the second

aspect is the case where the number of the points is greater than the number of the processors. To evaluate the performance of the parallel convex hull algorithms, a system called the *Parallel Convex Hull Simulation System* was developed. Using this system, performance data for the convex hull algorithms was collected and extensive analysis was conducted.

In previous sequential algorithms, the order of polar angles of points and the order of x-coordinates were commonly used in the first steps of the algorithms. By using the ordering relationship of the slopes of points, we developed an optimal and simplified sequential convex hull algorithm. The algorithm constructs four quarters separately instead of constructing two half-hulls. Since the edges connecting two adjacent vertices in a quarter must be monotonically increasing or decreasing, the decision of whether or not a vertex should be included in the partial quarter currently under construction is easier than that for the partial half-hull, as can be observed in procedures *QUARTERHULL* and *HALFHULL* in Chapter 2. This algorithm was initially designed for solving convex hull problem of a simple polygon. We then extended it to handle any set of planar points without changing its time complexity. The algorithm has two steps. The first step is to sort the points according to their x-coordinates. The second step is to sort slopes among the sorted points. This sorting approach is the foundation for the new parallel convex hull algorithms presented in Chapter 3 and 4.

To convert the sequential algorithm to a parallel algorithm, we followed the design idea in the sequential algorithm and applied it to the parallel situation. Since the first step of sorting is based on the coordinates of points, any parallel sorting algorithm is applicable. To compare the ordering relationship of slopes, we selected the Odd-Even transposition sort algorithm with a modification because the ordering relationship of slopes is decided by three points instead of two points. This Odd-Even linear array convex hull algorithm uses $O(n)$ processors in $O(n)$ time, which is optimal.

In addition to the Odd-Even linear array convex hull algorithm, we also developed versions of the Odd-Even convex hull algorithm for a two or more dimensional mesh-array architecture and for the case where the number of the points is greater than number of the processor. The algorithm for the multi-dimensional mesh-array uses $O(n)$ processors with $O(d^2\ n^{1/d})$ in time where $n \leq p$ and $O((n/p)\log(n/p) + np^{1/d-1})$ in time when $n \gg p$. Both algorithms are optimal.

PCHSS was developed to evaluate and compare the performance of the algorithms designed in this research to the algorithms developed previously. The system is client/server based. The parallel machine is the Maspar. The front-end machine is an IBM RS/6000 clusters where the client system, the window for collecting points and supported by the X-window/Motif, resides. The communication between the client and server system is accomplished by the Ethernet with the TCP/IP protocol and UNIX Socket mechanism.

Since a sequential algorithm is involved in measuring the speed-up, some extreme cases may affect the computation time for the sequential algorithm because the stack is used extensively in some extreme cases. Therefore, to evaluate the speed-up, the sets of points were collected based on two strategies, random-point-collection and designed-point-collection. With the random-data-collection strategy, the time required by the algorithms for computing the convex hull on each of 30 sets of 100 randomly generated points were recorded. With the designed-data-collection strategy, three cases with 10 point sets per case, a total of 30 point sets were generated and the time required for each of the algorithms for computing the convex hull for each of the cases was recorded. Both experiments show the Odd-Even convex hull algorithm has a higher speed-up than Holey's algorithm regardless of whether the set of points is generated randomly or by design. To further analyze the performance of the algorithms we used four methods to obtain test data. These four methods are called

"time"-related, "point"-related, "size"-related and "communication"-related. We wanted to see whether the time the program was invoked ("time"-related), the locations of the points ("point"-related), the sizes of the sets of points ("size"-related) and the communication mechanisms affected the time required to compute the convex hull. Based on the experimental results, to a 99.9% confidence level, the Odd-Even convex hull algorithm is more efficient than Holey's algorithm when the number of the points is less than the number of PEs in the Maspar.

6.2 Significance of the Research

The convex hull problem has a very close relationship to sorting problems. Sorting against the polar angles of points or x-coordinates of points is commonly used in many sequential and parallel convex hull algorithms as the first step of the construction of the convex hull of a set of planar points. Since the sorting against polar angles of points is more difficult than the sorting against x-coordinates of points, the sorting against x-coordinates is often used as the first step of the convex hull algorithms. By observing the ordering relationship among the slopes of the vertices of a convex hull, this research introduced a new method to design sequential and parallel convex hull algorithms. By applying this method to the convex hull problem of a simple polygon, we have presented an optimal, simple and practical algorithm.

The algorithm designed to construct the convex hull of a simple polygon uses $O(n)$ time, which is optimal. By using the quarter partition instead of the half partition, our sequential algorithm is simpler conceptually and easier to implement. In previous algorithms [GY83, L83], four vertices and three reference lines are used for constructing the convex hull of a simple polygon. In our algorithm, only three vertices and two reference lines are needed. Consequently, our algorithm is more efficient than the previous algorithms. In addition, since the convex hull is built by quarters, some points pushed onto the stack S in the previous algorithms will not be put onto the stack

S in our algorithm. As a result, the algorithm designed in this research saves time by reducing the number of stack operations.

The algorithm designed to construct the convex hull of a simple polygon was extended for constructing the convex hull of a set of planar points. If a set of planar points is sorted against their x-coordinates, the result of the sorted list of points is considered an upper chain as well as a lower chain of a simple polygon. After determining the point which has the largest y-coordinate and the point which has the smallest y-coordinate, we apply the algorithm designed for the simple polygon to this list of sorted points. This algorithm costs $O(nlogn)$, which is optimal. With four partition rather two partition used in the previous algorithms, the algorithm designed in this research saves time by reducing the number of stack operation.

Using the design idea presented in our sequential algorithm, we developed an Odd-Even parallel convex hull algorithm using the ordering relationship of the slopes of points. This algorithm uses $O(n)$ processors in $O(n)$ time to construct the convex hull of a set of n planar points, which is optimal. The significance of the new parallel convex hull algorithm is its simplicity and extensibility. Previous algorithms need either special chips in order to run the algorithms, such as the algorithms proposed in [C84, CCL87], or more calculation time, such as the algorithm proposed in [HI92]. In [C84], two different linear array architectures are required. One for identifying the extreme points and another for sorting the extreme points. Although only one linear array architecture is required in [CCL87], a special processor, called collector, has to be connected to the linear array to control the calculation of the convex hull. In the algorithms designed in this research, since a sorting algorithm is used to handle the construction of the convex hull of a set of planar points with proper modification, the control scheme is simple and no special chip is required. The newly designed algorithm has better performance than previous algorithm because it requires less

calculation in a single processor of a linear processor array. Since the algorithms designed in [C84] and [CCL87] are based on some special arrangement of processors, the comparison of the performance is focused on Holey's algorithm and the Odd-Even convex hull algorithm.

Holey's algorithm is designed for a general linear array, but for a single processor it requires more calculations than the Odd-Even parallel convex hull algorithm. First, Holey's algorithm uses (20 * n + 2)/3, about $6n$ to $7n$, point comparisons whereas in the Odd-Even parallel convex hull algorithm, only n point comparisons are needed. In other words, Holey's algorithm requires as many as six to seven times more point comparisons than the Odd-Even parallel convex hull algorithm. Secondly, the Odd-Even convex hull algorithm costs less time than Holey's algorithm to make the function calls. In the Odd-Even convex hull algorithm, $2n$ function calls are required; however, $(4 * n + 2)$ function calls are required in Holey's algorithm. Thus, the Odd-Even convex hull algorithm requires only 50% of the number of the function calls required for Holey's algorithm. As a result, the Odd-Even convex hull algorithm requires less time for comparing points and calling function than Holey's algorithm and is therefore more efficient.

To extend the Odd-Even parallel convex hull algorithm to handle more generalized cases, we proposed two additional algorithms. First, if the number of points n is less than the number of the processors p, the Odd-Even parallel convex hull algorithm can be used directly with $p - n$ processors holding the point which has the smallest x-coordinate. If the number of points is greater than the number of the processors, the Odd-Even Merge-Split sort is used in the sorting step and the convex hull is constructed by computing the sub-hulls in each individual processor using our sequential algorithm and the bridges of sub-hulls between the processors. The algorithm takes $O((n/p)log(n/p)) + O(n)$ because the sorting step of the algorithm takes

$O((n/p)log(n/p))$ + $O(n)$ time and the second step of the algorithm uses $O(n)$ time with p processors for a set of n points, which is optimal.

In addition, the Odd-Even parallel convex hull algorithm is extended to a two or more dimensional mesh-array convex hull algorithm. If the number of points is less than or equal to the number of processors, the algorithm takes $O(d^2n^{1/d})$ in time with n processors for computing the convex hull of a set of n points. If $n > p$, the algorithm costs $O((n/p)log(n/p) + np^{1/d-1})$ in time with p processors for a set of n points. Both algorithms are optimal.

The generalized algorithms in this research are more efficient than Holey's algorithms. Each processor in our algorithms uses 3 memory cells, whereas Holey's algorithms require 5 memory cells for a linear array and 7 memory cells for d-dimensional mesh-array. Our algorithms require an explicit sorting step but the construction of the upper hull and lower hull can share the result of the sorting step so that the computation time is reduced. On the other hand, since there is no explicit sorting step in Holey's algorithms, the sorting operations must be performed for both the upper hull and lower hull. In addition, in each step of our algorithm, a processor takes two communication operations, whereas four communication operations are required in each step of Holey's algorithms. The result and thus the significance of the algorithms developed in this research are that they are more efficient and have simpler control schemes.

It is important to note that converting an efficient algorithm that is designed for smaller number of processors to an algorithm using more processors with the same or greater efficiency is much more difficult than the reverse - coverting an algorithm designed for larger number of processors to an algorithm for smaller number of processors. It is not clear how to convert Holey's algorithm to the case where $n > 3p$. Therefore, our algorithms can be considered more general than Holey's algorithms.

Many parallel sorting algorithms have been developed for different parallel architectures. The method provided in this research gives a framework for designing parallel convex hull algorithms for multiple architectures.

Although many parallel convex hull algorithms have been proposed, *PCHSS* represents a unique contribution in which the performance of parallel convex hull algorithms can be evaluated with up to 8192 points and 8192 processors at a time. Since the Maspar can have up to 16384 processors, *PCHSS* can be further used to evaluate a set of 16384 points using 16384 processors without changing the program.

In order to describe the performance of their algorithms, Holey conducted a manual experiment with only 23 points. Therefore, based on our review of literature, *PCHSS* is the first performance analysis system for parallel convex hull algorithms.

6.3 Future Research

We anticipate future research in the following three areas. First, we can extend the algorithms by using other sorting algorithms which are based on other platform or architecture of parallel machines. As noted in chapter 3, the key to using a sorting algorithm for solving the convex hull problem is to find a proper third point for the comparison of slopes. Another problem is how to solve the point locality feature which means that each point must be compared with its ordered neighbors. This problem could be solved either by using the axioms introduced in [K92] or by assuming that the input points are in order since a sorting step is considered as the first step of the algorithm. Architecture type is another concern.

Secondly, we plan to investigate the extent to which the new method can be applied to other computational geometry problems, such as triangulation and voronoi diagrams. We plan to determine whether there is an ordering relationship inside such problems. If the triangulation can be solved by several embedded convex hulls, the methods to separate and select points to construct the convex hulls are needed.

Finally, the PCHSS system can be further extended to handle performance evaluation on other computational geometry problems and for other parallel architecture computers. Since the system was written in a modular manner, procedures can be attached to the system in a straightforward manner.

Bibliography

[A89] S. G. Akl, *The Design and Analysis of Parallel Algorithms,* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[A82] S. G. Akl, "A constant-time parallel algorithm for computing convex hulls," *BIT*, Vol. 22, 1982, pp130-134.

[A93] H. M. Al-Ammal, "Parallel Graph Coloring on the MasPar", Master Project, Dept. of Computer Science, LSU, 1993.

[A87] M. D. Atkinson, "An Optimal Algorithm for Geometrical Congruence", *Journal of Algorithms*, Vol. 8, 1987, pp308-315.

[A79] A. M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", *Information Processing Letters*, Vol 9:5, 1979, pp 216-219.

[A78] K. R. Anderson, "A reevaluation of An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set," *Information Processing Letters*, Vol. 7, No. 1, Jan., 1978, pp53-55.

[ACGOY88] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlang and C. Yap, "Parallel Computational Geometry",*Algorithmica,* Vol.3, 1988, pp293-327.

[AG86] M. J. Atallah and M.T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *Journal of Parallel & Distributed Computing*, Vol.3, 1986, pp492-507.

[AL93] S. G. Akl and K. A. Lyons, *Parallel Computational Geometry,* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[AOB93] B. Abali, F. Ozauner and A. Bataineh, "Balanced Parallel Sort on Hypercube Multiprocessors",*IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 5, May 1993, pp572-581.

[B78] A. Bykat, "Convex Hull of a Finite Set of Points in Two Dimensions", *Inform. Process. Lett.,* Vol. 7, No. 6, Oct., 1978, pp201-206.

[BS78a] J. L. Bentley and M. I. Shamos, "Divide and Conquer for Linear Expected Time," *Inform. Process. Lett.,* Vol. 7, No. 2, Feb. 1978, pp87-91.

[BS78b] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers", *IEEE Trans. on Computers,* C-27 (1), 1978, pp84-87.

[C95] D. Z. Chen, "Efficient Geometric Algorithms on the EREW PRAM," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 1, Jan. 1995, pp41-47.

[C84] B. Chazelle, "Computational Geometry on A Systolic Chip," *IEEE Trans. Comput. C-33(9)*, 1984, pp774-785.

[CCL87] G. H. Chen, M. S. Chern, and R. C. T. Lee, "A new systolic architecture for convex hull and half-plane intersection problems," *BIT*, Vol. 27, 1987, pp141-147.

[C81] A. L. Chow, "A Parallel Algorithms for Determining convex hulls of sets of points in two dimensions," *Proceedings of the Nineteenth Annual Allerton Conference on Communication, Control And Computing,* Monticello, Illinois, 1981, pp214-223.

[CG88] R. Cole and M. T. Goodrich, "Optimal parallel algorithms for polygon and point-set problems(preliminary version)," *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry,* Urbana-Champaign, Illinois, 1988, pp201-210.

[CLR90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms,* The MIT Press, 1990.

[CS93] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP*, Vol. III, Prentice-Hall, 1993.

[CS92] R. Cypher and J. L. C. Sanz, "Cubesort: A Parallel Algorithm for Sorting N data items with S-Sorters", *Journal of Algorithms*, Vol. 13, No. 2, June 1992, pp211-234.

[DT93] L. Devroye and G. Toussaint, "Convex Hulls for Random Lines", *Journal of Algorithms*, Vol. 14, No. 3, May 1993, pp381-394.

[G72] R. L. Graham, "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set", *Information Processing Letters*, Vol. 1, 1972, pp132-133.

[G87] M. T. Goodrich, "Finding the Convex Hull of A Sorted Point Set in Parallel", *Inform. Process. Lett.*, Vol. 26, No. 4, Dec. 1987, pp173-179.

[GS93] A. Gibbons and P. Spirakis, *Lectures on Parallel Computation*, Cambridge University Press, 1993.

[GY83] R. L. Graham and F. F. Yao, "Finding the Convex Hull of a Simple Polygon", *Journal of Algorithms*, Vol. 4, No. 4, 1983, pp324-331.

[H93] C. Harris, *The IBM RISC System/6000*, McGraw-Hill Book Company, 1993.

[HF94] D. Heller and P. M. Ferguson, *Motif Programming Manual -- for OSF/Motif Release 1.2*, O'Reilly & Associates, Inc., 1994.

[HI92] J. A. Holey and O. H. Ibarra, "Iterative algorithms for planar convex hull on mesh-connected arrays," *Parallel Computing,* 18, 1992, pp281-296.

[HRD92] T. Hsu, V. Ramachandran and N. Dean. "Implementation of Parallel Graph Algorithms on the MasPar", *DIMACS Series in Discrete Mathematics*, American Mathematical Society 1992.

[HB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[K92] D. E. Knuth, *Axioms and Hulls*, Springer - Verlag, 1992.

[KJ78] J. Koplowitz and D. Jouppi, "A More Efficient Convex Hull Algorithm," *Inform. Process. Lett.,* Vol. 7, No. 1, Jan., 1978, pp56-57.

[KH83] M. Kumar and D. S. Hirschberg, "An efficient implementation of batcher's odd-even merge algorithm and its application in parallel sorting schemes," *IEEE Trans. Comput.* C-32, 1983, pp254-264.

[L83] D. T. Lee, "On Finding the Convex Hull of a Simple Polygon", *International Journal of Computer and Information Sciences*, Vol. 12, No. 2, 1983, pp87-98.

[L92] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,"* Morgan Kaufmann Publishers, 1992.

[L93] Y. C. Lin, "On Balancing Sorting on a Linear Array", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 5, May 1993, pp566-571.

[LR93] G. Lerman and L. Rudolph, *Parallel Evolution of Parallel Processors*, Plenum Press, 1993.

[LZ92] J. Liu and S. Q. Zheng, "A Simplified Optimal Algorithm for Constructing the Convex Hull of a Simple Polygon", *Proceedings of the 30th Annual ACM Conference(Southeast Region)*, March, 1992, pp144-148.

[M57] E. A. Maxwell, *The methods of plane projective geometry based on the use of general homogeneous coordinates,* Cambridge University Press, 1957.

[M91] The MPL Reference Manual, MasPar Corporation, 1991.

[MA79] D. McCallum and D. Avis, "A Linear Algorithm for Finding the Convex Hull of a Simple Polygon", *Inform. Process. Lett.,* Vol. 9, 1979, pp201-206.

[MM89] D. S. Moore and G. P. McCabe, *Introduction to the Practice of Statistics*, W. H. Freeman and Company, 1989.

[MS89] R. Miller and Q. F. Stout, "Mesh computer algorithms for computational geometry," *IEEE Transactions on Computers,* Vol. C-38, No 3, March 1989, pp321-340.

[NMB80] D. Nath, S. N. Maheshwari and P. C. P. Bhatt, "Parallel Algorithms for the Convex Hull Problem in Two Dimensions," *Technical Report EE 8005,* Department of Electrical Engineering, Indian Institute of Technology, Delhi Hauz Khas, New Delhi 110016, India, October 1980.

[NS79] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.* C-27(1), 1979, pp2-7.

[OL81] [8] M.H. Overmars and J. van Leeuwen, "Maintenance of Configurations in the Plane", *Journal of Computer and System Sciences,* 23, 1981, pp166-204.

[OSZ93] S. Olariu, J. L. Schwing, and J. Zhang, "Optimal Convex Hull Algorithms on Enhanced Meshes", *BIT*, 33, 1993, pp396-410.

[PB90] A. Park and K. Balasubramanian, "Reducing communication costs for sorting on mesh-connected and linearly connected parallel computers," *Journal of Parallel and Distributed Computing*, Vol. 9, 1990, pp318-322.

[P91] *Introduction to Parallel Computing Laboratory*, Dept. of Physics and Astronomy, LSU, Oct., 1991.

[PH77] F. P. Preparata and S. J. Hong, "Convex Hull of Finite Sets of Points in Two and Three Dimensions," *Communication of the ACM*, Vol. 20, No. 2, Feb. 1977, pp87-93

[PS88] F. P. Preparata and M. A. Shamos, *Computational Geometry: An Introduction*, Springer Verlag, (corrected and second printing), 1988.

[Q87] M. J. Quinn, *Designing Efficient Algorithms For Parallel Computers*, McGraw-Hill, International Edition, 1987.

[S93] J. R. Smith, *The Design and Analysis of Parallel Algorithms*, Oxford University Press, 1993.

[S88a] I. Stojmenovic, "Computational geometry on a hypercube," *Proceedings of the 1988 International Conference on Parallel Processing,* St. Charles, Illinois, Vol. III, 1988, pp100-103.

[S88b] Q. F. Stout, "Constant-time geometry on PRAMS," *Proceedings of the 1988 International Conference on Parallel Processing,* St. Charles, Illinois, Vol. III, 1988, pp104-107.

[S78] M. Shamos, "Problems in Computational Geometry", Ph.D. dissertation, Computer Science Department, Yale University, 1978.

[S72] J. Sklansky, "Measure Concavity on a Rectilinear Mosaic", *IEEE Trans. Comput.,* Vol. 21, 1972, 1355-1364.

[T93] M. Teillaud, *Towards Dynamic Randomized Algorithms in Computational Geometry,* Springer-Verlag, 1993.

[TK77] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Comm. ACM* 20(4), 1977, pp263-271.

[Y81] A. C. Yao, "A lower bound to finding convex hulls," *Journal of the ACM,* Vol. 28, No. 4, 1981, pp780-787.

Vita

Mr. Jigang Liu graduated from Beihang high school, Beijing, China in 1974. He went to the countryside and worked in the field for two years. From 1976 to 1978, he worked with the Central Broadcasting Services of China as an electrician and a construction coordinator. After four year study in Beijing University of Aeronautics and Astronautics, he received a B.S. degree in computer science in 1982. He then worked with the headquarters of the China Aviation Industry Corporation as a computer programmer, a system analyst and a project manager. In 1990, he joined the Ph.D. program in computer science at Louisiana State University. He has been a graduate teaching assistant and the system manager of the software engineering laboratory since that time.

Mr. Liu's current research interests are parallel and distributed computing, computational geometry, software engineering and database systems. He is a student member of the ACM and a member of the IEEE Computer Society.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Jigang Liu

Major Field: Computer Science

Title of Dissertation: Parallel Algorithms for Constructing Convex Hulls

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

7/6/95