1992

# Parallel Geometric Algorithms.

Fenglien Lee
*Louisiana State University and Agricultural & Mechanical College*

Order Number 9302909

# Parallel geometric algorithms

Lee, Fenglien, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# PARALLEL GEOMETRIC ALGORITHMS

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
Fenglien Lee
B.Ed., National Taiwan Normal University, 1980
M.S., Indiana University - Bloomington, 1984
August 1992

**In memory of my beloved father**

**Teh-Ming Lee (Oct. 22, 1911 - Aug. 6, 1991)**

## Acknowledgements

I express my heartfelt appreciation to my advisor, Professor Si-Qing Zheng, for his help, advice, guidance and encouragement throughout the course of my study and research at LSU.

I greatly appreciate my advisory committee members, Prof. S. Sitharama Iyengar, Prof. J. Bush Jones, Prof. John M. Tyler, Prof. Charles A. Harlow, and Prof. James G. Oxley, for their precious advices and suggestions about my research.

I also express my gratitude to Prof. Peter P. Chen, Prof. Donald L. Greenwell, and Prof. Jianhua Chen for their guidance and help during the period of this work.

I would like to thank my good friends Jin-Jwei Chen, Hla Min, Maung Maung Htay, Weian Deng, Chih-Ping Chu, Kyung Hee Kwon, Sangbum Lee, Ching-Liang Tseng and Jigang Liu for their help, encouragement and concern during the past years.

Special thanks go to my mother, Wei Yang Lee, my wife, Shujen Chen, and my two sons, Chung-Hau Lee and Chih-Hsuen Lee, for their constant love, sacrifice, and support throughout my life.

# Table of Contents

# List of Tables

# List of Figures

# Abstract

Geometric algorithms have many important applications in science and technology. Some geometric problems require fast response time that could not be achieved by traditional sequential algorithms. However, the speed, power and versatility of parallel computers can be exploited to develop efficient geometric algorithms as shown in this dissertation. Our study focuses on designing efficient parallel geometric algorithms and analyzing their computational complexities.

In this research, first we developed a parallel algorithm to find the *maxima* of a set of $N$ points in the d-dimensional space, d > 3, on a hypercube SIMD machine. Our algorithm is a parallel implementation from the sequential algorithm given by Kung, Luccio, and Preparata [KLP75]. Although the time complexity, $O(N^{0.77} \log^{d-1} N)$, of our algorithm is not optimal, it is the first *sublinear time* algorithm for solving the high dimensional maxima problem.

Next, we developed another parallel algorithm to construct the *Voronoi diagram* of a point set in the plane. Our algorithm is based on the sequential algorithm given by Brown [B79]. We use an $N \times N$ mesh of trees (MOT) SIMD computer and get the optimal time complexity $O(log^2 N)$.

Finally, we developed another MOT algorithm to solve the *congruent pattern* problem. Given a simple polygon P with $k$ edges and a planar graph G with $N$ edges, $N > k$. The problem is to find all the patterns (cycles) in G which are congruent to P. Our algorithm is based on the CREW PRAM algorithm given by Jeong, Kim, and Baek [JKB92]. We also use an $N \times N$ MOT and get the optimal time complexity $O(k \log N)$.

# Chapter 1

# Introduction

Geometry is a major branch of mathematics with important theory and applications. The treatises of ancient mathematicians often form the basis of efficient algorithms on geometry. Although algorithmic studies have appeared in the past century in the scientific literature and with an increasing intensity in the past two decades, it was only recently that systematic studies of geometric algorithms have been undertaken.

The field of geometric algorithms is interesting to study not only because of rich historical context but also because many new fundamental algorithms are being developed for important large-scale applications. *Computational Geometry*, as pioneered by Michael Ian Shamos [S75], is a young and fertile discipline dedicated to the design and analysis of algorithms for inherently geometric problems. A large number of applications such as pattern recognition, image processing, computer graphics, computer-aided design, VLSI circuit design, robotics, database search, operations research, and statistics have benefited from this discipline.

Geometric problems are usually easy to figure out, but some are relatively hard to solve on a computer. Many problems that can be solved easily by a person require non-trivial computer algorithms. For example, even a child can identify at a glance whether or not a given point lies inside a given polygon on a piece of paper. However, we need a special solving strategy and appropriate data structures to solve this problem on a computer.

Most of inherently geometric problems for which efficient algorithms have to be developed use both sequential and parallel processing techniques. The study of parallel geometric algorithms has recently begun in earnest, although the first major work was done by Anita Chow in 1980 [C80]. There are at least three reasons why parallel algorithms for geometric problems have become our special interest.

(1) Geometric algorithms are often needed in critical on-line systems which require short response time. Even if optimal sequential algorithms exist for such problems, they do not ensure acceptable (or tolerable) response time for the large amount of data, such as image data, to be processed.

(2) Efficient sequential algorithms do not exist for a large class of important high dimensional geometric problems.

(3) It is feasible to design and implement efficient parallel geometric algorithms on an increasing number of commercial parallel machines [DS88].

## 1.1 Basic Geometric Problems

According to the nature of the geometric objects involved, such as points, lines, and polygons, some basic geometric problems can be categorized as follows.

(1) *Convex Hull*

The convex hull of a point set S in d-dimensional Euclidean space is the boundary of the smallest convex domain which contains S [PS88] (see Figure 1.1). Convex hull problems consist of finding the convex hull of a two-dimensional point set, the convex polygon of a simple polygon, the convex polyhedron of a set of three-dimensional point set, and the maxima of a d-dimensional point set ,etc..

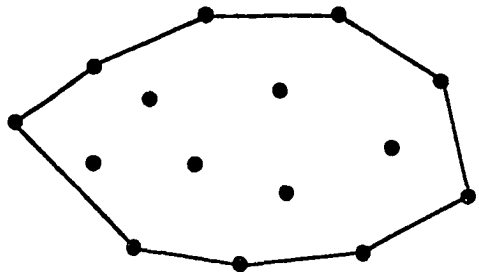

Fig. 1.1 The convex hull of a two-dimensional point set.

(2) *Intersections*

This problem is to detect the intersections of geometric objects, such as intersections of a set of line segments, intersections of a set of polygons, intersections of a set of polyhedra, and intersections of a set of iso-oriented rectangles.

(3) *Proximity*

(a) Closest pair : Given a set of points in the plane, find two points whose mutual distance is smallest.

(b) All nearest neighbors : Given a set of points in the plane, find a nearest neighbor for each point.

(c) Minimum distance : Given two sets $S_1$ and $S_2$ of objects, such as arbitrary or convex point sets, find the minimum distance $d(s_i, s_j)$, where the pair $(s_i, s_j) \subset S_1 \times S_2$.

(d) Voronoi diagram : Given a set of points S in the plane, construct the Voronoi diagram of S. Constructing Voronoi diagram is an extension of proximity problems.

(4) *Visibility Problems*

(a) Visibility polygon from an edge : Given a simple polygon $P$ and an edge $e$ of $P$, find the portion of the boundary of $P$ that is visible from $e$. A point $q$ in $P$ is visible from the edge e if there exists a point $r$ on $e$ such that the line segment of $q$ and $r$ does not intersect the boundary of $P$.

(b) Visibility polygon from a point : Given a set of disjoint polygons in the plane, find the portion of the boundary of these polygons which is visible from a given view-point.

(c) Visibility of a polygon : For a polygon with or without holes, determine the portion of the polygon which is visible from a given view-point.

(d) Visibility of segments : For a set of line segments in the plane or three-dimensional space, determine which segments are visible from a given point.

(5) *Decomposition Problems*

The problem is to decompose an object into more meaningful components. A common task is to decompose an object into convex parts. Decompositions are obtained by adding segments inside the object.

(a) Triangulation of a Point Set : This problem is to find a triangulation of the input set in two or higher dimensions (see Figure 1.2).

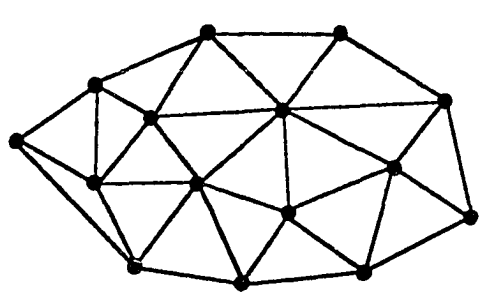

Fig. 1.2 Triangulation of a point set

(b) Triangulation of a Line Segment Set : This problem is to find a triangulation of a set of input line segments in two or higher dimension.

(c) Trapezoidal Partitioning : This problem is to find a partitioning of the input polygon into trapezoids.

## 1.2 Computation Models for Parallel Geometric Algorithms

SIMD computers are the major computing models for designing parallel geometric algorithms. In this class, all processors are supervised by the same control unit and receive the same instruction broadcast from the control unit, but they operate on different data sets from distinct data streams. According to the communication approaches, SIMD computers can be divided into the following two subclasses.

(1) *Shared-Memory (SM) SIMD Computers*

This subclass is also known as the Parallel Random Access Machine (PRAM). As the name implies, processors share a common memory. When two processors wish to communicate, they do so through the shared memory (see Figure 1.3). According to whether two or more processors can gain access to the same memory location simultaneously, the class of SM SIMD computers can be further divided into CREW (Concurrent-Read, Exclusive-Write), EREW, CRCW, and ERCW four models.

(2) *Interconnection-Network SIMD Computers*

The major disadvantage of SM PRAM is that it is not easily scalable. Limited system bus capacity and memory access collisions prohibit the use of a very high degree of parallelism. In the interconnection-network models each processor possesses a local memory and messages are passed over an interconnection network. They do

not suffer from the unscalability problem. The communication bandwidth and the number of communication links will increase when the number of processors increases. It is generally agreed upon that significant orders of speed improvement can only come from this subclass of SIMD computers [VDP89]. The following are popular networks for designing parallel geometric algorithms (refer Figure 1.4).

(a) Linear Array : This is the simplest structure of interconnection networks. Each processor has its own memory. Processor $P_i$ is linked to its two neighbors $P_{i-1}$ and $P_{i+1}$ through a two-way communication line. Each of the end processors, namely, $P_1$ and $P_N$, has only one neighbor.

(b) Mesh : In a mesh network, the nodes are arranged into a two-dimensional lattice. Computation is allowed only between neighboring nodes; hence interior nodes communicate with four other processors. Some variants of the mesh model allow wrap-around connections between processors on the edge of the mesh. These connections may connect processors in the same row or column or they may be toroidal.

(c) Tree : In this network, processors form a complete binary tree. Such a tree has d levels, numbered from 0 to d-1, and N = $2^d$-1 nodes each of which is a processor. The root is processor at level d-1 and all the leaves are at level 0.

(d) Mesh of Trees : A mesh of trees of size $m$, where $m$ is a power of 2, is defined as follows : Begin with $m^2$ nodes arranged in a square grid, but without any connecting edges. For each row of nodes, say $v_1$, ..., $v_m$, in order from left, build a complete binary tree with these nodes as leaves, adding the interior nodes, which are not among the original $m^2$ nodes. Do the same for each column of the grid [U84].

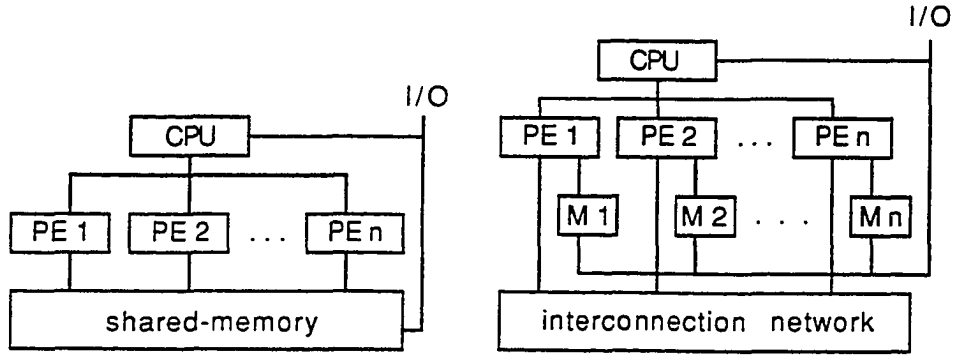


Fig. 1.3 Shared-memory and interconnection network SIMD models.

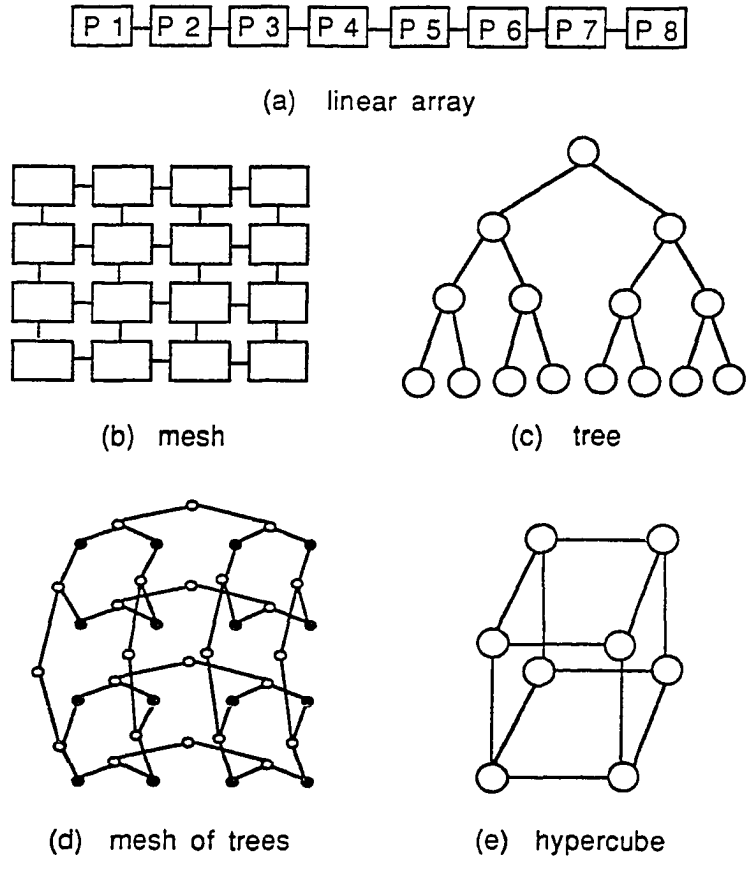


Fig. 1.4 Some popular SIMD interconnection networks.

(e) Hypercube : The hypercube structure is generally defined in a recursive manner. A hypercube of dimension 0 consists of one node. A hypercube of dimension d consists of two hypercubes of dimension d-1 whose corresponding nodes are connected. The nodes of a d-dimensional hypercube can be numbered by d-bit binary numbers in such a way that there is a physical connection between two nodes if and only if their binary representations differ by exactly one bit.

## 1.3 Outline of the Dissertation

In chapter 2, we present a parallel algorithm on a hypercube machine to solve the d-dimensional maxima problem, for $d > 3$. This is the first parallel algorithm for the d-dimensional maxima problem, and also the first *sublinear time* algorithm by using SIMD interconnection-network models.

In chapter 3, we develop an optimal parallel algorithm to construct the Voronoi diagram of a two-dimensional point set on a mesh of trees. This is the first algorithm developed on a mesh of trees and is the fastest algorithm for constructing Voronoi diagram by using SIMD interconnection-network models.

In chapter 4, we design a parallel algorithm on a mesh of trees to find all the patterns in a planar graph which are congruent to a given test pattern. This is the first and fastest algorithm for this problem by using SIMD interconnection-network models.

Finally, we conclude our research and discuss our future work in chapter 5. The three parallel geometric algorithms in chapter 2, 3, and 4 are summarized as follows.

(1) *The Maxima Problem*

Let $S$ be a set of $N$ points in $d$-dimensional space. For a point $p \in S$, we use $x_k(p)$ to denote the $k$th-dimensional coordinate value of $p$. For two distinct points $p_i$ and $p_j$ in S, we say that $p_i$ dominates $p_j$ if $x_k(p_i) \geq x_k(p_j)$, for $k = 1, 2, \ldots, d$. A point $p$ in $S$ is called a *maximum* of $S$ if there does not exist any other point $q$ of $S$ such that $q$ dominates $p$. The maxima problem consists of finding all the maximums of $S$. This problem has many important applications in statistics, economics, operations research and computational geometry.

For this problem, we present a parallel algorithm on a hypercube machine. For a set of $N$ points in $d$-dimensional space, our algorithm runs in $O(N^{0.77} \log^{d-1} N)$ time, assuming that there are $N$ processors in the hypercube machine. If the size $N$ of input point set is greater than $O(P)$, where $P$ is the number of processors in the hypercube, our algorithm can be generalized to solve the maxima problem in $O((N/P^{0.23} \log^{d-2} P(\log(N/P) + \log P))$ time.

(2) *The Voronoi Diagram*

The Voronoi diagram of a set $S$ of $N$ points in the plane is a polygonal subdivision of $N$ regions in the plane, each containing one point of $S$, such that the region for each point $p$ of $S$ is the locus of points in the plane which are closer to $p$ than to other points of $S$. The Voronoi diagram is an extremely useful data structure with many important applications in robotics, image processing, graph theory, and computational geometry [PS88, S78].

For this problem, we present an $O(log^2 N)$-time parallel algorithm on an $N \times N$ mesh of trees. Our algorithm transforms points of $S$ into a set $S'$ of points on a sphere in $E^3$, then uses the divide-and-conquer approach to construct the three-dimensional

convex hull $CH(S')$ of $S'$, and finally transforms $CH(S')$ into the Voronoi diagram of $S$. Our convex hull algorithm can be used to construct the convex hull of any set of points in $E^3$ in $O(log^2 N)$ time. Our algorithms are the fastest $NC$ algorithms ever known for constructing planar Voronoi diagrams and three-dimensional convex hulls.

(3) *The Congruent Patterns*

Two planar patterns $P_1$ and $P_2$ are congruent if their corresponding boundary edges (e.g. edge $e_i$ in $P_1$ and edge $e_j$ in $P_2$) are of equal length and the corresponding interior angles (e.g. interior angle $\theta_i$ in $P_1$ and interior angle $\theta_j$ in $P_2$) between any pair of adjacent boundary edges are also equal. Finding congruent patterns arises frequently in computer-aided design (CAD), image analysis, pattern recognition, and computer vision. Given a test pattern P which is a simple polygon with $k$ edges, and a planar straight-line graph (PSLG) G with $n$ edges, where $k \leq n$, the *congruent pattern problem* is to locate all the cycles in G which are congruent to P.

For this problem, we present a parallel algorithm on an $N \times N$ mesh of trees. Our algorithm takes the optimal time complexity $O(k \log N)$, where $N = n$.

# Chapter 2

# The Maxima Problem

## 2.1 Introduction

Let $S$ be a set of $N$ points in $d$-dimensional space. For a point $p \in S$, we use $x_k(p)$ to denote the $k$-dimensional coordinate value of $p$. For two distinct points $p_i$ and $p_j$ in S, we say that $p_i$ dominates $p_j$ if $x_k(p_i) \geq x_k(p_j)$, $k = 1, 2, \ldots, d$. A point $p$ in $S$ is called a *maximum* of $S$ if there does not exist any other point $q$ of $S$ such that $q$ dominates $p$. The maxima problem consists of finding each maximum of $S$. This problem has many important applications in statistics, economics, operations research and computational geometry. An example of 2-dimensional maxima is shown in Figure 2.1.

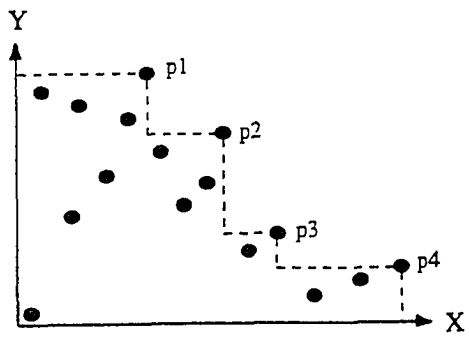


Fig. 2.1 The set of maxima = {p1, p2, p3, p4}.

In [KLP75], Kung, Luccio and Preparata showed that the sequential time complexity of the maxima problem is $O(N \log N)$ for $d \leq 3$. Their remarkable divide-and-conquer method solved the $d$-dimensional maxima problem sequentially in $O(N \log^{d-2} N + N \log N)$ time. It still remains an open problem whether or not this sequential time complexity can be improved.

In the past few years, several attempts were made for developing efficient parallel algorithms for the maxima problem ([ACG87] [AG86] [D86] [RS87] [S88b]). The algorithms given in [ACG87] and [AG86] solve the 2-dimensional maxima problem on CREW PRAM in $O(\log N)$ time using $O(N)$ processors. Furthermore, the algorithm in [ACG87], which is a clever implementation of the sequential algorithm given in [KLP75], solves the d-dimensional maxima problem on CREW PRAM in $O(\log^{d-2} N)$ time using $O(N)$ processors. Currently, this is the best parallel algorithm for the maxima problem on PRAM; any further improvement of this algorithm will imply the existence of better sequential algorithms. The PRAM models are based on abstract data communications. In reality, all multiprocessor systems are based on sparse interconnection networks such as trees, rings, meshes, shuffle-exchange networks, and hypercubes. Hence, it is important to investigate the parallel maxima algorithms for these realistic machine models.

In [D86], Dehne presented a parallel maxima algorithm for an $N^{1/2} \times N^{1/2}$ mesh-connected computer. He showed that, for $|S| = N$, this algorithm runs in $O(N^{1/2})$ time when $d \leq 3$, and $O(N^{0.5+\log(d-2)})$ time when $d > 3$. Clearly, this algorithm is optimal when $d \leq 3$, but it can be extremely bad when $d > 3$. For example, if $d = 4$, the time complexity of this algorithm is $O(N^{1.5})$, which is much worse than the best sequential algorithm. In [S88b], a parallel algorithm for solving 2-dimensional maxima problems on a hypercube machine is given. This algorithm runs in $O(\log^2 N)$ time using $N$ processors for $N$ points. To our knowledge, no sublinear-time parallel algorithm for the maxima problem of dimension greater than 3 has been developed for parallel systems with sparse interconnection networks.

In this chapter, we present a parallel hypercube implementation of the sequential algorithm given in [KLP75]. We show that for a set of $N$ points in $d$-dimensional space, our algorithm runs in $O(N^{0.77}\log^{d-1} N)$ time, on a hypercube with $O(N)$ processors. If the size $N$ of input point set exceeds $O(P)$, where $P$ is the number of processors in the hypercube machine, our algorithm can be generalized to solve the maxima problem in $O((N/P^{0.23})\log^{d-2} P(log(N/P) + \log P))$ time.

## 2.2 Preliminaries

In this section, we introduce the data structures in each processor of hypercube , the architecture of hypercube machine, and the operations of data communications inside hypercube.

### 2.2.1 Data Structures

For the correctness and efficiency of our algorithm, we implement a stack $STK_i$ in the private memory associated with processor $PE_i$. A stack entry consists of two fields, *point* , which is used to keep the $d$ coordinate values $x_1(p_k), \ldots, x_d(p_k)$ of some point $p_k$, and *flag*, which is used to indicate whether or not $p_k$ has been detected as being dominated by some other point(s) in the input point set $S$ of size $N = 2^n$, where $n$ is the dimension of the hypercube, during the computing process. These two fields of the $j$-th entry of $STK_i$, where $j \geq 0$, are denoted as $STK_i[j].\ point$ and $STK_i[j].\ flag$. The 0-th entry of $STK_i$ is called the bottom entry of the stack.

If $STK_i[j].\ point$ contains point $p_k$, then $STK_i[j].\ flag = 1$ indicates that point $p_k$ is not a maximum of $S$. For each stack $STK_i$, we maintain a stack pointer $SP_i$, which points to the *current top entry* of $STK_i$. Two stack operations $push[p, f; STK_i]$ and

*pop*[$STK_i$] are defined on $STK_i$. The *push* operation increments $SP_i$ by 1, and store $p$ and $f$ into $STK_i[SP_i]$. *point* and $STK_i[SP_i]$. *flag*, respectively. The *pop* operation returns pair ($p$, $f$), which is ($STK_i[SP_i]$.*point*, $STK_i[SP_i]$.*flag*), and then decrements $SP_i$ by 1.

### 2.2.2 Hypercube Machine and Data Communications

An $n$-dimensional hypercube $Q_n$ is a graph of $P = 2^n$ vertices, each labeled by a unique $n$-bit binary number, such that two vertices are connected by an edge if and only if their binary labels differ in one bit position. We define a $k$-dimensional subcube $S_k$ of $Q_n$ as a subgraph induced by vertices with $(n-k)$-bit prefixes of their binary labels equal to a given $(n-k)$-bit binary constant. Clearly, by ignoring the $n-k$ leftmost bits of the vertex labels, $S_k$ is a $k$-dimensional hypercube. A $k$-dimensional subcube induced by vertex set $\{b, b+1, \ldots, b+2^k-1\}$ is denoted as $S_k(b)$.

We say that two $k$-dimensional subcubes $S_k(b_1)$ and $S_k(b_2)$ of $Q_n$ are adjacent if there are two adjacent vertices $u$ and $v$ (i.e. $u$ and $v$ are connected by an edge in $Q_n$) such that $u$ is in $S_k(b_1)$ and $v$ is in $S_k(b_2)$. By the definition of $Q_n$, if $S_k(b_1)$ and $S_k(b_2)$ are adjacent, then the vertices in $S_k(b_1)$ and $S_k(b_2)$ are pairwisely adjacent in the following way : vertex $u$ in $S_k(b_1)$ and vertex $v$ in $S_k(b_2)$ are adjacent if the $k$-bit postfixes of their binary labels are identical. For two adjacent $k$-dimensional subcubes $S_k(b_1)$ and $S_k(b_2)$ of $Q_n$, we use ($S_k(b_1)$, $S_k(b_2)$) to denote the subgraph of $Q_n$ induced by vertices of $S_k(b_1)$ and $S_k$ ($b_2$). Since the $(n-k)$-bit prefix of vertices of $S_k(b_1)$ and the $(n-k)$-bit prefix of vertices of $S_k(b_2)$ are distinct in one bit position, ($S_k(b_1)$, $S_k(b_2)$) can be treated as a $(k+1)$-dimensional hypercube by replacing one of these two

prefixes with 0 and the other with 1. This simple fact will allow us to partition a hypercube into subcubes in a more general sense. A 3-dimensional hypercube and its 2 subcubes are shown in Figure 2.2.

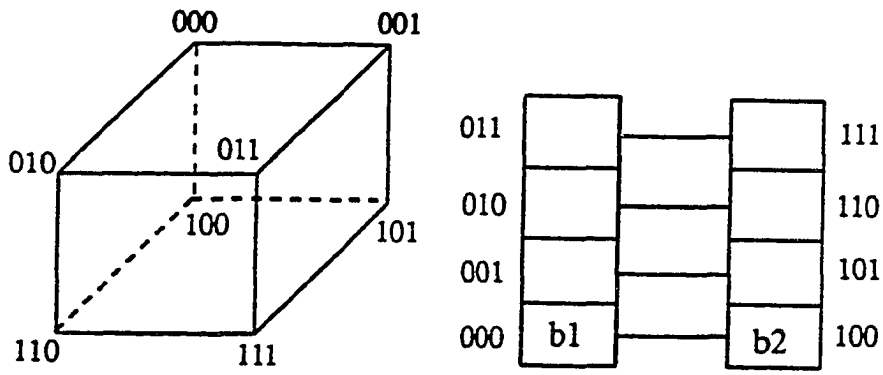


Fig. 2.2 A 3-d hypercube and its two subcubes with base addresses $b_1$ and $b_2$.

A hypercube machine is a multiprocessor system in which processors and data links correspond to vertices and edges of a hypercube graph, respectively. For convenience, we also use $Q_n$ to denote a hypercube machine with $P = 2^n$ processors. Each processor $PE_i$, where $i$ is an $n$-bit binary label, is associated with a private (local) memory. Due to their regularity, small diameters, low connection degrees, and many additional attractive combinatorial and topological properties, hypercube machines have become one of the most important architectures for parallel computing.

Before we discuss our algorithm, we introduce two pairs of useful parallel operations *BROADCAST / CENSUS*, and *SORT / UNSORT*, The parameters for *BROADCAST* and *CENSUS* are $b_1$, $m_1$, $b_2$, $m_2$, $b_3$ and $m_3$, where $b_1 \geq b_3$, and $b_2 \geq b_3$, $m_1 < m_3$, and $m_2 < m_3$. Here, each pair of $b_i$ and $m_i$ defines the subcube $S_{m_i}(b_i)$ of $Q_n$, and both $S_{m_1}(b_1)$ and $S_{m_2}(b_2)$ are subcubes of $S_{m_3}(b_3)$.

(1) *BROADCAST*($b_1$, $m_1$, $b_2$, $m_2$, $b_3$, $m_3$)

This operation is a generalized parallel data broadcasting operation on a hypercube in the following sense: it broadcasts the pair ($p$, $f$) of the current top stack entry of $PE_{b_1+j}$, to the $j$-th processor in each of the $m_1$-dimensional subcubes of ($S_{m_3}(b_3)$ - $S_{m_2}(b_2)$) - $S_{m_1}(b_1)$, then each of the processors in ($S_{m_3}(b_3)$ - $S_{m_2}(b_2)$) - $S_{m_1}(b_1)$ performs a *push* stack operation on the pair it received, see Figure 2.3. By treating each $m_1$-dimensional subcube as a "super vertex", *BROADCAST* can be carried out in $O(m_3 - m_1)$ time using conventional hypercube data broadcasting algorithm.

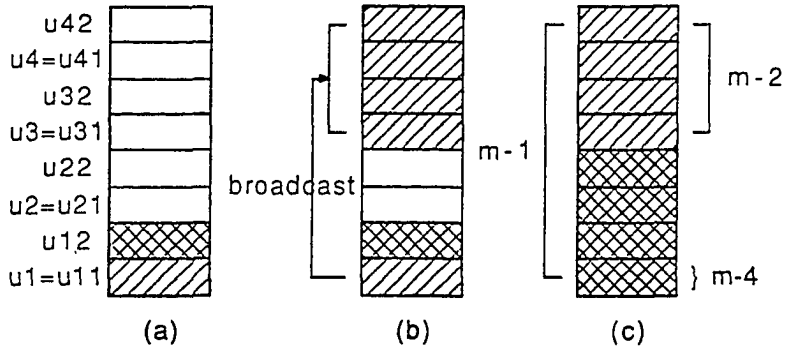


Fig. 2.3 Broadcast data in the 2 subcubes in (a) with base addresses $u_{11}$ and $u_{12}$ to other 6 subcubes.
(b) After BROADCAST($u_{11}$, $m-4$, $u_1$, $m-2$, $u_1$, $m-1$).
(c) After BROADCAST($u_{12}$, $m-4$, $u_3$, $m-2$, $u_1$, $m-1$).

(2) *CENSUS*($b_1$, $m_1$, $b_2$, $m_2$, $b_3$, $m_3$)

The *flag* fields of the current top stack entries of the $j$-th processors of all $m_1$-dimensional subcubes of ($S_{m_3}(b_3)$ - $S_{m_2}(b_2)$) - $S_{m_1}(b_1)$ are logically *OR*ed, and the obtained boolean value replaces the $STK_{b_1+j}[SP_{b_1+j}]$. *flag*. Furthermore, each of the processors in ($S_{m_3}(b_3)$ - $S_{m_2}(b_2)$) - $S_{m_1}(b_1)$ performs a *pop* stack operation to discard the *OR*ed top entry. Considering the data communication pattern

involved, *CENSUS* is a reverse operation of *BROADCAST*, and it also takes $O(m_3 - m_1)$ time.

(3) *SORT*$(b, m, k)$

Each processor $PE_i$ in $S_m(b)$ performs $(p^i, f^i) := pop(STK_i)$. Then, all the pairs of $(p^i, f^i)$, one in each processor, are sorted in parallel on their $x_k$ coordinate values. Let $(p^{j_i}, f^{j_i})$ be the pair in $PE_i$ after sorting, then $x_k(p^{j_i}) < x_k(p^{j_{i+1}})$. Once this is done, all processors $PE_i$ in $S_m(b)$ perform $push[p^{j_i}, f^{j_i}; STK_i]$ in parallel. Using bitonic hypercube sorting algorithms, *SORT* takes $O(m^2)$ time.

(4) *UNSORT*$(b, m, k)$

This is the reverse operation of the latest *SORT* in the recursive process. Each processor $PE_i$ in $S_m$(b) performs $(p, f) := pop(STK_i)$. Clearly, $(p, f) = (p^{j_i}, f^{j_i})$, where $(p^{j_i}, f^{j_i})$ is the pair received by $PE_i$ in the corresponding *SORT* operation. Then, each $PE_i$ in $S_m$(b) sends $(p^{j_i}, f^{j_i})$ back to $PE_{j_i}$, where the pair was stored before the corresponding *SORT* operation. Finally, all processors in $S_m(b)$ perform *push* operation on the pair it received. This can be done in $O(m^2)$ time if data communication pattern of the corresponding *SORT* has been properly recorded.

The operations *BROADCAST* and *CENSUS*, and operations *SORT* and *UNSORT* are always performed in pairs on the current top stack entries. Proper pairing of *BROADCAST / CENSUS* and *SORT / UNSORT* operations is important for the correctness of algorithms *FILTER_S* and *FILTER_G*. They ensure the concerned points at different recursion levels to be put in the current top stack entries of all involved

processors, and their dominance status to be recorded in parallel. For this reason, we introduce a well-formed sequence of the four operations. We use "<" and ">" to represent *BROADCAST* and *CENSUS*, respectively. When *BROADCAST* or *CENSUS* is invoked, $m_3 = m_2 + 1$ must hold. We say that a processor $PE_i$ is involved in a *BROADCAST* (*CENSUS*) operation if $PE_i$ pushs (pops) an entry into (from) its stack during the operation. Clearly, the processors involved in a pair of *BROADCAST*($b_1$, $m_1$, $b_2$, $m_2$, $b_3$, $m_3$) / *CENSUS*($b_1$, $m_1$, $b_2$, $m_2$, $b_3$, $m_3$) are those in $(S_{m_3}(b_3)\text{-}S_{m_2}(b_2))$ - $S_{m_1}(b_1)$. Similarly, we use "(" and ")" to represent *SORT* and *UNSORT*, respectively. We say that a processor $PE_i$ is involved in a pair of *SORT*($b$, $m$, $k$) / *UNSORT*($b$, $m$, $k$) operations if $PE_i$ is in $S_m(b)$.

For any processor $PE_i$, if we list all *BROADCAST*, *CENSUS*, *SORT* and *UNSORT* operations $PE_i$ involved from left to right as time passes during the computation, we get a sequence of mixed "<", ">", "(" and ")", which is called a *BCSU sequence*. We define *well-formed BCSU sequence* as follows : (a) an empty sequence is a *BCSU* sequence; (b) both $<T_1>$ and $(T_1)$ are well-formed *BCSU* sequences, if $T_1$ is a well-formed *BCSU* sequence, and "<" and ">" ("(" and ")", respectively) correspond to a *BROADCAST* / *CENSUS* (*SORT* / *UNSORT*, respectively) pair with the same parameter values ; (c) $T_1T_2$ is a well-formed *BCSU* sequence, if $T_1$ and $T_2$ are well-formed sequences; and (d) any sequence that can not be generated by (a), (b) and (c) is not a well-formed *BCSU* sequence. We will enforce the *BCSU* sequence for each processor to be well-formed. The symbolic notation of *BCSU* sequences for *BROADCAST* / *CENSUS* and *SORT* / *UNSORT* will provide us a very simple way of analyzing the correctness of our algorithm.

## 2.3 Parallel Maxima Algorithm

Our parallel algorithm for the maxima problem is based on an efficient and elegant sequential algorithm given in [KLP75]. A concise description of this algorithm appears in [PS88]. The divide-and-conquer algorithm employs double-recursion strategy, one on the problem size and the other on the dimension of points. The unique technique used in [KLP75] lead to a complexity that has not been surpassed for many years.

### 2.3.1 The Main Algorithm

The framework of our algorithm is as follows. Initially, points in $S$ are distributed in $N$ processors in the following way: (1) each $PE_i$ has one point in its 0-th stack entry; (2) the input points are sorted on the $x_d$ coordinate values such that for all $PE_i$ in $S_{n-1}(b_1)$, the $x_d$ coordinate value of $STK_i[0].\ point$ is less than the $x_d$ coordinate value of any $STK_j[0].\ point$ in $PE_j$, where $PE_j$ is in $S_{n-1}(b_2)$; (3) if $d \geq 3$, all points in $S_{n-1}(b_1)$ are sorted on their $x_{d-1}$ coordinate values in such a way that $x_{d-1}(STK_i[0].\ point) < x_{d-1}(STK_j[0].\ point)$ if and only if $i < j$ for $PE_i$ and $PE_j$ in $S_{n-1}(b_1)$, and all points in $S_{n-1}(b_2)$ are sorted on their $x_{d-1}$ coordinate values in the same way as in $S_{n-1}(b_1)$; and (4) $STK_i[0].\ flag$ is set to 0, a maximum, for all $PE_i$ in $Q_n$. Using hypercube bitonic sorting algorithm for preprocessing, these initial conditions can be enforced in $O(\log^2 N)$ time. Since the preprocessing time is insignificant compared with the time required for the rest of the algorithm, we will ignore it in the time analysis.

**Algorithm** *MAXIMA*($b$, $m$, $d$);

(* $b$ : base address of subcube $m$ ($m$-dimensional subcube) *)

(* $d$ : $d$-dimensional point set *)

(* initial value : $b = 0$ and $m = n$ (hypercube dimension) *)

*Input* : a set $S$ of $n$ points in $d$-dimensional space;

*Output* : the set of all maximums in $S$;

**begin**

**if** $m = 1$ (* $S$ has only 2 points *)

**then if** $STK_b[SP_b]$. *point* is dominated by $STK_{b+1}[SP_{b+1}]$. *point*

**then** $STK_b[SP_b]$. *flag* := 1

**else begin**

**parbegin**

*MAXIMA*($b$, $m-1$, $d$);

*MAXIMA*($b+2^{m-1}$, $m-1$, $d$);

**parend**

**if** $m \leq 3$

**then** *FILTER_S*($b$, $b+2^{m-1}$, $m-1$, $d-1$)

**else** *FILTER_G*($b$, $b+2^{m-1}$, $m-1$, $d-1$)

**end;**

**end** of *MAXIMA*;

Algorithm *MAXIMA* is initially invoked with $b = 0$ and $m = n$. Once it is terminated, each processor $PE_i$ contains a point in the bottom entry of $STK_i$ such that $STK_i[0]$. *flag* is 0 if and only if $STK_i[0]$. *point* is a maximum of $S$. To accomplish this, algorithm *MAXIMA* employs a divide-and-conquer strategy as follows: it divides

the $m$-dimensional subcube $S_m(b)$ into two subcubes $S_{m-1}(b)$ and $S_{m-1}(b+2^{m-1})$, and find the maximum points of the two subsets of points associated with these two subcubes in parallel; then, by using either *FILTER_S* or *FILTER_G*, all points associated with $S_{m-1}(b)$ which are dominated by any point in $S_{m-1}(b+2^{m-1})$ are detected and marked. The correctness and efficiency of *MAXIMA* depends on how *FILTER_S* and *FILTER_G* are designed.

The parameters $b_1$, $b_2$ and $m$ of subalgorithms *FILTER_S*$(b_1, b_2, m, k)$ and *FILTER_G*$(b_1, b_2, m, k)$ specify two adjacent ($m$-1)-dimensional subcubes $S_{m-1}(b_1)$ and $S_{m-1}(b_2)$ of $Q_n$, and the combination $(S_{m-1}(b_1), S_{m-1}(b_2))$, as explained earlier, specifies an $m$-dimensional subcube $S_m$ of $Q_n$ in a general sense. During the computation, right before any *FILTER_S*$(b_1, b_2, m, k)$ or *FILTER_G*$(b_1, b_2, m, k)$ is invoked, the following *filter invariant condition* is enforced by the previous computation:

> For all $PE_i$ in $S_{m-1}(b_1)$, the $x_l$ coordinate value of $STK_i[SP_i].$ *point* is less than the $x_l$ coordinate value of any $STK_j[SP_j].$ *point*, where $PE_j$ is in $S_{m-1}(b_2)$ and $k < l \leq d$. If $k \geq 3$, the $x_k$ coordinate value of $STK_i[SP_i].$ *point* of all $PE_i$ in $S_{m-1}(b_1)$ ($S_{m-1}(b_2)$, respectively) are sorted in such a way that $x_k($ $STK_i[SP_i].$ *point* $) < x_k($ $STK_j[SP_j].$ *point* $)$ if and only if $i < j$ for two distinct $PE_i$ and $PE_j$ in $S_{m-1}(b_1)$ ($S_{m-1}(b_2)$, respectively).

Subalgorithms *FILTER_S*$(b_1, b_2, m, k)$ and *FILTER_G*$(b_1, b_2, m, k)$ are used to carry out the following operations :

> For all $PE_i$ in $S_{m-1}(b_1)$ if $STK_i[SP_i].$ *point* is dominated by any $STK_j[SP_j].$ *point* of $PE_j$ in $S_{m-1}(b_2)$ in all coordinate values on $x_1, \cdots, x_k$, then set $STK_i[SP_i].$ *flag* to 1.

### 2.3.2 The Subalgorithm FILTER_S

When $m \leq 3$, the subalgorithm *FILTER_S*($b_1$, $b_2$, $m$, $k$) is invoked to perform the following operations (see Figure 2.4).

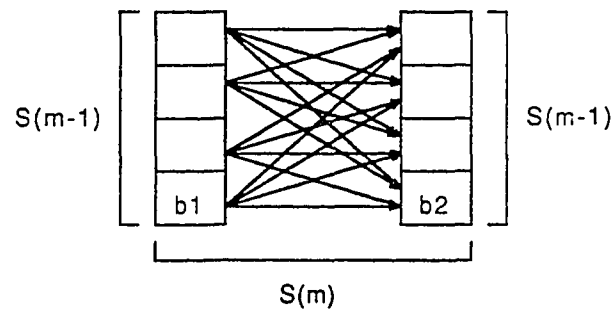


Fig. 2.4 Actions of subalgorithm *FILTER_S*.

**procedure** *FILTER_S*($b_1$, $b_2$, $m$, $k$);

**begin**

(1) Let $A_1$ and $A_2$ be the set of all entries of $S_{m-1}(b_1)$ and $S_{m-1}(b_2)$, respectively;

(2) Send $A_2$ to each PE in $S_{m-1}(b_1)$;

(3) For each point $p$ associated to an entry in $A_1$, determine whether or not $p$ is dominated by any point $q$ associated to an entry in $A_2$ on all $x_j$ coordinate values, where $1 \leq j \leq k$, if $p$ is dominated by $q$ then logically *OR*ing the *flag* field of the current top stack entry in its PE of $S_{m-1}(b_1)$ with an 1 else *OR*ing with a 0.

(4) Send each entry of $A_2$ in $S_{m-1}(b_1)$ back to its original PE in $S_{m-1}(b_2)$;

**end** of *FILTER_S*;

Since m $\leq 3$, ($S_{m-1}(b_1)$, $S_{m-1}(b_2)$) is a generalized subcube of dimension $m$ with no more than eight PEs. The time taken by *FILTER_S* is $O(2^{m+1}k) = C_1$, a constant.

### 2.3.3 The Subalgorithm FILTER_G

Subalgorithm *FILTER_G*($b_1$, $b_2$, $m$, $k$) is somewhat more complicated. This subalgorithm is a parallel implementation of the sequential algorithm in [KLP75].

**procedure** *FILTER_G*($b_1$, $b_2$, $m$, $k$);

(* $b_1$ and $b_2$ are the base addresses of the 2 subcubes of $m$ *)

(* $k$ is the # of dimension of point set *)

**begin**

(1) **if** $m \leq 3$ **then** *FILTER_S*($b_1$, $b_2$, $m$, $k$); (* only 8 points or less left *)

(2) **else if** $k = 2$ **then** *FILTER_G2*($b_1$, $b_2$, $m$); (* find 2-d maxima *)

**else begin**

(3) Decompose $(m-1)$-dimensional subcube $S_{m-1}(b_1)$ into four $(m-3)$-dimensional subcubes $S_{m-3}(u_1)$, $S_{m-3}(u_2)$, $S_{m-3}(u_3)$ and $S_{m-3}(u_4)$, where $u_1 = b_1$, $u_2 = b_1 + 2^{m-3}$, $u_3 = b_1 + 2 * 2^{m-3}$ and $u_4 = b_1 + 3 * 2^{m-3}$;

(4) Decompose each $S_{m-3}(u_i)$ into two $(m-4)$-dimensional subcubes $S_{m-4}(u_{i1})$ and $S_{m-4}(u_{i2})$, where $u_{i1} = u_i$ and $u_{i2} = u_i + 2^{m-4}$;

(5) Decompose the $(m-1)$-dimensional subcube $S_{m-1}(b_2)$ into four $(m-3)$-dimensional subcubes $S_{m-3}(v_1)$, $S_{m-3}(v_2)$, $S_{m-3}(v_3)$ and $S_{m-3}(v_4)$, where $v_1 = b_2$, $v_2 = b_2 + 2^{m-3}$, $v_3 = b_2 + 2 * 2^{m-3}$ and $v_4 = b_2 + 3 * 2^{m-3}$;

(6) Decompose each $S_{m-3}(v_i)$ into two $(m-4)$-dimensional subcubes $S_{m-4}(v_{i1})$ and $S_{m-4}(v_{i2})$, where $v_{i1} = v_i$ and $v_{i2} = v_i + 2^{m-4}$;

(* By the definition of hypercubes, $(S_{m-3}(u_i), S_{m-3}(v_i))$ is an $(m-2)$-dimensional subcube, and $(S_{m-3}(u_{ij}), S_{m-3}(v_{ij}))$ is an $(m-3)$-dimensional subcube, where $1 \leq i \leq 4$ and $1 \leq j \leq 2$; *)

(7) Let $p^i$ denote the point $STK_{u_i}[SP_{u_i}]$. *point*, and let $p^{\max}$ and $q^{mid}$

denote the points $STK_{b_1+2^{m-1}-1}[SP_{b_1+2^{m-1}-1}]$. *point*

and $STK_{b_2+2^{m-2}}[SP_{b_2+2^{m-2}}]$. *point*, respectively;

(8) Compare $x_k(q^{mid})$ with $x_k(p^{\max})$, $x_k(p^4)$, $x_k(p^3)$, $x_k(p^2)$ and $x_k(p^1)$;

*Case 1* : **if** $x_k(p^{\max}) < x_k(q^{mid})$ **then** *FILTER1*$(b_1, b_2, m, k)$;

*Case 2* : **if** $x_k(p^4) < x_k(q^{mid}) < x_k(p^{\max})$ **then** *FILTER2*$(b_1, b_2, m, k)$;

*Case 3* : **if** $x_k(p^3) < x_k(q^{mid}) < x_k(p^4)$ **then** *FILTER3*$(b_1, b_2, m, k)$;

*Case 4* : **if** $x_k(p^2) < x_k(q^{mid}) < x_k(p^3)$ **then** *FILTER4*$(b_1, b_2, m, k)$;

*Case 5* : **if** $x_k(p^1) < x_k(q^{mid}) < x_k(p^2)$ **then** *FILTER5*$(b_1, b_2, m, k)$;

*Case 6* : **if** $x_k(q^{mid}) < x_k(p^1)$ **then** *FILTER6*$(b_1, b_2, m, k)$;

**end**;

**end** *of FILTER_G*;

Clearly, by the definition of hypercube, the 6 conditions in step (8) can be checked in $O(m) = C_3\ m$ time, where $C_3$ is a constant.

The procedure *FILTER_G2* in step (2), which is adapted from the 2-d maxima algorithm in [D86], uses divide-and-conquer method to solve the problem recursively and in parallel as follows.

**procedure** *FILTER_G2*$(b_1, b_2, m)$;

**begin**

(1) Sort the received point set $s$ ( a subset of $S$) by $x_2$ coordinate values in increasing order. Hence, the left half L of $s$, with lower $x_2$ values, will be moved to subcube $S_{m-1}(b_1)$ and the right half R of $s$, with higher $x_2$ values, will be moved to subcube $S_{m-1}(b_2)$.

(2) In order to combine the solutions in both subcubes, the processor containing a

point $p \in$ R with minimum $x_2$ value ($x_2$_min) with respect to R broadcasts $x_1(p)$ and $x_2$_min to all PEs in $S_{m-1}(b_1)$.

(3) If any PE in $S_{m-1}(b_1)$ containing a point $t \in$ L that is a maximum with respect to L but $x_1(t) \leq x_1(p)$ then set their *flag* field of the current top stack entry to 1. This means that the point is dominated by a point in R.

(4) Recursively solve the problem by steps (2) and (3).

**end** of *FILTER_G2*;

For the time complexity of *FILTER_G2*, we know that step (1) needs $O(log^2 n)$ time by hypercube bitonic sorting, while steps (2) and (3) need only $O(log\ n)$, respectively. Therefore, totally *FILTER_G2* needs $O(log^2 n)$ time, where $n$ is the size of $s$.

For executing *FILTER_G*($b_1$, $b_2$, $m$, $k$), we enforce the following conditions to be held always.

(*A*) The previous stated filter invariant condition hold for all *FILTER_G*($b_1$, $b_2$, $m$, $k$) calls for $k \geq 3$. When $k = 2$, current top stack entry point of any $PE_i$ in $S_{m-1}(b_1)$ is dominated by any current top stack entry point of any $PE_j$ in $S_{m-1}(b_2)$ in coordinates $x_3, ..., x_d$.

(*B*) Subalgorithms *FILTER_S*($b_1$, $b_2$, $m$, $k$) and *FILTER_G2*($b_1$, $b_2$, $m$) correctly set the *flag* fields of current top stack entries during their computation.

(*C*) If $STK_i[SP_i]$. *point* in $PE_i$ of $S_{m-1}(b_1)$ is dominated by a $STK_j[SP_j]$. *point* in $PE_j$ of $S_{m-1}(b_2)$ in the $x_k$ coordinate value at the current recursion level, then *FILTER_G* continues to test whether or not it is dominated by $STK_j[SP_j]$. *point* in the $x_{k-1}$ coordinate value at lower recursion levels.

(*D*) If the $x_k$ coordinate value of $STK_i[SP_i]$. *point* in $PE_i$ of $S_{m-1}(b_1)$ has not been detected not dominated by the $x_k$ coordinate values of some points

$STK_j[SP_j]$. *point* in $PE_j$ of $S_{m-1}(b_2)$ in the current recursion level, then *FILTER_G* continues to test whether or not it is dominated by any of these points in the $x_k$ coordinate value at lower recursion levels.

(*E*) For any processor $PE_i$, if we list all *BROADCAST*, *CENSUS*, *SORT* and *UNSORT* operations $PE_i$ involved from left to right as time passes during the computation, then we will get a well-formed *BCSU* sequence.

In what follows, we describe the operations to be taken by *FILTER_G*($b_1$, $b_2$, $m$, $k$), when $m > 3$ and $k > 2$, on a case-by-case basis. Let $S^1$ and $S^2$ be two sets of subgraphs of processors of $Q_n$. To simplify our presentation, we adopt the notion $filter_j[S^1|S^2]$ for the following operation: find all points in the current top stack entries of processors in $S^1$ that are dominated in $x_l$ coordinate values, $1 \le l \le j$, by any point in the current top stack entries of processors in $S^2$, and set their *flag* fields to 1.

### 2.3.3.1 Case 1 of FILTER_G

For *Case 1* : $x_k(q^{mid}) > x_k(p^{max})$, procedure *FILTER*1 has two tasks, as in Figure 2.5.

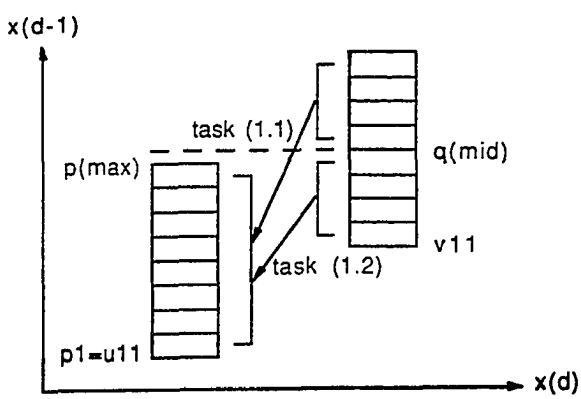


Fig. 2.5 Tasks for Case 1 in subalgorithm *FILTER_G*.

**procedure** *FILTER*1($b_1$, $b_2$, $m$, $k$);

**begin**

(1.1) call procedure *FILTER_K*1(1, 4, $m-2$, $v3$);

(* do $filter_{k-1}[S_{m-1}(u_1)|S_{m-2}(v_3)]$ *)

(1.2) call procedure *FILTER_K*(1, 4, $m-2$, $v1$);

(* do $filter_k[S_{m-1}(u_1)|S_{m-2}(v_1)]$ *)

**end** of *FILTER1*;

The procedures *FILTER_K*1 and *FILTER_K* are as follows.

**procedure** *FILTER_K*1(*first*, *last*, *mm*, *vv*);

**begin**

*BROADCAST*(*vv*, *mm*, *vv*, *mm*, $v_1$, $m-1$);

**for** $r :=$ *first* **to** *last* **do**

**begin**

*BROADCAST*($u_{r1}$, *m*-**4**, $u_1$, $m-2$, $u_1$, $m-1$);

*BROADCAST*($u_{r2}$, *m*-**4**, $u_3$, $m-2$, $u_1$, $m-1$);

**for all** $S_{m-4}(u_{ij})$ **and** $S_{m-4}(v_{ij})$, $1 \le i \le 4$ **and** $1 \le j \le 2$,

**parbegin**

*SORT*($u_{ij}$, $m-4$, $k-1$); *SORT*($v_{ij}$, $m-4$, $k-1$);

**parend;**

**for all** $S_{m-4}(u_{ij})$ **and** $S_{m-4}(v_{ij})$, $1 \le i \le 4$ **and** $1 \le j \le 2$,

**parbegin**

*FILTER_G*($u_{ij}$, $v_{ij}$, $m-3$, $k-1$);

**parend;**

**for all** $S_{m-4}(u_{ij})$ **and** $S_{m-4}(v_{ij})$, $1 \le i \le 4$ **and** $1 \le j \le 2$,

```
            parbegin
              UNSORT(u_ij, m − 4, k − 1); UNSORT(v_ij, m − 4, k − 1);
            parend;
          CENSUS(u_r2, m-4, u_3, m − 2, u_1, m − 1);
          CENSUS(u_r1, m-4, u_1, m − 2, u_1, m − 1);
        end; (* end of outer for loop *)
      CENSUS(vv, mm, vv, mm, v_1, m − 1);
    end;    (* end of procedure FILTER_K1 *)

procedure FILTER_K(first, last, mm, vv);
    begin
      BROADCAST(vv, mm, vv, mm, v_1, m − 1);
      for r := first to last do
        begin
          BROADCAST(u_r1, m-4, u_1, m − 2, u_1, m − 1);
          BROADCAST(u_r2, m-4, u_3, m − 2, u_1, m − 1);
          for all S_{m−4}(u_ij) and S_{m−4}(v_ij), 1 ≤ i ≤ 4 and 1 ≤ j ≤ 2,
            parbegin
              FILTER_G(u_ij, v_ij, m − 3, k − 1);
            parend;
          CENSUS(u_r2, m-4, u_3, m − 2, u_1, m − 1);
          CENSUS(u_r1, m-4, u_1, m − 2, u_1, m − 1);
        end; (* end of outer for loop *)
      CENSUS(vv, mm, vv, mm, v_1, m − 1);
    end;    (* end of procedure FILTER_K *)
```

Procedure *FILTER_K*1 carries out task (1.1) and it has four iterations. Each iteration has eight concurrent calls of *FILTER_G*($u_{ij}, v_{ij}, m-3, k-1$), and engages $2^{m-3}$ *PE*s that form an $(m-3)$-dimensional subcube in the general sense. By their preceding *SORT* operations, for each of the thirty-two calls, the condition (*A*) is enforced, assuming that (*A*) holds for $x_k, ..., x_d$ coordinate values of the points in the current top stack entries of all involving processors before call procedure *FILTER_K*1. The *BROADCAST* operations associated with these calls enforce conditions (*C*) and (*D*). Furthermore, it is easy to see that all *BROADCAST / CENSUS* and *SORT / UNSORT* pairs associated with any processor in $S_{m-1}(b_1)$ form a partial well-formed *BCSU* sequence ... <<( ... )>> ..., where "..." between "(" and ")" is the *BCSU* sequence that will appear in the lower recursion levels of *FILTER_G*, and the two subsequences denoted as "..." not included between "(" and ")" correspond to the operations *BROADCAST*, *CENSUS*, *SORT*, and *UNSORT* performed in the higher recursive levels.

Clearly, if for each processor involved, *BROADCAST*, *CENSUS*, *SORT*, and *UNSORT* operations performed in the higher and lower recursion levels satisfy the requirement of well-formed BCSU sequence, then the entire BCSU sequence for the processor is well-formed, which is condition (*E*). Similar arguments can be used to show that conditions (*A*), (*C*), (*D*) and (*E*) hold for procedure *FILTER_K*, assuming that they hold for all other cases and in all recursion levels of concern (not including the bottom levels in which either *FILTER_S* or *FILTER_G2* is executed).

For task (1.1), which is carried out in procedure *FILTER_K*1, eight parallel calls of *FILTER_G*($u_{ij}, v_{ij}, m-3, k-1$) are created and executed concurrently for each of four iterations. Similarly, for task (1.2), eight parallel calls of *FILTER_G*($u_{ij}, v_{ij}, m-3, k$) are created and executed concurrently for each of the

four iterations in procedure *FILTER_K*. Let $F_k(2^m)$ denote the time taken by *FILTER_G*($b_1$, $b_2$, $m$, $k$). Summarizing the previous discussions, we have

$$F_k(N) = 4F_k(N/8) + 4F_{k-1}(N/8) + 4S(N/8) + O(\log N),$$

where $S(N/8)$ is the time taken by a pair of *SORT* and *UNSORT*, and $O(\log N)$ is the time taken by all *BROADCAST* and *CENSUS* operations in the current recursion level.

### 2.3.3.2 Cases 2 to 6 of FILTER_G

For cases 2 to 6 in step (8) of FILTER_G, we develop a procedure for each case in the following.

- *Case 2* : $x_k(p^4) < x_k(q^{mid}) < x_k(p^{max})$. There are three tasks in procedure *FILTER2* (see Figure 2.6).

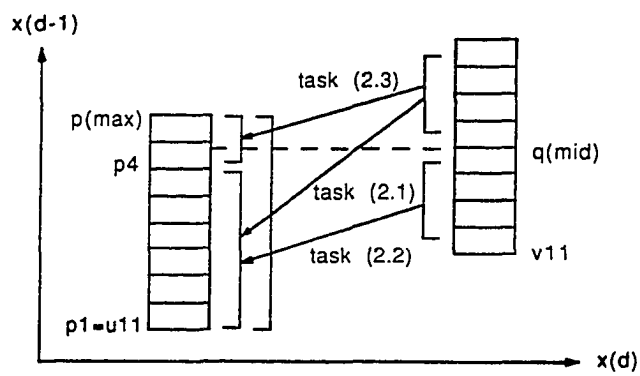


Fig. 2.6 Tasks for Case 2 in subalgorithm *FILTER_G*.

**procedure** *FILTER2*($b_1$, $b_2$, $m$, $k$);

**begin**

(2.1) call procedure *FILTER_K*1(1, 3, $m-2$, $v3$);

(* do $filter_{k-1}[\{S_{m-3}(u_1), S_{m-3}(u_2), S_{m-3}(u_3)\} \mid S_{m-2}(v_3)]$ *)

(2.2) call procedure *FILTER_K*(1, 4, $m-2$, $v1$);

(* do $filter_k[S_{m-1}(u_1)|S_{m-2}(v_1)]$ *)

(2.3) call procedure *FILTER_K*(4, 4, $m-2$, $v3$);

(* do $filter_k[S_{m-3}(u_4)|S_{m-2}(v_3)]$ *)

**end** of *FILTER*2;

Following the idea used in *Case 1*, we can carry out these tasks in three steps, steps 2.1, 2.2 and 2.3. In step 2.1, task (2.1) is divided into eight parallel *FILTER_G*($u_{ij}$, $v_{ij}$, $m-3$, $k$) calls. In step 2.2, task (2.2) is carried out in three substeps, each substep performing eight parallel *FILTER_G*($u_{ij}$, $v_{ij}$, $m-3$, $k-1$) calls. In step 2.3, task (2.3) is carried out in four substeps, each step performing eight concurrent *FILTER_G*($u_{ij}$, $v_{ij}$, $m-3$, $k$) calls. Therefore, the total time required is

$$F_k(N) = 5F_k(N/8) + 3F_{k-1}(N/8) + 3S(N/8) + O(\log N).$$

Furthermore, conditions (*A*), (*C*), (*D*) and (*E*) hold for this case, assuming that they hold for all other cases in all recursion levels of concern.

For the remaining four cases, we only list the tasks need to be carried out, and the time complexities. Using *Case 1* and *Case 2* as references, the details of the involved operations can be easily derived, and conditions (*A*), (*C*), (*D*) and (*E*) can be partially verified, for each of these remaining cases. For brevity, we omit these details.

- *Case 3* : $x_k(p^3) < x_k(q^{mid}) < x_k(p^4)$. There are three tasks in procedure *FILTER*3.

**procedure** *FILTER*3($b_1$, $b_2$, $m$, $k$);

**begin**

(3.1) call procedure *FILTER_K*1(1, 2, $m-2$, $v3$);

(* do $filter_{k-1}[S_{m-2}(u_1)|S_{m-2}(v_3)]$ *)

(3.2) call procedure *FILTER_K*(3, 4, $m-2$, $v3$);

(* do $filter_k[S_{m-2}(u_3)|S_{m-2}(v_3)]$ *)

(3.3) call procedure *FILTER_K*(1, 3, $m-2$, $v1$);

(* do $filter_k[\{S_{m-3}(u_1), S_{m-3}(u_2), S_{m-3}(u_3)\} \mid S_{m-2}(v_1)]$ *)

**end** of *FILTER*3;

The time required for this case is :

$$F_k(N) = 5F_k(N/8) + 2F_{k-1}(N/8) + 2S(N/8) + O(log\ N).$$

- *Case 4* : $x_k(p^2) < x_k(q^{mid}) < x_k(p^3)$. There are 3 tasks in procedure *FILTER*4.

**procedure** *FILTER*4($b_1$, $b_2$, $m$, $k$);

**begin**

(4.1) call procedure *FILTER_K*1(1, 1, $m-2$, $v3$);

(* do $filter_{k-1}[S_{m-3}(u_1)|S_{m-2}(v_3)]$ *)

(4.2) call procedure *FILTER_K*(1, 3, $m-2$, $v3$);

(* do $filter_k[\{S_{m-3}(u_1), S_{m-3}(u_2), S_{m-3}(u_3)\} \mid S_{m-2}(v_3)]$ *)

(4.3) call procedure *FILTER_K*(1, 2, $m-2$, $v1$);

(* do $filter_k[S_{m-2}(u_1)|S_{m-2}(v_1)]$ *)

**end** of *FILTER*4;

The time required for this case is :

$$F_k(N) = 5F_k(N/8) + F_{k-1}(N/8) + S(N/8) + O(log\ N).$$

- *Case 5* : $x_k(p^1) < x_k(q^{mid}) < x_k(p^2)$. Two tasks are to be carried out in procedure *FILTER*5.

**procedure** *FILTER5*($b_1$, $b_2$, $m$, $k$);

**begin**

(5.1) call procedure *FILTER_K*(1, 4, $m-2$, $v3$);

(* do $filter_k[S_{m-1}(u_1)|S_{m-2}(v_3)]$ *)

(5.2) call procedure *FILTER_K*(1, 1, $m-2$, $v1$);

(* do $filter_k[S_{m-3}(u_1)|S_{m-2}(v_1)]$ *)

**end** of *FILTER5*;

The time required for this case is :

$F_k(N) = 5F_k(N/8) + O(log\ N)$

- *Case 6* : $x_k(q^{mid}) < x_k(p^1)$. One task to be carried out in procedure *FILTER6*.

**procedure** *FILTER6*($b_1$, $b_2$, $m$, $k$);

**begin**

(6.1) call procedure *FILTER_K*(1, 4, $m-2$, $v3$);

(* do $filter_k[S_{m-1}(u_1)|S_{m-2}(v_3)]$ *)

**end** of *FILTER6*;

The time required for this case is :

$F_k(N) = 4F_k(N/8) + O(log\ N)$.

### 2.3.3.3 Proof the Correctness of FILTER_G

Before we analyze the time complexity of algorithm *MAXIMA*, let us first show that subalgorithm *FILTER_G*($b_1, b_2, m, k$) is correct. Let us consider conditions (*A*) through (*E*) in section 4.3 for the subalgorithm *FILTER_G*($b_1$, $b_2$, $m$, $k$). For

procedure *FILTER_G2*, it is converted correctly from the 2-d maxima algorithm in [D86]. For *FILTER_S*, we know that condition (*B*) can be easily enforced. Since conditions (*A*), (*C*), (*D*) and (*E*) are enforced for all six cases (in a way as we demonstrated in *Case 1*), by a simple induction we know that conditions (*A*) through (*E*) hold for *FILTER_G*.

Then, we need to show that these conditions ensure that all points in the current top stack entries of all $PE_i$ in $S_{m-1}(b_1)$ that are dominated by any current top stack entry point of $S_{m-1}(b_2)$ will be eventually marked. Condition (*A*) allows for comparing the $x_l$, $l = k, k\text{-}1, \ldots, 1$, coordinate values of current top stack entry points of $S_{m-1}(b_1)$ with current top entry points of $S_{m-1}(b_2)$. Conditions (*C*) and (*D*) guarantee that all current top stack entry points of $S_{m-1}(b_1)$ dominated by any current top stack entry point of $S_{m-1}(b_2)$ will be eventually marked at the bottom level of recursion by either *FILTER_S* or *FILTER_G2*, using condition (*B*). Condition (*E*) is important, since it makes conditions (*A*), (*C*) and (*E*) valid during the entire computing process, and it allows for comparing the current top stack entry points of any $PE_i$ in $S_{m-1}(b_1)$ with many current top entry points of $S_{m-1}(b_2)$ at lower recursion levels, and carrying all the comparison results back so that the final status of this point can be properly recorded and reported.

## 2.4 Analysis of Time Complexity

At first, let us consider the time complexity of *FILTER_G*. Summarizing all six cases, we have

$$F_k(N) = \begin{cases} 4F_k(N/8) + 4F_{k-1}(N/8) + 4S(N/8) + O(\log N), & \text{for } \textit{Case } 1; \\ 5F_k(N/8) + 3F_{k-1}(N/8) + 3S(N/8) + O(\log N), & \text{for } \textit{Case } 2; \\ 5F_k(N/8) + 2F_{k-1}(N/8) + 2S(N/8) + O(\log N), & \text{for } \textit{Case } 3; \\ 5F_k(N/8) + F_{k-1}(N/8) + S(N/8) + O(\log N), & \text{for } \textit{Case } 4; \\ 5F_k(N/8) + O(\log N), & \text{for } \textit{Case } 5; \\ 4F_k(N/8) + O(\log N), & \text{for } \textit{Case } 6. \end{cases}$$

Since sorting $N = 2^n$ numbers on a hypercube machine $Q_n$ can be done in $O(log^2 N)$ time, $S(N) = O(log^2 N)$. Also, as we mentioned before, the time complexity $F_2(N)$, where $N = 2^n$, of *FILTER_G2*($b_1$, $b_2$, $n$) is $O(\log^2 N)$ time. Since both $S(N)$ and $F_2(N)$ are $O(\log^2 N)$, we may select a constant $C_2$ such that $S(N) \le C_2 \log_8^2 N$ and $F_2(N) \le C_2 \log_8^2 N$. We can also select a constant $C_3$ such that $C_3 \log_8 N$ is greater than each of the $O(log\ N)$ terms in the above relations. Remember that *FILTER_S* takes $C_1$ amount of time. Using these constants, the worst-case performance of *FILTER_G* can be determined by looking at the following relations.

(i) $F_k(N) = 4F_k(N/8) + 4F_{k-1}(N/8) + 4C_2 \log_8^2 N + C_3 \log_8 N$;

(ii) $F_k(N) = 5F_k(N/8) + 3F_{k-1}(N/8) + 3C_2 \log_8^2 N + C_3 \log_8 N$;

(iii) $F_k(N) = C_1$, if $N \le 8$; and

(iv) $F_2(N) = C_2 \log_8^2 N$.

Let

$$F_k^1(N) = 4F_k(N/8) + 4F_{k-1}(N/8) + 4C_2 \log_8^2 N + 4C_3 \log_8 N \quad \text{and}$$

$$F_k^2(N) = 5F_k(N/8) + 5F_{k-1}(N/8) + 5C_2 \log_8^2 N + 5C_3 \log_8 N.$$

Clearly, $F_k(N) < \max\{F_k^1(N), F_k^2(N)\}$. For $F_3^1(N)$, we have

$$F_3^1(N) = 4^j F_3^1(N/8^j) + \sum_{i=1}^{j} 4^i \{F_2^1(N/8^i) + C_2 \log_8^2(N/8^{i-1}) + C_3 \log_8(N/8^{i-1})\}$$

$$< C_1 4^{\log_8 N-1} + (1 + C_2 + C_3) \sum_{i=1}^{\log_8 N-1} 4^i (\log_8 N - i)^2$$

$$< C_1 4^{\log_8 N-1} + (1 + C_2 + C_3) 4^{\log_8 N} \log_8^2 N$$

$$< C 4^{\log_8 N} \log_8^2 N,$$

where $C = 1 + C_1 + C_2 + C_3$ is a constant. We prove that $F_k^1(N) < kC4^{\log_8 N} \log^{k-1} N$, for $k > 3$, by induction. Suppose that this claim is true for $k$-1, and consider $F_k^1(N)$.

$$F_k^1(N) = 4^j F_k^1(N/8^i) + \sum_{i=1}^{j} 4^i F_{k-1}^1(N/8^i) + C_2 \sum_{i=1}^{j} 4^i \log_8^2(N/8^{i-1}) + C_3 \sum_{i=1}^{j} 4^i \log_8(N/8^{i-1})$$

$$< C_1 4^{\log_8 N} + (k-1)C \sum_{i=1}^{\log_8 N-1} 4^i 4^{\log_8(N/8^i)} \log_8^{k-2}(N/8^i) + (C_2 + C_3) \sum_{i=1}^{\log_8 N-1} 4^i (log_8 N - i)^2$$

$$< C_1 4^{\log_8 N} + (k-1)C \sum_{i=1}^{\log_8 N-1} 4^{\log_8 N} \log_8^{k-2}(N/8^i) + (C_2 + C_3) 4^{\log_8 N} \log_8^2 N$$

$$< ((k-1)C + C_1 + C_2 + C_3) 4^{\log_8 N} \log_8^{k-1} N$$

$$< kC4^{\log_8 N} \log_8^{k-1} N.$$

This completes the induction. By similar analysis, we have

$$F_k^2(N) < kC5^{\log_8 N} \log_8^{k-1} N .$$

Therefore, $F_k(N) = \max \{F_k^1(N), F_k^2(N)\} = O(5^{\log_8 N} \log^{k-1}(N))$.

Let $T(N)$ be the total running time of algorithm *MAXIMA*, then

$$T(N) = T(N/2) + F_{d-1}(N) = O(5^{\log_8 N} \log^{d-1} N)$$

$$= O(N^{\log_8 5} \log^{d-1} N) = O(N^{0.77} \log^{d-1} N).$$

In summary, our hypercube algorithm *MAXIMA* solves the $d$-dimensional maxima problem of input size $N$ in $O(N^{0.77}\log^{d-1} N)$ time, using $O(N)$ processors. The final results are in all bottom stack entries (the 0-th entries) of all stacks (note that when *MAXIMA* is completed, the 0-th stack entry is also the current top stack entry) such that $STK_j[0].$ *point* is a maximum point if and only if $STK_j[0].$ *flag* = 0.

## 2.5 Generalizations and Concluding Remarks

A parallel algorithm for the maxima problem is adaptive if for a fixed $P$, which is the number of processors, it can be still usable and efficient when the input size increases. We can modify our algorithm *MAXIMA* to obtain an adaptive hypercube algorithm *MAXIMA_A*. Without loss of generality, assume that $N$ is a multiple of $P$. Then, initially each processor contains $N/P$ points.

To generalize our algorithm, we implement a stack in each processor such that each entry is an array of $N/P$ elements, each having two fields *point* and *flag*. By slightly modifying *BROADCAST*, *CENSUS*, *SORT* and *UNSORT* we can obtain similar algorithms *BROADCAST_A*, *CENSUS_A*, *SORT_A* and *UNSORT_A* for *MAXIMA_A*. Let $S^A(N)$ denote the time for the *SORT_A* operation, which sorts $N$ numbers in $n$-dimensional hypercube of $P$ processors such that the sorted sequence are distributed in the memories associated to these processors, each with $N/P$ numbers. By bitonic sorting algorithm, $S^A(N) = O((N/P)(log(N/P)+\log^2 P))$. The *BROADCAST_A* can be done in no more than $O((N/P)logP)$ time. The time complexity for *CENSUS_A* and *UNSORT_A* are the same as that for *BROADCAST_A* and *SORT_A*, respectively.

The generalized algorithm of *FILTER_S*($b_1$, $b_2$, $m$) can be converted into an algorithm *FILTER_S_A*($b_1$, $b_2$, $m$) that takes $O(N/P)$ time, since $m \leq 3$. By modifying the algorithm in [D86], we can obtain an algorithm *FILTER_G2_A*($b_1$, $b_2$, $m$), which is a generalized version of *FILTER_G2*($b_1$, $b_2$, $m$), that takes $O((N/P)(log(N/P) + m^2))$ time. Then *FILTER_G_A*($b_1$, $b_2$, $m$, $k$), the generalized version of *FILTER_G*($b_1$, $b_2$, $m$, $k$), is almost the same as the one described in the previous section. The six cases of *FILTER_G_A*($b_1$, $b_2$, $m$, $k$) are now defined slightly differently. Point $p_j$, $1 \leq j \leq 4$, is the point with the smallest $x_k$ coordinate value among all $N/P$ points in the current top stack entry of $S_{m-1}(u_j)$. Point $p^{\max}$ is the point with the largest $x_k$ coordinate value among all $N/P$ points in the current top stack entry of $S_{m-1}(u_4)$, and point $q^{mid}$ is the point with the smallest $x_k$ coordinate value among all $N/P$ points in the current top stack entry of $S_{m-1}(v_3)$.

Determining which of the six cases is true takes $O(m)$ time. Using similar analysis for $F_k(N)$, we obtain $F_k^A(2^m) = O(P^{0.77}(N/P)(\log(N/P)\log^{k-2} P + m^{k-1})) = O(P^{0.77}(N/P)\log^{k-2} P(\log(N/P) + \log P))$, which is the running time of *FILTER_G_A*($b_1$, $b_2$, $n$, $k$). Since the total time for the generalized maxima algorithm is $T^A(P) = T^A(P/2) + F_{d-1}^A(P)$, we know that $T^A(P) = O((N/P^{0.23})\log^{d-2} P(\log(N/P) + \log P))$. According to the currently best sequential algorithm, the best parallel algorithm should not take less than $O((N/P)\log^{d-2} N)$ time. However, this does not mean that a parallel algorithm running in less than $O((N/P)\log^{d-2} N)$ time is not possible, since it is not known whether or not $\Omega(N\log^{d-2} N)$ is the lower bound for the sequential time complexity of the maxima problem.

As we know, our algorithm *MAXIMA* is the first sublinear-time algorithm for the $d$-dimensional maxima problem on parallel machines with sparse interconnection networks when $d > 3$. The main factor in achieving the sublinear time complexity is subalgorithm *FILTER_G*. This algorithm partitions a given problem or subproblem into several subproblems, which we call as tasks, and each is assigned to a set of subcubes so that the load of all processors are balanced and the data communication between processors through the links connecting them is minimized as much as possible. One may wonder that the overhead caused by operations *SORT* and *UNSORT* might be a reason why the complexity is not as low as the algorithms on the PRAM models. By the analysis of $F_k(N)$, we note that even if we can somehow eliminate all *SORT* and *UNSORT* operations in the recursive process, the time complexity remains the same. The main reason for the factor $N^{0.77}$ lies in the way the problem is partitioned. It is interesting to note that we have tried to partition the operation $filter_k[S^1|S^2]$ in several different ways, and our analysis shows that the complexity we obtained is the best among all the possibilities we have tried.

It remains as an open problem whether or not our results can be significantly improved. For example, is it possible to obtain a hypercube maxima algorithm that runs in $O(N^c)$ time such that $c$ is a constant less than 1? More challengingly, can we design a hypercube maxima algorithm that runs in $O(\log^{f(d)})$ time, where $f(d)$ is a linear function of dimension $d$ of the point space? We believe that due to the limitation of data communication in hypercube machines, finding these algorithms are not simple.

Theoretically, our algorithm *MAXIMA* runs in less than linear time. In practice, however, the situation may be different. Since a hypercube contains a Hamiltonian circuit (as given by Gray Code), a pipeline algorithm using the ring structure

embedded in the hypercube runs in $O(dN)$ time. Such an algorithm may be better than the performance of *MAXIMA* for small $d$. Considering the communication overheads, sequential algorithms can be better than their parallel counterparts. Combining the sequential algorithm and pipeline processing on the ring embedded in the hypercube, we can easily obtain a hypercube maxima algorithm of time complexity $O(N \log^{d-3}(N/P) + (N/P)(\log^{d-2}(N/P) + \log^2 P))$ for the case $N > O(P)$. In general, one may combine our algorithm with pipeline algorithm and sequential algorithm and determine when and how they should be employed to yield an efficient parallel hypercube maxima algorithm that is most suitable for given $n$, $N$, and $d$, which are the dimension of hypercube, input point set and the dimension of point space, respectively.

# Chapter 3

# The Voronoi Diagram

## 3.1 Introduction

The Voronoi diagram of a set $S$ of $N$ points in the plane is a polygonal subdivision of $N$ regions in the plane, each containing a point of $S$, such that the region for each point $p$ of $S$ is the locus of points in the plane which are closer to $p$ than to other points of $S$. The Voronoi diagram is a very useful data structure with many applications in computational geometry, robotics, image processing, and graph theory ([PH77], [S78]).

The sequential time complexity of the problem of constructing a Voronoi diagram is $O(N \log N)$, where $N = |S|$. The $O(N \log N)$-time sequential algorithm described in [PH77] uses the divide-and-conquer strategy, but its merging step seems difficult to be parallelized . Previous parallel algorithms are based on the sequential algorithm proposed by Brown who reduced the construction of Voronoi diagram to the problem of finding the three-dimensional convex hull of a set of points on a sphere [B79]. In [C80], Chow gave a PRAM algorithm with time complexity $O(log^3 N \log \log N)$, where $N$ is the number of PEs in the PRAM. This has been improved to $O(log^3 N)$ in [ACGOY85]. Chow also proposed an $O(log^4 N)$-time algorithm on Cube-Connected-Circles [C80]. In [L86], Lu presented an $O(\sqrt{N} \log N)$-time algorithm for constructing the Voronoi diagram on the $\sqrt{N} \times \sqrt{N}$ mesh-connected computer. In [S88b], Stojmenovic presented an algorithm on the hypercube machine with $N$ processors, and the time complexity of this algorithm is $O(log^3 N)$ time.

In this chapter, we present an $O(log^2 N)$-time parallel algorithm for constructing the Voronoi diagram of a set $S$ of $N$ points in the plane on the $N \times N$ mesh of trees. Our algorithm is also based on the sequential algorithm presented in [B79]. More specifically, our algorithm transforms points of $S$ into a set $S'$ of points on a sphere, then uses the divide-and-conquer approach to construct the three-dimensional convex hull $CH(S')$ of $S'$, and finally transforms $CH(S')$ into the Voronoi diagram of $S$. Our convex hull algorithm can be used to construct the convex hull of any set of points in $E^3$ in $O(log^2 N)$ time. Our algorithms are the fastest $NC$ algorithms ever known for constructing planar Voronoi diagrams and three-dimensional convex hulls. This chapter is organized as follows. In section 3.2, we provide some fundamentals that are useful in the rest of the chapter. In section 3.3, we present our parallel Voronoi diagram algorithm. The subalgorithms used for constructing the three-dimensional convex hull of a set of points in $E^3$ will be presented in several subsections. We conclude this chapter in section 3.4.

## 3.2 Preliminaries

In this section, we describe the definition of planar Voronoi diagram, the structure and basic data routing operations of the mesh of trees, and the sequential algorithm for constructing Voronoi diagram as given in [B79].

### 3.2.1 Definition of Planar Voronoi Diagram

For any two points $p$ and $q$ in the plane, the set of points in the plane not farther to $p$ than to $q$ is the half-plane containing $p$ that is defined by the perpendicular bisector of the line segment connecting $p$ and $q$. We denote this half-plane by $H(p, q)$. For

a given set $S$ of points in the plane, the set of points not farther to $p_i \in S$ than to any other point in $S$ forms a convex polygonal region $V_i$ containing $p_i$ defined as

$$V_i = \bigcap_{\substack{p_j \in S \\ i \neq j}} H(p_i, p_j).$$

Each of these $N$ polygonal regions is called a *Voronoi polygon,* and the partition of the plane by these polygons is referred to as the *Voronoi diagram* of $S$, denoted by $Vor(S)$. The vertices and the line segments of the diagram are *Voronoi vertices* and *Voronoi edges,* respectively. To avoid degeneracies, we assume that *no four points of S are co-circular.* If a Voronoi polygon is bounded, then it is constructed entirely by the edges whose two endpoints are Voronoi vertices; if it is unbounded, then it includes two semi-infinite lines as edges. The semi-infinite edges are also called *rays*. Every Voronoi vertex $v$ is the common intersection of exactly three edges of the diagram, i.e. the degree of each vertex is 3; furthermore, $v$ is equidistant from the three points of $S$ which are closest to $v$, i.e. the circle determined by these three points is centered at $v$ and contains no other points of $S$. The Voronoi diagram of a point set is shown in Figure 3.1.

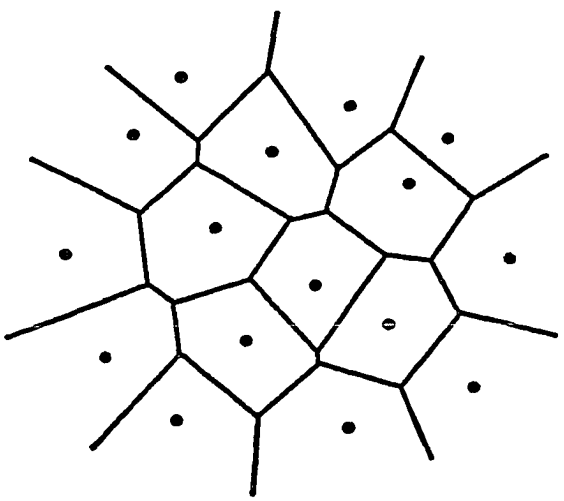

Fig. 3.1 The Voronoi diagram of a set of points.

### 3.2.2 Mesh of Trees and Data Communications

The $N \times N$ mesh of trees (*MOT*) is constructed from an $N \times N$ array of processors by adding processors and data links to form a complete binary tree in each row and column. The processors in the array are called *leaf processors,* denoted as $LP_{i,j}$, $0 \leq i, j \leq N-1$. The processors corresponding to the roots of the trees are called *root processors;* and the root processor of processors in row $i$ (resp. column $j$) is denoted as $RP_i^r$ (resp. $RP_j^c$). All remaining processors are called *internal processors* (denoted as *IP*'s). The total number of processors is $3N^2 - 2N$. Most of the computing is done in the $N^2$ $LP_{i,j}$'s. The remaining processors are used for communication between *LP*s, and may also be used to carry out some simple operations such as summing and comparison. A $4 \times 4$ *MOT* is shown in Figure 3.2.

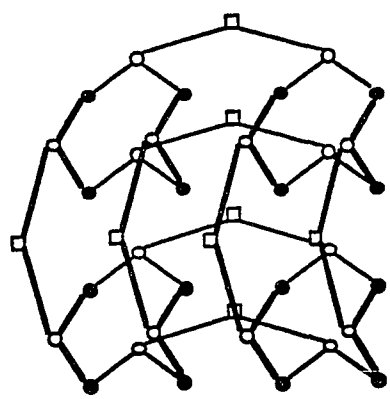

Fig. 3.2 A $4 \times 4$ mesh of trees (• : LP, ○ : IP, □ : RP).

The mesh of trees is the fastest general parallel multiprocessor architecture known when considered only in terms of speed. Many problems can be solved on the mesh of trees in $O(log^c N)$ time, where $c$ is a small constant. The following operations on the mesh of trees are useful for constructing more powerful operations that can be used for designing efficient algorithms.

*RTL_R(i, j)* : root to leaf in a row. Send a data item from $RP_i^r$, the root of row *i*, to $LP_{i,j}$ in the same row tree.

*LTR_R(i, j)* : leaf to root in a row. Send a data item from the leaf processor $LP_{i,j}$ to $RP_i^r$, the root of row *i*.

*LTL_R(i, j, k)* : leaf to leaf in a row. Send a data item from $LP_{i,j}$ to $LP_{i,k}$.

*BROADCAST_R(i)* : broadcast in a row. Broadcast a data item from $RP_i^r$, the root of row *i*, to all *LP*'s in row *i*.

*AND_R(i)* : logic AND in a row. Perform the logic AND function of the specified data item in each *LP* in row *i* and return the final result up to the root processor $RP_i^r$.

*SUM_R(i)* : summation of a row. Find the summation of specified data stored in all *LP*'s in row *i* and store the result in $RP_i^r$, the root of row *i*.

*SORT_R(i)* : sort in a row. Sort *N* numbers, which are stored in *LP*'s in row *i*, in non-decreasing order so that the sorted numbers are put into the *LP*'s in the same row, one number per *LP*.

Similarly, we can define operations *RTL_C*, *LTR_C*, *LTL_C*, *BROADCAST_C*, *AND_C*, *SUM_C*, and *SORT_C* to be performed among processors in a column tree. Note that *LTL* is constructed by a pair of *LTR* and *RTL* operations, and *SORT* is constructed by using *RTL*, *LTR*, *LTL*, *BROADCAST* and *SUM* operations [L92]. Additional census operations such as *OR*, *MAX* and *MIN* performed on a row or column tree can also be defined. Each of these operations takes $O(logN)$ time.

Since the *MOT* can be recursively defined [L92], we define a sub-*MOT* as a substructure of the $N \times N$ *MOT* that satisfies the definition of *MOT*. We are particularly interested in the sub-*MOT* that contains leaf processors in the first row, row 0, of the

$N \times N$ *MOT*. We use $MOT(m,k)$ to denote the sub-*MOT* that contains $2^m \times 2^m$ leaf processors $LP_{i,j}$, $0 \le i \le 2^m - 1$ and $k2^m \le j \le (k+1)2^m - 1$, for $0 \le k < N/2^m$, together with internal processors and the root processor to form a complete tree in each row and column of the $2^m \times 2^m$ array. Clearly, $MOT(m,k)$ is a $2^m \times 2^m$ *MOT*, and all parallel operations, including those mentioned above, for the $N \times N$ *MOT* can be executed on all sub-*MOT*'s $MOT(m,k)$ concurrently.

### 3.2.3 Constructing Voronoi Diagram by Geometric Inversion

An algorithm for constructing the Voronoi diagram of a set $S$ of $N$ points in the plane using a geometric transformation called *inversion* is given in [B79]. The inversion $I$ is a point-to-point transformation determined by two parameters, $q$, the *center* of inversion, and $r$, the *radius* of inversion. An inversion transforms a plane which does not contain its inversion center to a sphere with the inversion center on it, and vice versa. The interior and exterior of the sphere correspond to the two half-spaces separated by the plane. For a set $S$ of points in the $xy$-plane, if $q$ is not in the $xy$-plane, then by inversion we obtain a set $S'$ of points on a sphere in $E^3$, i.e. for a point $p$ in the $xy$-plane, $I(q, r, p)$ is on the sphere. Another important property of inversion is involuntary, that is, $I(q, r, I(q, r, p)) = p$. Inversion transformation is described in detail in [D72].

The following sequential algorithm for constructing the Voronoi diagram of a set $S$ of points in the plane is based on the sequential algorithm given in [B79] and the parallel algorithms given in [C80] and [L86]. The proof of correctness of this algorithm is given in [B79]. This algorithm can be implemented with sequential time complexity $O(N \log N)$, which is optimal.

**Algorithm** *VORONOI*(*S*)

**begin**

(1) Choose $q = (0, 0, 1)$ as the inversion center and choose radius $r = 1$. Perform the inversion for each point $p_i = (x_i, y_i) \in S$ to obtain $p_i' = (x_i', y_i', z_i')$, where $x_i' = y_i' = x_i/(x_i^2 + y_i^2 + 1)$, and $z_i' = (x_i^2 + y_i^2)/(x_i^2 + y_i^2 + 1)$. Let the set of inversion images of points in $S$ be $S'$ (see Figure 3.3).

(2) Construct the convex hull of $S'$. Denote this convex hull by $CH(S')$, which is the set of all triangular faces $F_j$ on the hull. Each face is represented by the three end points of the edges bounding the triangle.

(3) Each face $F_j$ performs the "reinversion" to map its end points $p_{j,1}'$, $p_{j,2}'$ and $p_{j,3}'$ to the points $p_{j,1}$, $p_{j,2}$ and $p_{j,3}$ on the $xy$-plane, and find the center $c_j$ of their circumcircle. Each $c_j$ is a Voronoi vertex in $Vor(S)$. If $q$ and $CH(S')$ are on the same side of the plane containing $F_j$ then set $v_j = 1$ else set $v_j = 0$.

(4) For each $c_j$ with $v_j = 1$, find the point $c_k$ such that $F_k$ is an adjacent face of $F_j$ in $CH(S')$. Note that there are three such $c_k$'s. If $v_k = 1$, then $(c_j, c_k)$ is a Voronoi edge with end points $c_j$ and $c_k$; otherwise, the semi-infinite line originating from $c_j$ is a ray in $Vor(S)$.

**end** *VORONOI*;

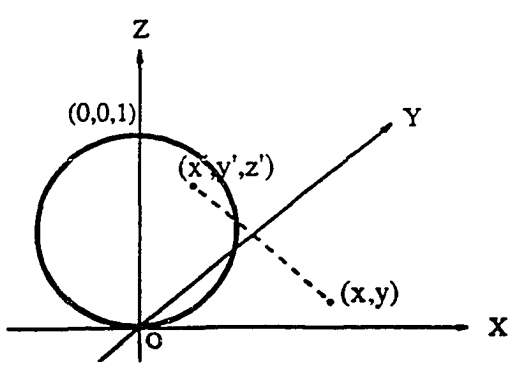


Fig. 3.3 Inversion between a point in the plane and its image on the sphere.

## 3.3 Constructing Voronoi Diagram on a Mesh of Trees

Our Voronoi diagram algorithm is derived directly from the algorithm *VORONOI* described in section 3.2.3. Assume there are exactly $N$ points in $S$, and initially each is stored in an $LP$ in the first row of the $N \times N$ mesh of trees. Then, step (1) of the parallel version of *VORONOI* can be done in $O(1)$ time. All faces of the three-dimensional convex hull $CH(S')$ are triangles, and there are no more than $2N$-4 faces in $CH(S')$. Suppose each processor $LP_{0,j}$ in the first row of the mesh of trees contains at most two faces of $CH(S')$, where each face is represented by the three points in $S'$ it passes through, and three edges connecting the points. Then, step (3) of the parallel version of algorithm *VORONOI* finds all Voronoi vertices of $Vor(S)$ in $O(1)$ time.

### 3.3.1 Constructing Voronoi Diagram from Convex Hull

Consider the parallel implementation of step (4) of algorithm *VORONOI*. Let the two faces stored in $LP_{0,j}$ be named as $F_{0,j,1}$ and $F_{0,j,2}$, respectively. The following procedure *MULTI_CAST* can be used to distribute all faces stored in row 0 of the mesh of trees to all $LP$'s such that each $LP_{i,j}$, $i \neq j$, contains faces $F_{0,i,1}$, $F_{0,i,2}$, $F_{0,j,1}$, and $F_{0,j,2}$.

**procedure** *MULTI_CAST*

**begin**

**for** $0 \leq j \leq N-1$ **do in parallel**

Use *LTR_C* to send $F_{0,j,1}$ and $F_{0,j,2}$ in $LP_{0,j}$ to $RP_j^c$;

Use *BROADCAST_C* to broadcast $F_{0,j,1}$ and $F_{0,j,2}$ in $RP_j^c$ to all $LP$'s in column $j$;

Use *LTR_R* to send $F_{0,j,1}$ and $F_{0,j,2}$ in $LP_{j,j}$ to $RP_j^r$;

Use *BROADCAST_R* to broadcast $F_{0,j,1}$ and $F_{0,j,2}$ in $RP_j^r$ to all *LP*'s in row $j$;

**endfor**

**end** *MULTI_CAST*

Consider the $4 \times 4$ *MOT*. The distribution of faces by *MULTI_CAST* is shown in Figure 3.4. In this figure, each 3-tuple $< 0, a, b >$ represents the face $F_{0,a,b}$ originally stored in $LP_{0,a}$. Procedure *MULTI_CAST* is used in several subalgorithms. Let us see how the step (4) of the parallel version of algorithm *VORONOI* can be carried out efficiently by using this procedure.

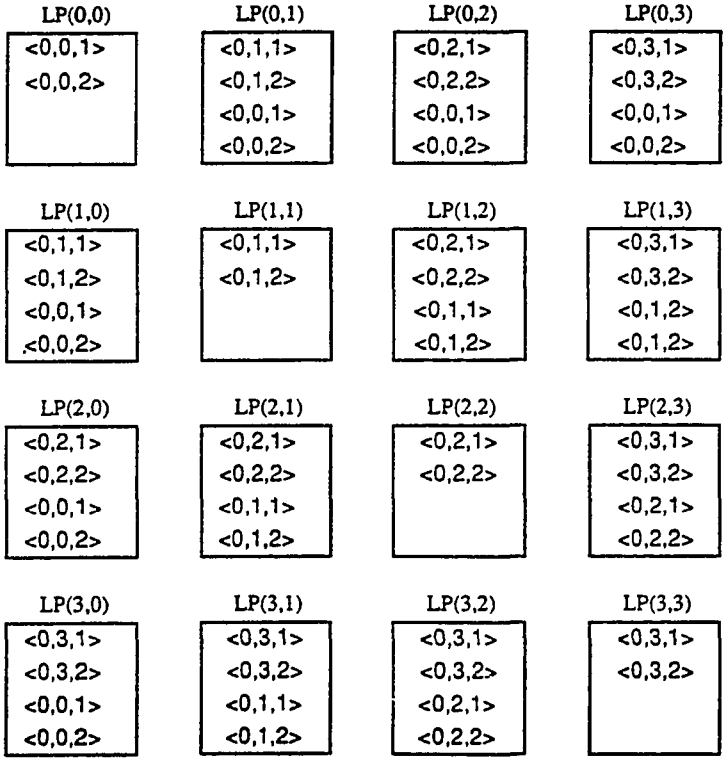


Fig. 3.4 The distribution of faces after the *MULTI_CAST* operation.

**procedure** *CH_TO_VOR*

**begin**

Broadcast $q$ to all *LP*'s of the mesh of trees.

Use procedure *MULTI_CAST* to distribute all faces stored in row 0 of the mesh of trees to all *LP*'s such that each $LP_{i,j}$, $i \neq j$, contains faces $F_{0,i,1}$, $F_{0,i,2}$, $F_{0,j,1}$, and $F_{0,j,2}$.

**for** all $LP_{i,j}$ **do in parallel**

if $i < j$ compare two pairs of faces, $F_{0,i,1}$ and $F_{0,j,1}$, and $F_{0,i,1}$ and $F_{0,j,2}$, to see whether they are adjacent or not. If yes, compute the Voronoi edges determined by them;

if $i > j$ compare two pairs of faces, $F_{0,i,2}$ and $F_{0,j,1}$, and $F_{0,i,2}$ and $F_{0,j,2}$, to see whether they are adjacent or not. If yes, compute the Voronoi edges determined by them;

**endfor**

**end** *CH_TO_VOR*

Procedure *CH_TO_VOR* converts the convex hull of $S'$ to the Voronoi diagram of $S$. Its time complexity is obviously $O(logN)$, since the procedure *MULTI_CAST* requires $O(logN)$ time and the parallel for-statement takes $O(1)$ time. Let $T_{CH}(N)$ denote the time taken for constructing the convex hull of $N$ points in $E^3$. Since any convex hull algorithm on the $N \times N$ *MOT* requires $\Omega(logN)$ time, we have the following fact:

**Lemma 1** : The Voronoi diagram of a set $S$ of $N$ points in the plane can be constructed in $T_{CH}(N)$ time on an $N \times N$ mesh of trees.

### 3.3.2 Constructing Three-Dimensional Convex Hull

The crucial part of our Voronoi diagram algorithm is constructing the three-dimensional convex hull $CH(S')$ in parallel. Note that the points in $S'$ obtained by inversion have coordinates $x'$, $y'$ and $z'$. For convenience, we rename them as $x$, $y$ and $z$, respectively. It is important to note that the $x$ and $y$ coordinates we are used in the rest of section 3.3 are not the same as the ones used for points in $S$. Without loss of generality, assume that no two points in $S'$ have the same $x$-coordinate value. Our algorithm for constructing three-dimensional convex hull uses the divide-and-conquer technique to divide $S'$ into subsets, by the sorted $x$-coordinate values, constructs the convex hulls of these small subsets in parallel using sub-*MOT*'s, then merges these small three-dimensional convex hulls into larger three-dimensional convex hulls in parallel. An important invariant is that the faces of the convex hull of a subset of points are always stored in *LP*'s in the first row of the sub-*MOT* assigned to the subset, each *LP* containing no more than two faces, before the merge process of the next level starts. This invariant is required not only to simplify the algorithm, but also to ensure the claimed time complexity of the algorithm. The mainframe of our convex hull algorithm is given below.

**Algorithm** *CONVEX_HULL*

*Input*: a set of $N$ points on a sphere in $E^3$, one point in each $LP_{0,j}$;

*Output*: the set of faces in $CH(S')$, at most two faces in each $LP_{0,j}$.

**begin**

Sort all points in $LP_{0,j}$'s such that $x(p_{0,j}) < x(p_{0,j+1})$,where $p_{0,j}$ is the point in $LP_{0,j}$ and $x(p_{0,j})$ is the $x$-coordinate of $p_{0,j}$;

**for** all $2^i \times 2^i$ $MOT(2, k)$ such that $0 \leq k < N/4$ **do in parallel**

Construct the convex hull of 4 points in $4 \times 4$ sub-*MOT* $MOT(2, k)$;

**endfor**

**for** $i = 3$ **to** $\log N$ **do**

**for** all $MOT(i, k)$ such that $0 \leq k < N/2^i$ **do in parallel**

Use procedure *MERGE* to merge the convex hulls of points in two $2^{i-1} \times 2^{i-1}$ sub-*MOT*'s of $MOT(i, k)$ to obtain the convex hull of all points in $MOT(i, k)$;

**endfor**

**endfor**

**end** *CONVEX_HULL*

The sorting step takes $O(logN)$ time. Finding the convex hulls of four points in all $4 \times 4$ sub-*MOT*s can be done in constant time. There are $O(logN)$ iterations for recursively constructing the convex hulls of subsets of $S'$ by procedure *MERGE*. Thus, the overall time complexity of *CONVEX_HULL* depends on the efficiency of procedure *MERGE*.

### 3.3.3 Algorithm for Merging Two Convex Hulls

Merging a pair of smaller three-dimensional convex hulls $CH(A)$ and $CH(B)$, whose faces are stored in two sub-*MOT*'s, of point sets $A$ and $B$ to obtain the convex hull $CH(A \cup B)$ consists of two basic tasks: (1) for each face $F_{Ai}$ on $CH(A)$ (resp. $F_{Bj}$ on $CH(B)$), determine it as an external or internal face of $CH(A \cup B)$. An *external face* is a face which will not be "covered" by $CH(A \cup B)$ after the merging, while an *internal face* means one that will be covered in $CH(A \cup B)$. (2) Remove all internal faces of $CH(A)$ and $CH(B)$, and find new faces that will appear on $CH(A \cup B)$. By the

assumption that no two points in $A \cup B$ have the same $x$-coordinate value, $CH(A)$ and $CH(B)$ are separated by an $xy$-plane. Recall that a face in the convex hull of any subset of $S'$, with more than two points, is a triangle defined by three points in $S'$ and three edges connecting them. Thus, an external face in $CH(A)$ (resp. $CH(B)$) is adjacent to at most one internal face of $CH(A)$ (resp. $CH(B)$) along their shared edge. We refer to the external faces that are adjacent to internal faces as *critical external faces.*

It is a simple fact that all edges shared by external and internal faces of $CH(A)$ (resp. $CH(B)$) form a closed circuit. We denote this circuit by $C_A$ (resp. $C_B$)). Each new face of $CH(A \cup B)$ is defined either by two vertices on $C_A$ (they must be the endpoints of an edge in $C_A$) and one vertex on $C_B$, or by two vertices on $C_B$ (they must be the endpoints of an edge in $C_B$) and one vertex on $C_A$ (see Figure 3.5). Once all critical external faces are identified, new faces can be created along $C_A$ and $C_B$.

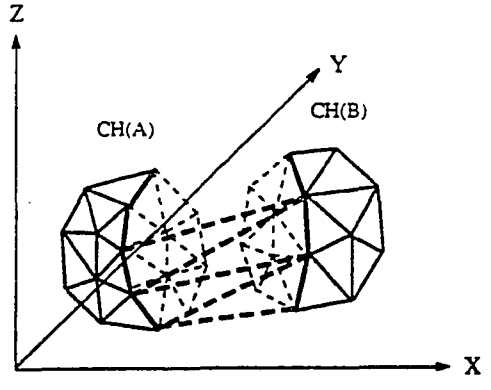


Fig. 3.5 Merge convex hulls $CH(A)$ and $CH(B)$. The thick solid line segments are the edges on circuits $C_A$ and $C_B$, respectively.

The procedure *MERGE* below is presented for merging on the $N \times N$ mesh of trees, which is $MOT(log_2 N, 0)$. The faces of $CH(A)$ and $CH(B)$ are stored in $LP$'s in the first rows of the sub-$MOT$'s $MOT(log_2 N - 1, 0)$ and $MOT(log_2 N - 1, 1)$, respectively. This procedure can be generalized to the ones for all sub-$MOT(i, k)$'s that are executed concurrently.

**procedure** *MERGE*

*Input*: Faces of $CH(A)$ and $CH(B)$ stored in the *LP*'s in row 0 of $MOT(log_2N-1,0)$ and $MOT(log_2N-1,1)$, respectively.

*Output*: Faces of $CH(A \cup B)$ stored in the *LP*'s in row 0 of $MOT(log_2N,0)$.

**begin**

Use procedures *CHECKFACE_A* and *CHECKFACE_B* to determine all internal faces of $CH(A)$ and $CH(B)$;

Use procedures *CRITICAL_FACE_A* and *CRITICAL_FACE_B* to determine all critical external faces of $CH(A)$ and $CH(B)$;

Use procedures *NEW_FACE_A* and *NEW_FACE_B* to discover all new faces of $CH(A \cup B)$;

Use procedure *REBALANCE* to enforce the invariant that each *LP* in the first row, row 0, of the mesh of trees contains at most 2 faces of $CH(A \cup B)$.

**end** *MERGE*

#### 3.3.3.1 Determining Internal and External Faces

The following description is for determining the external and internal faces in $CH(A)$. We use the same approach, but switch the roles of $CH(A)$ and $CH(B)$, to determine the external and internal faces in $CH(B)$. Let $P_{Ai}$ be the plane containing the face $F_{Ai}$ of $CH(A)$. $P_{Ai}$ divides the entire 3-dimensional space into two half-spaces. Denote the half-space that contains $CH(A)$ as $H_{Ai}$. Face $F_{Ai}$ belongs to $CH(A \cup B)$ if $CH(B)$ lies entirely in the half-space $H_{Ai}$. The equation for the plane containing the triangle face with three end points $(x_1, y_1, z_1)$, $(x_2, y_2, z_2)$, and $(x_3, y_3, z_3)$ can be expressed as follows:

$$\begin{bmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{bmatrix} = ax + by + cz + d = 0$$

All faces of $CH(A)$ are stored in $LP_{0,j}$, $0 \le i \le N/2\text{-}1$, and all faces of $CH(B)$ are stored in $LP_{0,j}$, $N/2 \le j \le N-1$. By our recursion invariant, each processor contains at most two faces. For convenience, we may enforce each of $LP$ to contain exactly two faces by adding dummy faces if necessary. These dummy faces will be replaced by new faces when they are generated. The following procedure determines for each of the faces of $CH(A)$, which are stored in $LP_{0,j}$, $0 \le j \le N/2\text{-}1$, whether it is internal or external. A flag, *extn*, is assumed for each face. We assume that each leaf processor $LP_{i,j}$ has a local variable $EXT_{i,j}$, and each root processor $RP_j^r$ and $RP_j^c$ has a local variable $EXT_j^r$ and $EXT_j^c$, respectively.

**procedure** *CHECKFACE_A*

*Input*: Faces of $CH(A)$ and $CH(B)$ stored in the $LP$'s in row 0 of $MOT(log_2 N/2, 0)$ and $MOT(log_2 N/2, 1)$, respectively.

*Output* : The faces remain where they were, but the *extn* field of a face of $CH(A)$ is set 1 if it is external, 0 otherwise.

**begin**

**for** all $LP_{i,j}$ **do in parallel**

$EXT_{i,j} := 1$;

**endfor**

Use procedure *MULTI_CAST* to distribute all faces stored in row 0 of the *MOT* to all $LP$'s such that each $LP_{i,j}$, $i \neq j$, contains faces $F_{0,i,1}$, $F_{0,i,2}$, $F_{0,j,1}$, and $F_{0,j,2}$;

**for** all $LP_{i,j}$ **do in parallel**

**if** $i < N/2$ and $j > N/2$ and $i \neq j$ **then**

**if** $CH(A)$, $F_{0,j,1}$ and $F_{0,j,2}$ are not all on the same side of plane $P_{0,i,1}$ **then**

$EXT_{i,j} := 0$;

**if** $j < N/2$ and $i > N/2$ and $i \neq j$ **then**

**if** $CH(A)$, $F_{0,j,1}$ and $F_{0,j,2}$ are not all on the same side of plane $P_{0,i,2}$ **then**

$EXT_{i,j} := 0$;

**endfor**

**for** $0 \leq j \leq N/2\text{-}1$ **do in parallel**

Perform an *AND* operation on the $EXT_{i,j}$'s in row $j$ using the row tree and store the result, which is denoted by $EXT^r_j$, in $RP^r_j$;

Route $EXT^r_j$ to $LP_{0,j}$ using a pair of *RTL_R* and *LTL_C* operations, set $extn_{0,j,1} := EXT^r_j$;

**endfor**

**for** $0 \leq j \leq N/2\text{-}1$ **do in parallel**

Perform an *AND* operation on the $EXT_{i,j}$'s in column $j$ using the column tree and store the result, which is denoted by $EXT^c_j$, in $RP^c_j$;

Route $EXT^c_j$ to $LP_{0,j}$ using a *RTL_C* operation, set $extn_{0,j,2} := EXT^c_j$;

**endfor**

**end** *CHECKFACE_A*

After the *MULT_CAST* operation, only those $LP_{i,j}$'s such that $i < N/2$ and $j > N/2$, or $i > N/2$ and $j < N/2$ contains faces of $CH(A)$ and $CH(B)$. Each of such $LP$'s contains exactly four faces (including dummy ones), two from each of $CH(A)$ and $CH(B)$. Furthermore, $LP_{i,j}$ and $LP_{j,i}$ receive the same set of faces (see Figure 3.4).

For the sake of load balancing, we let each of these *LP*'s compare one face from $CH(A)$ with two faces from $CH(B)$. The time complexity of this procedure is $O(logN)$, because all of its parallel operations take no more than $O(logN)$ time.

#### 3.3.3.2 Identifying Critical External Faces

As mentioned in the beginning of section 3.3.3, each of the critical external faces of $CH(A)$ (resp. $CH(B)$) shares exactly one edge with an internal face of $CH(A)$ (resp. $CH(B)$), and these shared edges form a circuit $C_A$ (resp. $C_B$). To generate new faces on $CH(A \cup B)$, we need to locate these critical external faces and the shared edges. The following procedure is used to find all critical external faces on $CH(A)$, and for each of them identify its edge shared by an internal face of $CH(A)$. The logical-and Boolean operator "•" is used. The procedure *CRITICAL_FACE_B* for identifying critical external faces of $CH(B)$ is similar.

**procedure** *CRITICAL_FACE_A*

*Input*: Faces of $CH(A)$ stored in the *LP*'s in row 0 of $MOT(log_2N/2, 0)$. Faces of $CH(A)$ have been classified as internal or external.

*Output*: The faces remain where they were, but the edges shared by the critical external and internal faces of $CH(A)$ are marked.

**begin**

Use procedure *MULTI_CAST* to distribute all faces stored in row 0 of $MOT(log_2N/2, 0)$ to all *LP*'s such that each $LP_{i,j}$, $i \neq j$, contains faces $F_{0,i,1}$, $F_{0,i,2}$, $F_{0,j,1}$, and $F_{0,j,2}$.

**for** all $LP_{i,j}$ **do in parallel**

**if** $i < j$ and $F_{0,i,1}$ and $F_{0,j,1}$ share an edge $e$ and $(extn_{0,i,1} \bullet extn_{0,j,1} = 0)$ **then**

**if** $extn_{0,i,1} = 0$ and $extn_{0,j,1} = 1$ **then** mark the edge $e$ of $F_{0,j,1}$

**else if** $extn_{0,i,1} = 1$ and $extn_{0,j,1} = 0$ **then** mark the edge $e$ of $F_{0,i,1}$;

**if** $i < j$ and $F_{0,i,1}$ and $F_{0,j,2}$ share an edge $e$ and $(extn_{0,i,1} \bullet extn_{0,j,2} = 0)$ **then**

**if** $extn_{0,i,1} = 0$ and $extn_{0,j,2} = 1$ **then** mark the edge $e$ of $F_{0,j,2}$

**else if** $extn_{0,i,1} = 1$ and $extn_{0,j,2} = 0$ **then** mark the edge $e$ of $F_{0,i,1}$;

**if** $i > j$ and $F_{0,i,2}$ and $F_{0,j,1}$ share an edge $e$ and $(extn_{0,i,2} \bullet extn_{0,j,1} = 0)$ **then**

**if** $extn_{0,i,2} = 0$ and $extn_{0,j,1} = 1$ **then** mark the edge $e$ of $F_{0,j,1}$

**else if** $extn_{0,i,2} = 1$ and $extn_{0,j,1} = 0$ **then** mark the edge $e$ of $F_{0,i,2}$;

**if** $i > j$ and $F_{0,i,2}$ and $F_{0,j,2}$ share an edge $e$ and $(extn_{0,i,2} \bullet extn_{0,j,2} = 0)$ **then**

**if** $extn_{0,i,2} = 0$ and $extn_{0,j,2} = 1$ **then** mark the edge $e$ of $F_{0,j,2}$

**else if** $extn_{0,i,2} = 1$ and $extn_{0,j,2} = 0$ **then** mark the edge $e$ of $F_{0,i,2}$;

**endfor**

**for** all $LP_{0,j}$, $0 \le j \le N/2\text{-}1$, **do in parallel**

**begin**

**if** $F_{0,j,1}$ and $F_{0,j,2}$ share an edge $e$ and $extn_{0,j,1} = 0$ and $extn_{0,j,2} = 1$ **then** mark $e$ of $F_{0,j,2}$;

**if** $F_{0,j,1}$ and $F_{0,j,2}$ share an edge $e$ and $extn_{0,j,1} = 1$ and $extn_{0,j,2} = 0$ **then** mark $e$ on $F_{0,j,1}$;

**end**

**endfor**

Use procedure *MULTI_COLLECT* to collect the status of external faces of $CH(A)$ in all $LP$'s and update the faces in $LP$'s the first row, row 0, of $MOT(\lceil log_2 N/2 \rceil, 0)$;

**end** *CRITICAL_FACE_A*

In procedure *CRITICAL_FACE_A*, all faces of $CH(A)$ are distributed to all *LP*'s. Then, faces of $CH(A)$ are compared with each other, and all critical external faces are marked. Finally, all faces of $CH(A)$ are sent back to where they were before the procedure *MULTI_CAST* was executed. The marked faces replace their original copies in $LP_{0,j}$'s. Before we present procedure *MULTI_COLLECT*, we define a new operation on the mesh of trees.

*COLLECT_R*($i$) : collect items in a row. Collect a set of data items stored in *LP*'s in row $i$, and put all collected items in $RP_i^r$, the root of row $i$.

We similarly define an operation *COLLECT_C*($i$), to collect items in column $i$. Due to the bounded interconnection degree of binary tree structure, the performance of *COLLECT* depends on the number of data items to be collected and the size of each data item. If the number of data items is a small constant, and the size of each item is bounded by a small constant, then *COLLECT_R*($i$) requires $O(logN)$ time. In our case, since each critical external face of $CH(A)$ can share exactly one internal face of $CH(A)$, the number of data items to be collected in a row or column is at most two, and each item is a record for a face, with contains a fixed amount of information.

**procedure** *MULTI_COLLECT*

**begin**

**for** $0 \leq j \leq N-1$ **do in parallel**

Use *COLLECT_C* to collect faces with marked edges in all *LP*'s in column $j$ and store the collected faces in $RP_j^c$;

Use *COLLECT_R* to collect faces with marked edges in all *LP*'s in row $j$ and store the collected faces in $RP_j^r$;

Use *RTL_R* to send collected faces in $RP_i^r$ to $LP_{0,j}$;

Use *RTL_C* to send collected faces in $RP_i^c$ to $LP_{0,j}$;

**endfor**

**for** $LP_{0,j}$ **do in parallel**

Replace the original $F_{0,j,1}$ and/or $F_{0,j,2}$ by ones with marked edges (if any);

**endfor**

**end** *MULTI_COLLECT*

Since *MULTI_COLLECT* takes $O(logN)$ time, the time complexity of procedure *CRITICAL_FACE_A* is $O(logN)$.

### 3.3.3.3 Generating New Faces

The marked edges of critical external faces of $CH(A)$ (resp. $CH(B)$) form a circuit $C_A$ (resp. $C_B$), as shown in Figure 3.5. The new faces of $CH(A \cup B)$ are generated along these two circuits. For a critical external face $F$ of $CH(A)$ with marked edge $e = (p_a, p_b)$, where $p_a = (x_a, y_a, z_a)$, $p_b = (x_b, y_b, z_b) \in S'$, we define its *representative point* of $F$ as follows: the representative point of $F$ is $p_a$, if $\cos^{-1}(y_a/(y_a^2 + z_a^2)^{1/2}) > \cos^{-1}(y_b/(y_b^2 + z_b^2)^{1/2})$; $p_b$, otherwise. Note that $\cos^{-1}(y/(y^2 + z^2)^{1/2})$ is the angle between $op'$ and the $y$-axis, where $p'$ is the projection of $p = (x, y, z)$ in the $yz$-plane and $o$ is the origin. Clearly, the set of representative points of all critical external faces of $CH(A)$ is exactly the vertex set of circuit $C_A$. The representative points of critical external faces of $CH(B)$ are defined similarly.

Let $C'_A$ and $C'_B$ be the circuits in the $yz$-plane obtained by projecting $C_A$ and $C_B$ to the $yz$-plane, respectively. Let $o'_A$ and $o'_B$ be two points in the $yz$-plane such that they are internal to $C'_A$ and $C'_B$, respectively. Clearly, $o'_A$ (resp. $o'_B$) can be found in

$O(logN)$ time by finding a point in the triangle formed by any three non-collinear points of $C'_A$ (resp. $C'_B$). We introduce two polar coordinate systems $\Sigma_A$ and $\Sigma_B$: $\Sigma_A$ has origin $o'_A$ and polar axis in the direction of the $y$-coordinate, and $\Sigma_B$ has origin $o'_B$ and polar axis in the opposite direction of the $y$-coordinate. Let $\gamma_A(p)$ and $\gamma_B(p)$ be the polar angle of the image of point $p$ in $\Sigma_A$ and $\Sigma_B$, respectively. The vertices of $C_A$ (resp. $C_B$) are uniquely ordered by the non-decreasing order of their images' polar angles $\gamma_A$ in $\Sigma_A$ (resp. $\Sigma_B$). These two coordinate systems and the orders of vertices in $C'_A$ and $C'_B$ are shown in Figure 3.6. These orders are important to determine the new faces of $CH(A \cup B)$.

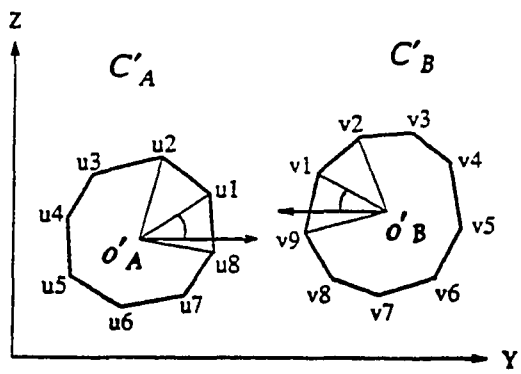


Fig. 3.6 Polar coordinate systems $\Sigma_A$ and $\Sigma_B$. $u_i$'s and $v_j$'s are projections of vertices on $C'_A$ and $C'_B$, respectively.

For each critical external face $F_{Ai}$ of $CH(A)$ with a marked edge $e_{Ai}$ in circuit $C_A$, a new triangle face $F(e_{Ai}, p_{Bj})$ of $CH(A \cup B)$ bounded by $e_{Ai}$ and a vertex $p_{Bj}$ of circuit $C_B$ can be identified. Let $P(e_{Ai}, p)$ denote the plane that contains edge $e_{Ai}$ of $C_A$ and vertex $p$ of $C_B$. Define angle $\theta(F_{Ai}, p)$ associated with critical external face $F_{Ai}$ of $CH(A)$ and a vertex point $p$ of $C_B$ as the convex angle formed by the plane $P(e_{Ai}, p)$ and face $F_{Ai}$. This angle can be computed using the normal vectors of $P(e_{Ai}, p)$ and $F_{Ai}$.

Let $\xi(F_{Ai}) = \max \{ \theta(F_{Ai}, p_j) \mid p_j$ is a vertex of $C_B \}$, $V(F_{Ai}) = \{p \mid p$ is a vertex of $C_B$ and $\theta(F_{Ai}, p) = \xi(F_{Ai})\}$, and $p(F_{Ai})$ be the point $p$ in $V(F_{Ai})$ with the smallest polar angle $\gamma_B(p)$. For each critical external face $F_{Ai}$, we can uniquely define a face $F(e_{Ai}, p(F_{Ai}))$. Let $F_A = \{F(e_{Ai}, p(F_{Ai})) \mid F_{Ai}$ is a critical external face of $CH(A)\}$, be the set of new faces of $CH(A \cup B)$ that share edges of critical external faces of $CH(A)$. Similarly, we can define $P(e_{Bi}, p)$, $\theta(F_{Bi}, p)$, $V(F_{Bi})$ and $p(F_{Bi})$ for each critical external face $F_{Bi}$ of $CH(B)$ by interchanging "$A$" and "$B$" in their corresponding definitions for critical external faces $F_{Ai}$ of $CH(A)$. Then, we can obtain a set $F_B$ of external faces of $CH(A \cup B)$. It is easy to verify that the faces in $F_A \cup F_B$ form a set of new faces of $CH(A \cup B)$. We are ready to present the procedure for generating the new faces of $CH(A \cup B)$. The following procedure generates new faces which share marked edges of critical external faces of $CH(A)$.

**procedure** *NEW_FACE_A*

*Input* : Faces of $CH(A)$ and $CH(B)$ stored in the $LP$'s of row 0 of $MOT(\log_2 N/2, 0)$ and $MOT(\log_2 N/2, 1)$, respectively, with edges of the critical external faces shared by the internal faces marked.

*Output* : Original faces of $CH(A)$ and $CH(B)$ remain where they were, and new faces of $CH(A \cup B)$ sharing edges with external faces of $CH(A)$ are generated and stored in $LP_{0,0}, \ldots, LP_{0,N/2-1}$.

**begin**

Find $o'_B$ for the marked edges stored in row 0 of $MOT(\log_2 N/2, 1)$.

**for** $j \geq N/2$ **do in parallel**

Compute $\gamma_B(p)$ for the representative point $p$ of each of the critical external face of $CH(B)$ stored in row 0 of $MOT(\log_2 N/2, 1)$, using coordinate system $\Sigma_B$;

**endfor**

Use procedure *MULTI_CAST* to distribute all faces stored in row 0 of the mesh of trees to all *LP*'s such that each $LP_{i,j}$, $i \neq j$, contains faces $F_{0,i,1}$, $F_{0,i,2}$, $F_{0,j,1}$, and $F_{0,j,2}$;

**for** all $LP_{i,j}$ **do in parallel**

**if** $i < N/2$ and $j \geq N/2$ **then**

**begin**

**if** $F_{0,i,1}$ and $F_{0,j,1}$ are both critical external faces **then**

Use the marked edge $e_{0,i,1}$ of $F_{0,i,1}$ and the representative point $p_{0,j,1}$ of $F_{0,j,1}$ to compute $\theta(e_{0,i,1}, p_{0,j,1})$ and create a triangular face $F'_{i,j}(1) = F(e_{0,i,1}, p_{0,j,1})$;

**if** $F_{0,i,1}$ and $F_{0,j,2}$ are both critical external faces **then**

Use the marked edge $e_{0,i,1}$ of $F_{0,i,1}$ and the representative point $p_{0,j,2}$ of $F_{0,j,2}$ to compute $\theta(e_{0,i,1}, p_{0,j,2})$ and create a triangular face $F'_{i,j}(2) = F(e_{0,i,1}, p_{0,j,2})$;

**end**

**if** $j < N/2$ and $i \geq N/2$ **then**

**begin**

**if** $F_{0,i,2}$ and $F_{0,j,1}$ are both critical external faces **then**

Use the marked edge $e_{0,i,2}$ of $F_{0,i,1}$ and the representative point $p_{0,j,1}$ of $F_{0,j,1}$ to compute $\theta(e_{0,i,2}, p_{0,j,1})$ and create a triangular face $F''_{i,j}(1) = F(e_{0,i,2}, p_{0,j,1})$;

**if** $F_{0,i,2}$ and $F_{0,j,2}$ are both critical external faces **then**

Use the marked edge $e_{0,i,2}$ of $F_{0,i,2}$ and the representative point $p_{0,j,2}$ of

$F_{0,j,2}$ to compute $\theta(e_{0,i,2}, p_{0,j,1})$ and create a triangular face $F''_{i,j}(2) = F(e_{0,i,2}, p_{0,j,2})$;

**end**

**endfor**

**for** $0 \le j < N$ **do in parallel**

Use *MAX_R* operation on row $j$ to find the face $F'$ whose $\theta$ value is the maximum among all newly created faces $F'$ (if exists), and if there is more than one such face, find the one with the largest $\gamma_B$ value; put this face in $RP^r_j$;

Use *RTL_R* and *LTL_C* to route this face to $LP_{0,j}$ and set its *extn* to 1;

Use *MAX_C* operation on column $j$ to find the face whose $\theta$ value is the maximum among all newly created faces $F''$ (if exists), and if there is more than one such face, find the one with the largest $\gamma_B$ value; put this face in $RP^c_j$;

Use *RTL_C* and *LTL_R* to route this face to $LP_{0,j}$ and set its *extn* to 1;

**endfor**

**end** *NEW_FACE_A*

The time complexity of procedure *NEW_FACE_A* is obviously $O(logN)$. Procedure *NEW_FACE_B* is similar to *NEW_FACE_A*.

### 3.3.3.4 Load Balancing Operations

After creating new faces, we have to remove internal faces in $CH(A)$ and $CH(B)$ by the procedure *NEW_FACE*. We know that each $LP_{0,j}$ in row 0 contains at most four faces: at most two original faces from either $CH(A)$ or $CH(B)$, and at most two

new faces of $CH(A \cup B)$. However, we know that there are at most $2 \times (|A|+|B|)-4$ faces in $CH(A \cup B)$. The objective of the rebalancing operations is to enforce each processor $LP_{0,j}$ to contain at most two faces of $CH(A \cup B)$. This is the invariant required for the recursive algorithm *CONVEX_HULL* to operate correctly and efficiently. The technique used is the classical parallel packing method.

There are basically two types of packing problems, *down-packing* and *up-packing*. The down-packing problem is defined as follows. Consider a linear array such that each of its entry contains either one or no data item. The task of down-packing is to move data items from entries of high indices to entries of low indices in such a way that all data items are in consecutive entries, each of these low-indexed entries contains one item, and the relative order of the items is unchanged. The up-packing is similar to the down-packing; except for the direction of data movement. Consider how to perform down-packing on the *LP*'s in row 0 of the mesh of trees. Let each $LP_{0,j}$ have a flag $f_{0,j}$, which has value 1 if it has a data item to be packed, or 0, otherwise. The following procedure is used to compute the prefix sums of the $f_{0,j}$'s for all $LP_{0,j}$'s.

**procedure** *PREFIX*

**begin**

**for** $0 \le j \le N\text{-}1$ **do in parallel**

Use *LTR* to send $f_{0,j}$ from $LP_{0,j}$ to the root $RP_j^c$ of column $j$;

Use *BROADCAST_C* to broadcast $f_{0,j}$ to all *LP*'s in column $j$;

Perform *SUM_R* to compute the partial sum $prefix_{0,j} = \sum_{k=0}^{j} f_{0,k}$'s in row $j$, and

store $prefix_{0,j}$ in $RP_j^r$, the root of row $j$;

Use *RTL_R* and *LTL_C* operations to route $prefix_{0,j}$ from $RP_j^r$ to $LP_{0,j}$;

**endfor**

**end** *PREFIX*

With procedure *PREFIX* at disposal, the down-packing algorithm is simple.

**procedure** *DOWN_PACK*

**begin**

Use *PREFIX* to compute $prefix_{0,j}$ for all $LP_{0,j}$'s, the *LP*'s in row 0;

**for** all $LP_{0,j}$ **do in parallel**

$dest_{0,j} := prefix_{0,j} - 1$;

**endfor**

**for** $0 \le j \le N\text{-}1$ **do in parallel**

Use *LTR*, *RTL*, *LTR* and *RTL* operations to route $data_{0,j}$ from $LP_{0,j}$ to $RP_j^c$, then to $LP_{j,dest_{0,j}}$, then to $RP_{dest_{0,j}}$, then to $LP_{0,dest_{0,j}}$;

**endfor**

**end** *DOWN_PACK*

Similarly, we can define procedures *POSTFIX* and *UP_PACK*. Clearly, procedures *DOWN_PACK* and *UP_PACK* require $O(logN)$ time. The rebalancing algorithm, which has time complexity $O(logN)$, is as follows.

**procedure** *REBALANCE*

*Input* : Faces of $CH(A)$, $CH(B)$ and new faces of $CH(A \cup B)$ stored in the *LP*'s in row 0 of the *MOT* and each of these *LP*'s contains at most four faces.

*Output* : Faces of $CH(A \cup B)$ are stored in the *LP*'s in row 0 of the *MOT* and each of these *LP*'s contains at most two faces.

**begin**

**for** all $LP_{0,j}$, $0 \le j \le N\text{-}1$ **do in parallel**

Remove all faces in $LP_{0,j}$ whose *extn* is equal to 0;

**if** there are $k$ faces in $LP_{0,j}$ whose *extn*'s are equal to 1 **then**

Label them $f_1, ..., f_k$, respectively (* $k \le 4$ *);

**endfor**

Use *SUM_R* operations on *LP*'s in row 0 to compute the numbers $N_l$, $1 \le l \le 4$, where $N_l$ is the total number of faces labeled $f_l$ in all *LP*'s of row 0 (* Note: $N_1 \ge N_2 \ge N_3 \ge N_4$, and $\sum_{l=1}^{4} N_l \le 2N$ -4 *);

Use *DOWN_PACK* to pack all faces labeled $f_1$ in all $LP_{0,j}$'s;

Use *UP_PACK* to pack all faces labeled $f_2$ in all $LP_{0,j}$;

Use *DOWN_PACK* to pack all faces labeled $f_3$ in all $LP_{0,j}$'s;

Use *UP_PACK* to pack all faces labeled $f_4$ in all $LP_{0,j}$'s;

**if** $N_1 + N_2 > N$ **then**

**begin**

Use a modified *DOWN_PACK* procedure to pack all faces labeled $f_4$ in all $LP_{0,j}$'s to consecutive $LP_{0,j}$'s starting from $LP_{0,N_3}$;

Use *UP_PACK* to pack all faces labeled $f_2$ in all $LP_{0,j}$'s that contains a face labeled $f_1$ and a face labeled $f_2$;

**end**

**end** *REBALANCE*

If $N_1 + N_2 > N$, then after four packing operations, some $LP_{0,j}$'s may still contain more than two faces. Using a procedure similar to *DOWN_PACK*, we can move all faces labeled $f_4$ "downward" so that each of the first $N_3$ $LP_{0,j}$'s contains exactly one face labled $f_3$, and the next $N_4$ $LP_{0,j}$'s contain exactly one face labeled $f_4$. Then, by up-packing the faces labeled $f_2$ in those $LP_{0,j}$'s that contain a $f_1$-labeled face and a $f_2$-labeled face all the way "upward", each $LP_{0,j}$ is guaranteed to contain no more than two external faces of $CH(A \cup B)$.

#### 3.3.3.5 Analysis of Time Complexity

Based on the fact that each of the procedures *CHECKFACE*, *CRITICAL_FACE*, *NEW_FACE* and *REBALANCE*, takes $O(logN)$ time on the $N \times N$ mesh of trees, the complexity of *MERGE* is also $O(logN)$. Since there are log $N$ merging levels, we have the following results.

**Theorem 1**: For any set $S$ of $N$ points in $E^3$, with no more than three points in the same plane, the convex hull $CH(S)$ can be computed on the $N \times N$ mesh of trees in $O(log^2 N)$ time.

By lemma 1, we have

**Theorem 2**: For a set $S$ of $N$ points in $E^2$, the Voronoi diagram of $S$ can be computed on the $N \times N$ mesh of trees in $O(log^2 N)$ time. More specifically, the algorithm *VORONOI* of section 3.2.3 can be implemented on the $N \times N$ mesh of trees in $O(log^2 N)$ time.

### 3.3.3.6 Generalizations of the Convex Hull Algorithm

The assumption in Theorem 1 that there are no more than three points of $S$ in the same plane is important. If there are more than three points of $S$ in the same plane, triangularity of faces of $CH(S)$ may no longer hold. For any convex hull $CH(S)$ of any point set $S$ in $E^3$, one can always triangulate each face of $CH(S)$ into triangles to obtain a convex hull such that each of its face is a triangle. We call such a convex hull as a *triangulated convex hull* of $S$. By slightly modifying the second step of algorithm *CONVEX_HULL*, i.e. the step of computing convex hulls of four points in all $2 \times 2$ sub-*MOT*, a triangulated convex hull can be constructed. Then, without changing the rest of the algorithm *CONVEX_HULL*, a triangulated convex hull of any point set $S$ can be computed. Hence, we have a collorary of theorem 1.

**Corollary 1**: For any set $S$ of $N$ points in $E^3$, the triangulated convex hull of $S$ can be computed on the $N \times N$ mesh of trees in $O(log^2 N)$ time.

If in the process of procedure *MERGE*, associated with each triangle face $F_{0,j,k}$, $k = 1, 2$, we always keep its three adjacent faces, then after the triangulated convex hull is obtained, for all processor $L_{0,j}$ we perform a parallel step to mark the edges of $F_{0,j,k}$ shared by its adjacent face that are in the same plane containing $F_{0,j,k}$, and use a pair of *MULTI_CAST* and *MULTI_COLLECT* operations to eliminate all marked edges, then we obtain the set of all edges that bound the faces of $CH(S)$. Therefore, we have the following generalization of theorem 1.

**Theorem 3:** The convex hull of any set $S$ of $N$ points in $E^3$ can be computed on the $N \times N$ mesh of trees in $O(log^2 N)$ time.

### 3.4 Discussions

We presented efficient algorithms for constructing the Voronoi diagram of a set of $N$ points in $E^2$ and constructing the convex hull of a set of $N$ points in $E^3$. Both algorithms require $O(log^2 N)$ time. Our algorithms are the fastest $NC$ algorithms ever known for constructing planar Voronoi diagrams and three-dimensional convex hulls. Our Voronoi diagram algorithm can easily be converted into an $O(log^2 N)$-time algorithm for constructing the Delaunay triangulation of a set $N$ planar points.

The Voronoi diagram considered in this chapter is commonly called planar nearest point Voronoi diagram. There are many variations of the Voronoi diagram. For example, the planar farthest point Voronoi diagram of a set $S$ of points in the plane is the polygonal division of the plane, each region $V_i$ is the locus of points that is farther from $p_i$ than from other points in $S$. Based on the sequential algorithm given in [B79] and our convex hull algorithm, the planar farthest point Voronoi diagram of $N$ points can also be constructed in $O(log^2 N)$ time on the $N \times N$ mesh of trees.

Finding efficient parallel algorithms for constructing Voronoi diagrams of higher dimensions may be interesting. It remains open whether or not the general method, proposed in [B79], of reducing the problem of constructing the $k$-dimensional Voronoi diagram to the problem of constructing $(k+1)$-dimensional convex hull is suitable for efficient parallelization.

# Chapter 4

# The Congruent Patterns

## 4.1 Introduction

Two planar patterns $P_1$ and $P_2$ are congruent if their corresponding boundary edges (e.g. edge $e_i$ in $P_1$ and edge $e_j$ in $P_2$) are of equal length and the corresponding interior angles (e.g. interior angle $\theta_i$ in $P_1$ and interior angle $\theta_j$ in $P_2$) between any pair of adjacent boundary edges are also equal. Finding congruent patterns arises frequently in computer-aided design (CAD), image analysis, pattern recognition, and computer vision. Given a test pattern P which is a simple polygon with $k$ edges, and a planar straight-line graph (PSLG) G with $n$ edges, where $k \leq n$, the *congruent pattern problem* is to locate all the cycles in G which are congruent to P.

A *test pattern* P is a simple polygon which is used as a prototype for the congruence problem. Because P is a simple polygon, it has the same number of vertices and edges. We denote the test pattern as P = $\{V_p, E_p\}$, where the vertex set $V_p$ = $\{v_{p,1}, v_{p,2}, v_{p,3}, \ldots, v_{p,k}\}$ and the edge set $E_p = \{e_{p,1}, e_{p,2}, e_{p,3}, \ldots, e_{p,k}\}$. Here, $k$ is the number of vertices and edges of P, respectively. We assign the indices of $V_p$ and $E_p$ in clockwise order and denote $e_{p,i}$ as the edge between $v_{p,i}$ and $v_{p,i+1}$. If $i = k$ then set $i + 1 = 1$. We also denote the *interior angle* at vertex $v_{p,i}$ as $\theta_{p,i}$, and the set of interior angles as $\Theta_p$.

A *planar straight-line graph* (PSLG) is a planar graph whose edges are all straight-line segments. We denote the PSLG as G = $\{V_g, E_g\}$, where the vertex set $V_g$

$= \{v_{g,1}, v_{g,2}, v_{g,3}, \ldots, v_{g,m}\}$ and the edge set $E_g = \{e_{g,1}, e_{g,2}, ..., e_{g,n}\}$. Here, $m$ and $n$ are the number of vertices and edges of G, respectively. According to the classical *Euler's formula*, we know that $m \leq n$. A test pattern and a PSLG are shown in Figure 4.1.

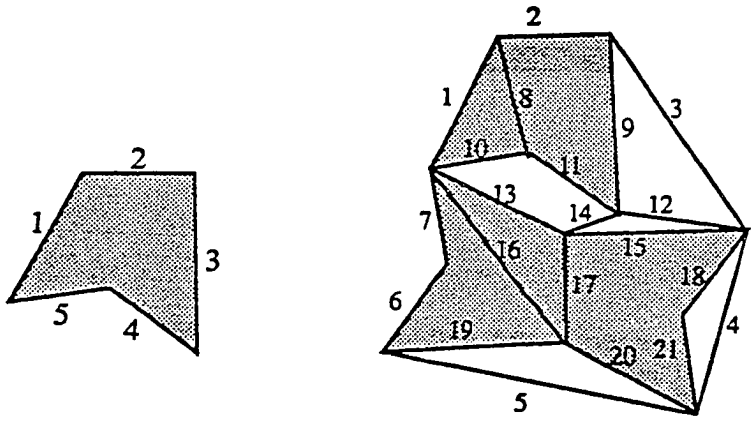


Fig. 4.1 A test pattern P and a PSLG with 3 patterns congruent to P.

There are some previous papers that discuss the congruence problem. In 1984, Sugihara described a sequential algorithm for determining whether two polyhedra are congruent or not. The asymptotic time complexity of this algorithm is bounded by $O(n \log n)$, where $n$ is the number of edges of the polyhedra [S84]. In 1987, Atkinson developed a sequential algorithm for determining all the Euclidean congruences between two $n$-point sets in 3-dimensional space. The time complexity of this algorithm is $O(n \log n)$ which is optimal [A87]. In 1990, Shih, Lee and Yang developed an $O(n^2)$ parallel algorithm for finding the congruent patterns P in a PSLG on shared memory model with $k$ PEs and $O(kn)$ memory, where $n$ and $k$ are the number of edges in $G$ and $P$ respectively [SLY90]. In 1992, Jeong, Kim and Baek presented another parallel algorithm to solve the same problem on SIMD machine. Their algorithm takes $O(\log n \log \log n)$ time if each edge in the test pattern has unique length; otherwise in $O(k \log n \log \log n)$ time, where $k$ is the number of patterns [JKB92].

In this chapter, we present an $O(k \log n)$-time parallel algorithm for reporting all the patterns in G which are congruent to P on an $n \times n$ mesh of trees. The structure of mesh of trees and all the operations of data communications in this chapter, such as $RTL\_R(i,j)$, $LTR\_R(i,j)$, $LTL\_R(i,j,k)$, $AND\_R(i)$, and $BROADCAST\_R(i)$, have been defined in section 3.2.2. Our algorithm determines the congruent patterns by considering all cases of whether edges in G or P are distinct or not. On the other hand, if G and P are polyhydra and are projected on a two dimensional plane then our algorithm with slight modification, may also be able to report the congruent patterns in G and its time complexity is still $O(log\ n)$. This chapter is organized as follows. In section 4.2, we present a mesh of trees algorithm to solve the congruent pattern problem. We also analyze the time complexity for each subalgorithm and the main algorithm. Finally, we present conclusion in section 4.3.

## 4.2 Finding Congruent Patterns on a Mesh of Trees

Since our test pattern P is a simple polygon, to find a pattern in G congruent to P is just to find a *cycle* which congruent to P. A *cycle* in a graph is a *path* in which the origin and destination vertices are the same. There are three key points we have to check for finding congruent cycle (denoted as C, i.e. a pattern) in G. (1) Each edge in C should be equal to its corresponding edge in P. (2) Each interior angle between any two adjacent edges in C should be equal to its corresponding interior angle in P. (3) The turning direction between any two adjacent edges in C should be equal to its corresponding turning direction in P. Besides, different edge length between P and G will generate different amount of congruent patterns. For this property, we need to consider the following four cases :

*Case* 1 : P has some equal-length edges, but G has distinct edges;

*Case* 2 : P and G have distinct edges, respectively;

*Case* 3 : G has some equal-length edges, but P has distinct edges;

*Case* 4 : P and G have some equal-length edges, respectively.

**Lemma 4.1** : If the test pattern P has equal-length edges, but G has no equal-length edges, then there is no any pattern in G congruent to P.

**Proof** : Assume there is at least one pattern in G congruent to P, then the congruent pattern in G would have equal-length edges because there are equal-length edges in P. A contradiction. □

**Lemma 4.2** : If P and G have distinct edges respectively, then there is zero or only one pattern in G congruent to P.

**Proof** : Assume that more than one congruent patterns can be found in G, then each pair of corresponding edges in these patterns is equal. Therefore, G will have some edges with equal length. This is a contradiction. So we conclude that there is zero or only one congruent pattern can be found in G. □

**Lemma 4.3** : If G has equal-length edges, but P has distinct edges, then there are zero, one, or more patterns in G congruent to P.

**Proof** : Assume there is no more than one congruent pattern in G, then according to Lemma 4.2, we know that there are no equal-length edges in G. A contradiction. Since it is possible to form more congruent patterns by using those equal-length edges as the common edges between any two adjacent or overlapping congruent patterns. □

**Lemma 4.4** : If G has equal-length edges, and P also has equal-length edges, then there are zero, one, or more patterns in G congruent to P.

**Proof :** The proof is same as Lemma 4.3, because the congruent pattern number in G is independent of the edge-length equality in $E_p$ of P. □

In this section, we first develop two procedures to perform the initial operations on P and G, respectively. Then, we develop two other procedures to check the length equality in $E_p$ and $E_g$, respectively. In section 4.2.3, we discuss *Case* 1 and *Case* 2, and develop a procedure to find single pattern. In section 4.2.4, we discuss *Case* 3 and *Case* 4 , and develop another procedure to find multiple patterns. Finally, by calling the procedures in sections 4.2.1 to 4.2.4, we develop the main algorithm to find all congruent patterns in section 4.2.5.

### 4.2.1 Preprocessing

In order to modularize our algorithm, we have to do some initial operations. For test pattern P, first, we broadcast $e_{p,1}$ (the two end-points $v_{p,1}$ and $v_{p,2}$ of the starting edge) to all *LP*'s of the $k \times n$ MOT, and compute its length. Next, we broadcast $e_{p,i}$ (its two end-points, $v_{p,i}$ and $v_{p,i+1}$) to all *LP*'s in row $i$, for $2 \le i \le k$. Finally, we compute the *relative route* of $e_{p,i}$, denoted as $RR_{p,i}$, in each *LP* from row 2 to row $k$.

The relative route of $e_{p,i}$ contains $e_{p,1}$ and its length, $e_{p,i}$ and its length, the *bridge* between $e_{p,1}$ and $e_{p,i}$ , denoted as $b_{p,i}$, the interior angle between $e_{p,1}$ and bridge, denoted as $\alpha_{p,i}$, and the interior angle between bridge and $e_{p,i}$, denoted as $\beta_{p,i}$. That is,

$$RR_{p,i} = (e_{p,1},\ \alpha_{p,i},\ b_{p,i},\ \beta_{p,i},\ e_{p,i}).$$

Note :

1. Bridge $b_{p,i}$ connects $v_{p,2}$ and $v_{p,i}$.
2. The turning directions of first-edge and bridge, and bridge and second-edge are all clockwise.
3. In $RR_{p,2}$, there is no bridge and $\alpha_{p,2} = \beta_{p,2}$ because $e_{p,1}$ and $e_{p,2}$ are connective edges.
4. $e_{p,2}$ is the bridge of $RR_{p,3}$.

A relative path of $e_{p,i}$ is shown in Figure 4.2. For the initial operations of P we develop a procedure as follows.

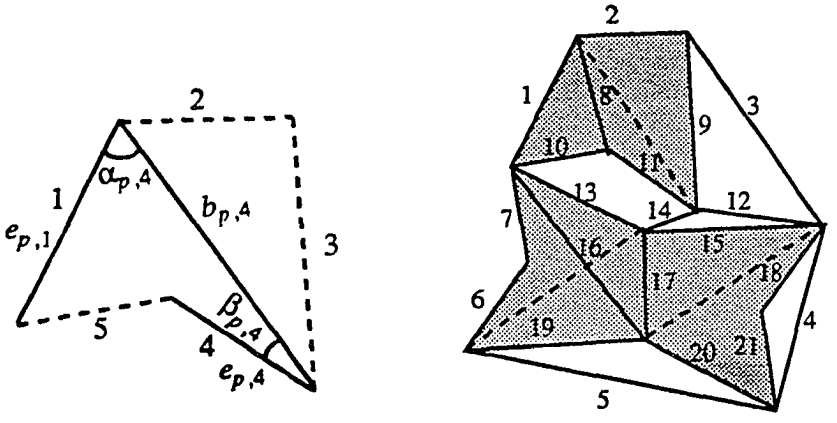


Fig. 4.2 Using $RR_{p,4}$ in P to compute $RR_{g,6}$, $RR_{g,11}$, and $RR_{g,18}$ in G.

**Procedure** *INITIALIZE_P*;

*Input* : The edge set $E_p$ with $k$ edges in P. Each edge is denoted by the coordinate values of its two end-points in the plane. These edges have been arranged in clockwise order previously.

*Output* : 1. Each $LP_{1,j}$ (in the first row) stores $e_{p,1}$ and its length.

2. From row 2 to row $k$, each $LP_{i,j}$ stores $RR_{p,i}$ (the relative route of $e_{p,i}$).

**begin**

(1) At first, assume the two end-vertices ($v_{p,1}$ and $v_{p,2}$) of $e_{p,1}$ are loaded to the root processor $RP_i^r$ in row $i$, for $1 \le i \le k$;

(2) **for** $1 \le i \le k$ **do in parallel**

Use *BROADCAST_R*($i$) to broadcast $v_{p,1}$ and $v_{p,2}$ in $RP_i^r$ to all *LP*'s in row $i$;

**endfor;**

(3) Then, assume the two end-vertices of $e_{p,i}$ ($v_{p,i}$ and $v_{p,i+1}$) are loaded to the root processor $RP_i^r$ in row $i$, for $2 \le i \le k$, if i=k then set i+1=1;

(4) **for** $2 \le i \le k$ **do in parallel**

Use *BROADCAST_R*($i$) to broadcast $v_{p,i}$ and $v_{p,i+1}$ in $RP_i^r$ to all *LP*'s in row $i$;

**endfor;**

(5) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

Each $LP_{i,j}$ computes the length of $e_{p,1}$ by its two end-points $v_{p,1}$ and $v_{p,2}$;

**endfor;**

(6) **for** $2 \le i \le k$ and $1 \le j \le n$ **do in parallel**

Each $LP_{i,j}$ computes the record $RR_{p,i}$ by the two end-points $v_{p,1}$ and $v_{p,2}$ and the two end-points $v_{p,i}$ and $v_{p,i+1}$;

**endfor;**

**end** of *INITIALIZE_P*;

Steps (2) and (4) need $O(log\ n)$ time, while other steps take constant time. Therefore, the time complexity for procedure *INITIALIZE_P* is $O(log\ n)$. After executing procedure *INITIALIZE_P*, each *LP* in row i ($1 \le i \le k$) stores the length value of $e_{p,1}$, and each *LP* in row j stores the record $RR_{p,j}$, for $2 \le j \le k$.

For the initial operations of planar graph G, first, we broadcast $e_{g,j}$ to all *LP*'s in column $j$, for $1 \leq j \leq n$, and each column has $k$ *LP*'s. Then compute the length of $e_{g,j}$ in each *LP*. The procedure is given below.

**Procedure** *INITIALIZE_G*;

*Input* : The edge set $E_g$ with $n$ edges in planar graph G. Each edge is denoted by the coordinate values of its two end-points in the plane.

*Output* : Each $LP_{i,j}$ (in the same column) stores the vertices $v_{g,j}$ and $v_{g,j+1}$ and the length of $e_{g,j}$ between them.

**begin**

(1) At first, assume each edge (say $e_{g,j}$) and its two end-points (e.g. $v_{g,h}$ and $v_{g,i}$) in G is randomly loaded to the root processor $RP_j^c$ in column $j$, for $1 \leq j \leq n$;

(2) **for** $1 \leq i, j \leq n$ **do in parallel**

Use *BROADCAST_C*($j$) to broadcast each vertex pair in $RP_j^c$ to all $LP_{i,j}$'s in column $j$;

**endfor**;

(3) **for** $1 \leq i, j \leq n$ **do in parallel**

Each $LP_{i,j}$ computes the length of edge $e_{g,j}$ with end-points $v_{g,h}$ and $v_{g,i}$;

**endfor**;

**end** of *INITIALIZE_G*;

In this procedure, step (2) takes $O(\log n)$ time, but other steps need only constant time. Hence, the time complexity for procedure *INITIALIZE_G* is $O(\log n)$. After P and G are initialized, the information stored in each *LP* are shown in Figure 4.3.

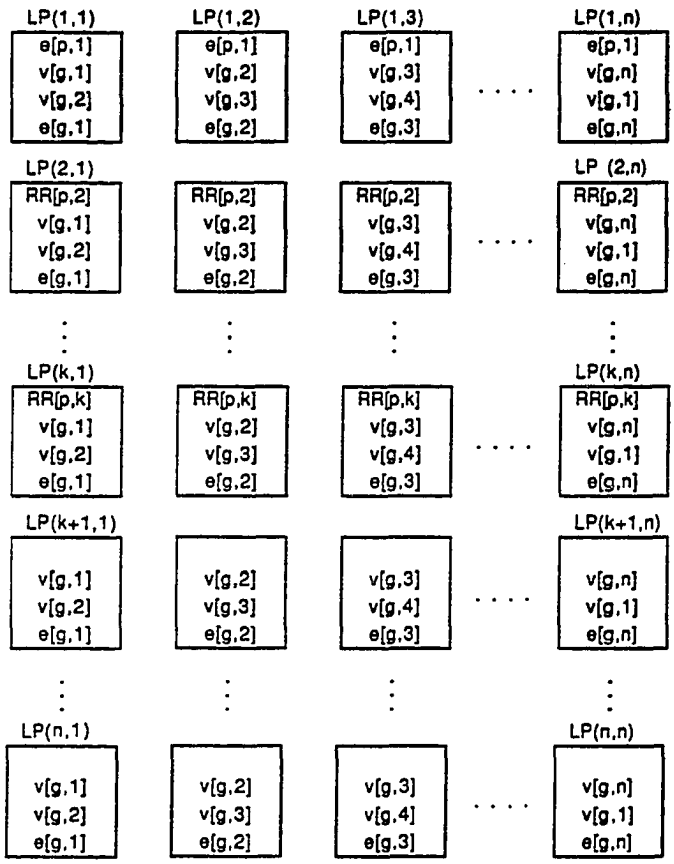


Fig. 4.3 Information stored in the $n \times n$ MOT after P and G are initialized.

### 4.2.2 Checking Edge-Length Equality of P and G

After invoking the initial operations on P and G, for the computing convenience of our main algorithm, we have to check the edge-length equality for the edge sets $E_p$ and $E_g$, respectively, and return the result to the main algorithm. The length of an edge is the *distance* between its two end-points. To check the edge-length equality of P and G, we assume that the initial operations of P and G have been done by procedures *INITIALIZE_P* and *INITIALIZE_G*, respectively. Then, for test pattern P, we develop the procedure *CHECK_P_LENGTH* to check whether there are equal-length

edges in P or not, and return the result (a TRUE/FALSE flag) to the root processor of column 1. Similarly, for planar graph G, we develop the procedure *CHECK_G_LENGTH* to check whether there are equal-length edges in G or not, and return the result to the root processor of row 1. Procedure *CHECK_P_LENGTH* is shown below.

**procedure** *CHECK_P_LENGTH*;

*Input* : The edge lengths of $e_{p,i}$ in $RP_{i,i}$, for $1 \le i \le k$, which are computed in procedure *INITIALIZE_P*;

*Output* : The final logic value of edge-length flag $EL_p$ in $RP_1^c$;

**begin**

(1) **for** $1 \le i \le k$ **do in parallel**

Use *BROADCAST_C*(*i*) to broadcast $e_{p,i}$ in $LP_{i,i}$ to all *LP*'s in column *i*;

**endfor**;

(2) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

(* Each $LP_{i,j}$, $i \neq j$, compares the length of $e_{p,i}$ and $e_{p,j}$. *)

**if** $|e_{p,i}| = |e_{p,j}|$ **then** set $EL_{i,j}$ = TRUE **else** set $EL_{i,j}$ = FALSE;

**endfor**;

(3) **for** $1 \le i \le k$ **do in parallel**

Use global *AND_R*(*i*) operation to do the logic AND function for each flag $EL_{i,j}$ in each *LP* of row *i*;

**endfor**; (* The ANDed result in each row is censused up to $RP_i^r$ in row *i*. *)

(4) **for** $1 \le i \le k$ **do in parallel**

Use *RTL_R*(*i*) operation to route the ANDed value of $EL_{i,j}$ in $RP_i^r$ to its left-most leaf processor $LP_{i,1}$;

**endfor;**

(* After this step, all ANDed $EL_{i,j}$ values are routed to all *LP*'s in column 1. *)

(5) Use global *AND_C(i)* operation to do the logic AND function for each flag $EL_{i,j}$ in each *LP* of column 1 and store the final value as $EL_p$ in $RP_1^c$;

(* The final ANDed value of $EL_p$ is censused up to $RP_1^c$ in column 1. *)

**end** of *CHECK_P_LENGTH*;

Step (2) in this procedure needs only constant time, while each of the other steps needs $O(log\ n)$ time. Thus, the time complexity for procedure *CHECK_P_LENGTH* is $O(log\ n)$. The procedure *CHECK_G_LENGTH* is shown below.

**procedure** *CHECK_G_LENGTH*;

*Input* : The edge length of $e_{g,j}$ in each $LP_{i,j}$ of the $n \times n$ MOT after executing procedure *INITIALIZE_G*;

*Output* : The final logic value of edge-length flag $EL_g$ in $RP_1^r$;

**begin**

(1) **for** $1 \le j \le n$ **do in parallel**

Use *BROADCAST_R(j)* to broadcast $e_{g,j}$ in $LP_{j,j}$ to all *LP*'s in row *j*;

**endfor;**

(2) **for** $1 \le i \le n$ and $1 \le j \le n$ **do in parallel**

(* Each $LP_{i,j}$, $i \neq j$, compares the length of $e_{g,i}$ and $e_{g,j}$. *)

**if** $|e_{g,i}| = |e_{g,j}|$ **then** set $EL_{i,j}$ = TRUE **else** set $EL_{i,j}$ = FALSE;

**endfor;**

(3) **for** $1 \le j \le n$ **do in parallel**

Use global *AND_C(j)* operation to do the logic AND function for each flag $EL_{i,j}$ in each *LP* of column *j*;

**endfor**;

(* Note : The ANDed result in each column is censused up to the root processor $RP_j^c$ in column *j*. *)

(4) **for** $1 \leq j \leq n$ **do in parallel**

Use *RTL_C(j)* operation to route the final ANDed value of $EL_{i,j}$ in $RP_j^c$ to its uppermost leaf processor $LP_{1,j}$;

**endfor**;

(* After this step, all the ANDed $EL_{i,j}$ values are routed to all *LP*'s in row 1. *)

(5) Use global *AND_R(i)* operation to do the logic AND function for each flag $EL_{i,j}$ in each *LP* of row 1 and store the final value as $EL_g$ in $RP_1^r$;

(* The final ANDed value of $EL_g$ is censused up to the root processor $RP_1^r$ in row 1. *)

**end** of *CHECK_G_LENGTH*;

Step (2) needs only constant time, but each of the other steps needs $O(\log n)$ time. So the time complexity of procedure *CHECK_G_LENGTH* is $O(\log n)$.

### 4.2.3 Finding Single Congruent Pattern

To find congruent patterns efficiently, we have to analyze the properties of length equality of P and G (i.e. *Case* 1 to *Case* 4). For *Case* 1, we know P has equal-length edges but all edges in G are distinct. Obviously, it is impossible to find any congruent patterns in G. Thus, we simply report the message "No congruent patterns found." In

*Case* 2, all edges in P and G are distinct, respectively. There can be at most one congruent pattern in G. For this case, we develop a procedure *SINGLE_PATTERN* to find the congruent pattern. We will discuss *Case* 3 and 4 in the next section.

**procedure** *SINGLE_PATTERN*;

*Input* : The outputs from procedures *INITIALIZE_P* and *INITIALIZE_G*;

*Output* : The congruent pattern in G for *Case* 2;

**begin**

(1) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**begin**

(* Each $LP_{i,j}$ compares the length of $e_{p,i}$ and $e_{g,j}$ which are computed in *INITIALIZE_P* and *INITIALIZE_G* respectively. *)

**if** $|e_{p,i}| = |e_{g,j}|$ **then** EQ = TRUE **else** EQ = FALSE;

**if** $|e_{p,1}| = |e_{g,j}|$ **then** set starting edge $e_{g,s} = e_{g,j}$ and set ss = j in $LP_{1,j}$;

(* EQ is the flag for indicating whether the two edges are equal or not. *)

(* starting edge $e_{g,s}$ is the first edge of a congruent pattern in G. *)

(* ss is the index of the starting edge of the congruent pattern in G. *)

**end;**

**endfor;**

(2) **for** $2 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**if** $LP_{i,j}$ with EQ = TRUE

**then** use the record $RR_{p,i}$ as the reference, to compute $RR_{g,j1}$ and $RR_{g,j2}$, and find the starting edges $e_{g,s1}$ and $e_{g,s2}$;

**endfor;**

(* Refer Figure 4.2 on page 76 and the comments at the end of this procedure. *)

(3) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

Use *LTR_R*$(i, j)$ and *BROADCAST_R*$(i)$ operations to broadcast the starting edge $e_{g,s}$ and index ss in $LP_{1,ss}$ (with EQ = TRUE), to all $LP_{i,j}$'s;

**endfor;**

(4) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**if** $LP_{i,j}$ with EQ = TRUE and $e_{g,s1} \neq e_{g,s}$ and $e_{g,s2} \neq e_{g,s}$

**then** set EQ = FALSE;

**endfor;**

(5) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**if** $LP_{i,j}$ with EQ = TRUE

**then** use *LTL* and *LTR* operations to send 1 and $e_{g,j}$ to $RP^c_{ss}$;

**endfor;**

(6) $RP^c_{ss}$ computes the sum $S$ of all 1's received from step (5);

(7) **if** $S = k$ **then** $RP^c_{ss}$ outputs all the edges, $e_{g,j}$, received from step (5);

(* These edges form a congruent pattern. *)

**else** print "No congruent pattern found.";

**end** of *SINGLE_PATTERN*;

(*** Comments of step (2) :

To compute $RR_{g,j}$, we have to consider the following two conditions. Assume $v_{g,j1}$ and $v_{g,j2}$ are the two end-points of $e_{g,j}$.

1. Use $v_{g,j1}$ as the common vertex of $e_{g,j}$ and $b_{g,j}$ to compute $RR_{g,j1}$ and find $e_{g,s1}$.

2. Use $v_{g,j2}$ as the common vertex of $e_{g,j}$ and $b_{g,j}$ to compute $RR_{g,j2}$ and find $e_{g,s2}$.

For computing the relative route $RR_{g,j}$, we also need to check the turning directions between $e_{g,s}$ and bridge $b_{g,j}$, and between $b_{g,j}$ and $e_{g,j}$, respectively. According to the property of $RR_{p,i}$, we know the above two turning directions should be all clockwise.

To check the turning direction between two connective edges, we have to compute the following determinant. If the value is negative then the turning direction is clockwise, otherwise the turning direction is counterclockwise.

For example, let $(x_{i-1}, y_{i-1})$ and $(x_i, y_i)$ be the two end-points of $e_{g,s}$, and $(x_i, y_i)$ and $(x_{i+1}, y_{i+1})$ are the two end-points of $b_{g,j}$, then

$$\Delta = \begin{bmatrix} x_{i-1} & y_{i-1} & 1 \\ x_i & y_i & 1 \\ x_{i+1} & y_{i+1} & 1 \end{bmatrix}$$

if $\Delta < 0$ then the turning direction between $e_{g,s}$ and $b_{g,j}$ is clockwise. ***)

For time complexity, steps (3) and (5) need $O(\log n)$ respectively, while other steps need constant time only. Therefore, the total time complexity of procedure *SINGLE_PATTERN* is $O(\log n)$.

### 4.2.4 Finding Multiple Congruent Patterns

For *Case* 3, G has equal-length edges but P has distinct edges. There can be 0, 1 or more congruent patterns in G and some pairs of them may be adjacent (i.e. have a common edge). For *Case* 4, both P and G have equal-length edges, respectively. We may also find 0, 1 or more congruent patterns in G, and some pairs of them may be adjacent or intersect each other. Hence, any edge in G may be a *common edge* of two adjacent or intersecting congruent patterns.

To find multiple patterns, first, we introduce some notations to distinguish different patterns in G. We use ST1 and ST2 to denote the starting edges of the first and second congruent pattern which $e_{g,j}$ belongs to, respectively. We also use flags EQ1 and EQ2 to denote that $e_{g,j}$ belongs to congruent pattern 1 and pattern 2, respectively.

Then, as in the procedure *SINGLE_PATTERN* in section 4.2.3, each *LP* also needs to parallel compare each edge pair $e_{p,i}$ and $e_{g,j}$, for $1 \le i \le k$ and $1 \le j \le n$, which are broadcasted by procedures *INITIALIZE_P* and *INITIALIZE_G*. If the pair of edges are of equal length, then set flags EQ1 and EQ2 to TRUE and compute ST1 and ST2 by the $RR_{p,i}$ in this *LP* else set EQ1 and EQ2 to FALSE.

#### 4.2.4.1 Checking the Existence of Starting Edges

Next, we have to check whether or not the two starting edges ST1 and ST2, in each *LP*, exist in G. For this step, we develop procedures *CHECK_EDGES_1* and *CHECK_EDGES_2* to parallel check all starting edges in *Case* 3 and *Case* 4, respectively. If starting edge ST1 (or ST2) is in G, then store its index (order) in G (e.g. the $j$ of $e_{g,j}$ in row 1) into this *LP*, else set EQ1 (or EQ2) to FALSE (i.e. ST1 or ST2 is not an edge of G).

For *Case* 3, since all edges in P are distinct, after comparing the length of the edge pair $e_{p,i}$ and $e_{g,j}$ in each *LP*, we will find that only one pair of them is equal in the *LP*'s of each column. Therefore, for each column, procedure *CHECK_EDGES_1* needs to check starting edges only once. On the other hand, for *Case* 4, it is possible that one or more ($e_{p,i}$, $e_{g,j}$) pairs are equal in each column processors. In this case, for each column, procedure *CHECK_EDGES_2* needs to check starting edges $k$ times. Procedures *CHECK_EDGES_1* and *CHECK_EDGES_2* are given below.

**procedure** *CHECK_EDGES_*1;

(* For *Case* 3, to check starting edges ST1 and ST2 in each *LP*, with EQ1 = TRUE and EQ2 = TRUE, are edges in G or not. *)

**begin**

(1) **for** $1 \le i \le n$ **do in parallel**

Use *LTR_R*$(i, i)$ and *BROADCAST_R*$(i)$ operations to broadcast $e_{g,i}$ in $LP_{i,i}$ to all the *LP*'s in row $i$;

**endfor**;

(2) **for** $1 \le i \le n$ and $1 \le j \le k$ **do in parallel**

**if** EQ1 = TRUE and EQ2 = TRUE in $LP_{i,j}$

**then** use *LTR_C*$(i, j)$ and *BROADCAST_C*$(i)$ operations to broadcast ST1 , ST2, EQ1, EQ2, and their column index $y$ (here, $y = j$) in $LP_{i,j}$ to all *LP*'s (from $LP_{1,r}$ to $LP_{n,r}$, for $1 \le r \le n$) in column $i$;

(* Broadcast ST1, ST2, EQ1, EQ2, and $y$ to all $n$ *LP*s in same column. *)

(* Note : in *Case* 3, in each column there is only one *LP* with EQ1 = TRUE and EQ2 = TRUE. Hence, this step only executes once. *)

**endfor**;

(3) **for** $1 \le i \le n$ and $1 \le j \le n$ **do in parallel**

**if** ST1 = $e_{g,i}$ in $LP_{i,j}$

**then** set EQ1 = TRUE and set $x1 = i$ **else** set EQ1 = FALSE;

**if** ST2 = $e_{g,i}$

**then** set EQ2 = TRUE and set $x2 = i$ **else** set EQ2 = FALSE;

**endfor**; (* $x1$ and $x2$ are row indices of the starting edges found. *)

(4) **for** $1 \leq i \leq n$ and $1 \leq j \leq n$ **do in parallel**

**for** each $LP_{i,j}$ **do** the following steps in sequence

use OR_C(j) operation to perform logic OR function for all EQ1 in each column $j$ and send the final value to root processor $RP_j^c$;

then use *LTL_C*($x1$, $y$, $j$) operation to send EQ1 and $x1$ from $LP_{x1,j}$ back to $LP_{y,j}$;

use OR_C(j) operation to perform logic OR function for all EQ2 in each column $j$ and send the final value to root processor $RP_j^c$;

then use *LTL_C*($x2$, $y$, $j$) operation to send EQ2 and $x2$ from $LP_{x2,j}$ back to $LP_{y,j}$;

**endfor**;

**endfor**;

**end**; (* of *CHECK_EDGES*_1 *)

(* All the operations of mesh of trees are defined in section 3.2.2. *)

(* After this step, the starting edge ST1 (and ST2) in each *LP* has been assigned an index $x1$ (and $x2$) if its EQ1 (and EQ2) is updated to TRUE. *)

Steps (1), (2) and (4) take $O(log\ n)$ time, while step (3) takes constant time, thus, the total time complexity for *CHECK_EDGES*_1 is $O(log\ n)$.

**procedure** *CHECK_EDGES*_2;

(* For *Case* 4, to check starting edges ST1 and ST2 in each *LP*, with EQ1 = TRUE and EQ2 = TRUE, are edges in G or not. *)

**begin**

(1) **for** $1 \le i \le n$ **do in parallel**

Use *LTR_R(i, i)* and *BROADCAST_R(i)* operations to broadcast $e_{g,i}$ in $LP_{i,i}$ to all the *LP*'s in row *i*;

**endfor;**

(2) **for** $1 \le t \le k$ **do** (* in sequence *)

(2.1) **for** $1 \le i \le n$ and $1 \le j \le n$ **do in parallel**

**if** EQ1 = TRUE and EQ2 = TRUE in $LP_{i,j}$

**then** use *LTR_C(i, j)* and *BROADCAST_C(i)* operations to broadcast ST1 , ST2, EQ1, EQ2, and their column index *y* (here, $y = j$) in $LP_{i,j}$ to all *LP*'s in column *i*;

**endfor;**

(* Broadcast ST1, ST2, EQ1, EQ2, and *y* to all *n LP*s in same column. *)

(* Note : in *Case* 4, in each column there are may be more than one *LP* with EQ1 = TRUE and EQ2 = TRUE. Hence, this step has to be executed *k* times. *)

(2.2) **for** $1 \le i \le n$ and $1 \le j \le n$ **do in parallel**

**if** ST1 = $e_{g,i}$ in $LP_{i,j}$

**then** set EQ1 = TRUE and set $x1 = i$ **else** set EQ1 = FALSE;

**if** ST2 = $e_{g,i}$

**then** set EQ2 = TRUE and set $x2 = i$ **else** set EQ2 = FALSE;

**endfor;** (* $x1$ and $x2$ are row indices of the starting edges found. *)

(2.3) **for** $1 \le i \le n$ and $1 \le j \le n$ **do in parallel**

**for** each $LP_{i,j}$ **do** (* in sequence *)

use OR_C(j) operation to perform logic OR function for all

EQ1 in each column $j$ and send the final value to root processor $RP_j^c$;

then use $LTL\_C(x1, y, j)$ operation to send EQ1 and $x1$ from $LP_{x1,j}$ back to $LP_{y,j}$;

use OR_C(j) operation to perform logic OR function for all EQ2 in each column $j$ and send the final value to root processor $RP_j^c$;

then use $LTL\_C(x2, y, j)$ operation to send EQ2 and $x2$ from $LP_{x2,j}$ back to $LP_{y,j}$;

**endfor**;

**endfor**;

**endfor**; (* end of step (2) *)

**end**; (* of *CHECK_EDGES_2* *)

(* After this step, the starting edge ST1 (and ST2) in each *LP* has been assigned an index $x1$ (and $x2$) if its EQ1 (and EQ2) is updated to TRUE. *)

For time complexity, step (1) takes constant time, steps (2.1), (2.2) and (2.3) are same as steps (2), (3) and (4) in procedure *CHECK_EDGES_1*, respectively. In procedure *CHECK_EDGES_2* , step (2) executes $k$ iterations. According the analysis of *CHECK_EDGES_1*, we know each iteration of step (2.1), (2.2) and (2.3) needs $O(\log n)$ time, hence, the total time complexity of *CHECK_EDGES_2* is $O(k \log n)$.

#### 4.2.4.2 Counting Edge Number of Each Pattern

After check all starting edges, for each possible congruent pattern, we have to count the number (say S) of all edges (i.e. $e_{g,j}$) with same starting edge in G, respectively. If S equals $k$ (the edge number of P), this means all these edges can form a congruent pattern , then output this congruent pattern, else (S < $k$) no pattern with this starting edge is found. For this step, we develop other two procedures *COUNT_EDGES_1* and *COUNT_EDGES_2* to count the edge number of each possible congruent pattern for *Case* 3 and *Case* 4, respectively.

**procedure** *COUNT_EDGES_1*;

(* For *Case* 3, to send each group of edges in parallel, which may form a congruent pattern, to the *LP* in row 1 with the starting edge of this pattern. Since exactly only one congruent edge in each column, so one iteration to send edges is enough. *)

**begin**

**for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**begin** (* in sequence *)

(1) The only $LP_{i,j}$ in each column with EQ1 = TRUE sends 1 and its edge $e_{g,j}$ from $LP_{i,j}$ to $LP_{1,}$ $x1$ and stores in array $PA[ss, 1]$ by the following routing rule :

(i, j) => (j, j) => (j, x1) => (1, x1)

(* x1 is the column index of the starting edge of this pattern *)

(* The above routing rule will guarantee that all data in the same row routing in parallel without any conflict. *)

(* This rule is equivalent to the RAW (random access write) operation of PRAM machine described in [NS81] *)

The rule can be performed by the MOT operations as follows :

*LTL_C*($i, j, j$), *LTL_R*($j, j, x1$), and *LTL_C*($j, 1, x1$)

(2) The only $LP_{i,j}$ in each column with EQ2 = TRUE sends 1 and its edge $e_{g,j}$ from $LP_{i,j}$ to $LP_{1,x2}$ and stores in array $PA[ss, 2]$ by the following routing rule :

(i, j) => (j, j) => (j, x2) => (1, x2)

The rule can be performed by the MOT operations as follows :

*LTL_C*($i, j, j$), *LTL_R*($j, j, x2$), and *LTL_C*($j, 1, x2$)

**end**;

**endfor**;

**end** of *COUNT_EDGES_*1;

In procedure *COUNT_EDGES_*1, since all *LP*'s do the data routing operations in parallel, so each of them spends $O(\log n)$ time in the operations of steps (1) and (2), respectively. Hence, the total time complexity is still $O(\log n)$.

**procedure** *COUNT_EDGES_*2;

(* For *Case* 4, to send each group of edges sequentially (one column at a time) , which may form a congruent pattern, to the *LP* in row 1 with the starting edge of this pattern. *)

**begin**

**for** 1 ≤ i ≤ k **do** (* in sequence *)

**for** 1 ≤ j ≤ n **do in parallel**

**begin** (* in sequence *)

(1) The only $LP_{i,j}$ in each column with EQ1 = TRUE sends 1 and its edge $e_{g,j}$ from $LP_{i,j}$ to $LP_{1,x1}$ and stores in array $PA[ss, 1]$ by the following routing rule :

(i, j) => (j, j) => (j, x1) => (1, x1)

(* x1 is the column index of the starting edge of this pattern *)

(* The above routing rule will guarantee that all data in the same row can be routed in parallel without any conflict. This rule is equivalent to the RAW operation of PRAM machine *)

The rule can be performed by the MOT operations as follows :

$LTL\_C(i, j, j)$, $LTL\_R(j, j, x1)$, and $LTL\_C(j, 1, x1)$

(2) The only $LP_{i,j}$ in each column with EQ2 = TRUE sends 1 and its edge $e_{g,j}$ from $LP_{i,j}$ to $LP_{1,x2}$ and stores in array $PA[ss, 2]$ by the following routing rule :

(i,j) => (j,j) => (j,x2) => (1,x2)

The rule can be performed by the MOT operations as follows :

$LTL\_C(i, j, j)$, $LTL\_R(j, j, x2)$, and $LTL\_C(j, 1, x2)$

**end**;

**endfor**;

**endfor**;

**end** of *COUNT_EDGES_2*;

For time complexity of procedure *COUNT_EDGES_2*, each *LP* has to do the operations in steps (1) and (2) for $k$ iterations. Each iteration needs $O(log\ n)$ time, as in procedure *COUNT_EDGES_1*, therefore, the total time is $O(k \log n)$.

#### 4.2.4.3 The Main Procedure MULTIPLE_PATTERNS

We organize the above discussions and four procedures as the main procedure *MULTIPLE_PATTERNS* to find multiple congruent patterns in the following.

**procedure** *MULTIPLE_PATTERNS*($EL_p$);

*Input*: The outputs from procedures *INITIALIZE_P* and *INITIALIZE_G*;

*Output*: The multiple congruent patterns in G for *Case* 3 and *Case* 4;

**begin**

(1) **for** $1 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**begin**

**if** $|e_{p,i}| = |e_{g,j}|$

**then** set EQ1 = TRUE and EQ2 = TRUE ;

(* Each $LP_{i,j}$ compares the length of edges $e_{p,i}$ and $e_{g,j}$ *)

**if** $|e_{p,1}| = |e_{g,j}|$

**then** set sum $S_{1,j} = 1$ and ss = j in $LP_{1,j}$;

**end**;

**endfor**;

(2) **for** $2 \le i \le k$ and $1 \le j \le n$ **do in parallel**

**if** $LP_{i,j}$ sets its flag EQ1 = TRUE or EQ2 = TRUE

**then** use the record $RR_{p,i}$ as the reference, to compute $RR_{g,j1}$ and $RR_{g,j2}$, and find the starting edges $e_{g,s1}$, denoted as ST1, and $e_{g,s2}$, denoted as ST2;

**endfor**;

(3) **if** $EL_p$ = FALSE **then** call procedure CHECK_EDGES_1; (* for *Case* 3 *)

**else** Call procedure CHECK_EDGES_2; (* for *Case* 4 *)

(* To check whether edges ST1 and ST2 in each *LP* are starting edges or not. *)

(4) **if** $EL_p$ = FALSE **then** call procedure COUNT_EDGES_1 (* for *Case* 3 *)

**else** call procedure COUNT_EDGES_2; (* for *Case* 4 *)

(* To count the edge number of any possible congruent pattern. *)

(5) **for** $1 \leq ss \leq n$ **do in parallel**

**if** $S = k$ in $LP_{1,ss}$

**then** output the pattern arrays $PA[ss, 1]$ and $PA[ss, 2]$ in $LP_{1,ss}$ to its root processor $RP^c_{ss}$ in column $ss$.

(* These edges form a congruent pattern and its starting edge is $e_{g,ss}$ *)

**else** print "There is no congruent pattern with $e_{g,ss}$ as the starting edge.";

**endfor**;

**end** of *MULTIPLE_PATTERNS*;

For the analysis of time complexity of procedure *MULTIPLE_PATTERNS*, steps (1), (2), and (5) need only constant time. For *Case* 3, Steps (3) and (4) need $O(\log n)$ time. For *Case* 4, steps (3) and (4) need $O(k \log n)$ time. Thus, the time complexity is $O(\log n)$ for *Case* 3, and $O(k \log n)$ for *Case* 4.

### 4.2.5 The Main Algorithm

In our main algorithm, we first call procedures *INITIALIZE_P* and *INITIALIZE_G* to do the initial operations on P and G respectively. Then, we call procedures

*CHECK_P_LENGTH* and *CHECK_G_LENGTH* to check whether there are equal-length edges in P and G, respectively. Finally, according to the property of length equality, we call procedures *SINGLE_PATTERN* and *MULTIPLE_PATTERNS* to solve *Case* 1 to *Case* 4. The main algorithm *CONGRUENT_PATTERNS* is as follows.

**Algorithm** *CONGRUENT_PATTERNS*;

*Input* : A test pattern (simple polygon) P with $k$ edges and a planar straight-line graph G with $n$ edges, for $k \leq n$;

*Output* : The all patterns (cycles) in G which are congruent to P;

**begin**

(1) Call procedure *INITIALIZE_P* to do initial operations on P;

(2) Call procedure *INITIALIZE_G* to do initial operations on G;

(3) Call procedure *CHECK_P_LENGTH* to check the edge equality for P, and receive the $EL_p$ value from $RP_1^c$;

(4) Call procedure *CHECK_G_LENGTH* to check the edge equality for G; and receive the $EL_g$ value from $RP_1^r$;

(5) **If** $EL_p$ = TRUE but $EL_g$ = FALSE **then** print "No congruent patterns in G.";

(* P has equal-length edges but all edges in G are distinct. This is *Case* 1. *)

(6) **If** $EL_p$ = FALSE and $EL_g$ = FALSE **then** call procedure *SINGLE_PATTERN*;

(* All edges in P and G are distinct. Obviously, there is at most one congruent pattern in G. This is *Case* 2. *)

(7) **If** $EL_p$ = FALSE but $EL_g$ = TRUE **then** call procedure *MULTIPLE_PATTERNS*($EL_p$);

(* There are $x$ congruent patterns in G, for $x \geq 0$. This is *Case* 3 *)

(8) **If** $EL_p$ = TRUE and $EL_g$ = TRUE **then** call procedure *MULTIPLE_PATTERNS*($EL_p$);

(* There are $y$ congruent patterns in G, for $y \geq 0$. This is *Case* 4 *)

**end** of *CONGRUENT_PATTERNS*;

**Theorem 4.1** : Given a test pattern P (a simple polygon) with $k$ edges and a planar straight-line graph G with $n$ edges, all the patterns (cycles) in G which are congruent to P can be found in $O(\log n)$ time on a mesh of trees, if each edge in P has unique length (for *Case* 3), otherwise (for *Case* 4) the time complexity is $O(k \log n)$.

**Proof** : From the algorithm *CONGRUENT_PATTERNS*, we know that step (5) runs in constant time, steps (1), (2), (3), (4) and (6) need $O(\log n)$ time, and steps (7) and (8) need $O(\log n)$ time for *Case* 3 and $O(k \log n)$ time for *Case* 4. Hence, the total time complexity for algorithm *CONGRUENT_PATTERNS* is $O(\log n)$ for P with distinct edges, otherwise (for general case, *Case* 4) is $O(k \log n)$. □

**Corollary 4.1** : If all edges in P and G are distinct respectively, then the time complexity for finding congruent pattern on a mesh of trees is $O(\log n)$.

**Proof** : Obviously, this is *Case* 2. By Lemma 4.2, we know that there is zero or only one congruent pattern in G. Therefore, we have to execute steps (1), (2), (3), (4) and (6). So, from the proof in Theorem 4.1 , the total time complexity for this case is $O(\log n)$. □

## 4.3 Final Remarks

We presented a MOT (mesh of trees) algorithm for reporting all patterns in a planar straight-line graph which are congruent to a given simple pattern. The MOT has $n \times n$ processors and the time complexity is $O(log\ n)$ for the given has distinct edges, and $O(k \log n)$ for general case, they are all optimal.

It remains a problem whether or not MOT algorithms can give a same time complexity for finding similar patterns. A pattern in G is similar to a given pattern G if the *length rate* of each corresponding edge pair in P and G are equal, and they also have same interior angle and turning direction (as defined in Section 4.1). It also remains open that it is possible or not to develop an optimal parallel algorithm on a mesh of trees to solve the three-directional congruent pattern problems as described in [A87] and [S84]. If the polyhedra described in the two papers are projected on the plane, then maybe it is possible to develop similar MOT algorithms to solve these problems with better time complexity.

# Chapter 5

# Conclusion

We conclude the dissertation by discussing the contributions of our research, and the time complexities and open problems for our parallel geometric algorithms. Finally, we propose a plan for future work.

## 5.1 Discussing of The Research Results

(1) *The Maxima Problem*

In chapter 2, we developed a parallel algorithm to find the maxima of a point set on a hypercube machine with $N$ processors. For a set of $N$ points in $d$-dimensional space, our algorithm takes $O(N^{0.77}\log^{d-1} N)$ time. This is the first sublinear-time parallel algorithm for multiprocessor systems connected by sparse networks. If the size $N$ of input point set is greater than $P$, the number of processors in a hypercube, then our algorithm can be generalized to solve the maxima problem in $O((N/P^{0.23})log^{d-2}P(log(N/P) + \log P))$ time. Table 5.1 shows the time complexity of the various maxima algorithms.

It remains an open problem whether or not our results can be significantly improved. For example, is it possible to obtain a hypercube maxima algorithm which runs in $O(N^c)$ time such that $c$ is a constant less than 1? Moreover, can we design a hypercube maxima algorithm which runs in $O(log^{f(d)})$ time, where $f(d)$ is a linear function of the $d$-dimensional point set? Due to the limitation of data communication

in hypercube machine, we believe that it will not be easy to find such efficient algorithms.

The maxima problem has many important applications in statistics, economics, and operations research. With slight modifications, this algorithm can be extended to compute the contour which is spanned by the maximal elements and to compute the number of dominated points for every element of the input point set. This is the ECDF (empirical cumulative distribution function) searching problem [D86].

Table 5.1 Time Complexity for Maxima Algorithms

| Algorithm | Model | Time Complexity | Dimension |
|---|---|---|---|
| [KLP75] | sequential | $O(N \log N)$ | $d \leq 3$ |
| [KLP75] | sequential | $O(N \log^{d-2} N + N \log N)$ | $d > 3$ |
| [ACG87] | CREW PRAM | $O(\log N)$ | $d = 2$ and $N = P$ |
| [ACG87] | CREW PRAM | $O(\log^{d-2} N)$ | $d \geq 3$ and $N = P$ |
| [D86] | mesh-connected | $O(N^{0.5})$ | $d \leq 3$ and $N = P$ |
| [D86] | mesh-connected | $O(N^{0.5+\log(d-2)})$ | $d > 3$ and $N = P$ |
| [S88] | hypercube | $O(\log^2 N)$ | $d = 2$ and $N = P$ |
| [ZL91] | hypercube | $O(N^{0.77} \log^{d-1} N)$ | $d > 3$ and $N = P$ |
| [ZL91] | hypercube | $O(N/P^{0.23} \log^{d-2} P(\log(N/P) + \log P))$ | $d > 3$ and $N > P$ |

(2) *The Voronoi Diagram*

In chapter 3, we presented an optimal $O(log^2 N)$-time parallel algorithm to construct a three-dimensional convex hull of a point set in the space, where $N$ is the size of the given point set. We then used the three-dimensional convex hull algorithm to construct the Voronoi diagram of a set of $N$ points in the plane on an $N \times N$ mesh of

trees. The Voronoi diagram algorithm also runs in optimal time $O(log^2 N)$. A comparison of time complexity for the various parallel algorithms for Voronoi diagram is given in Table 5.2.

Table 5.2 Time Complexity for Voronoi Diagram Algorithms

| Algorithm | Model | Time Complexity |
|---|---|---|
| [B79] | sequential | $O(N \log N)$ |
| [PS88] | sequential | $O(N \log N)$ |
| [C80] | CREW PRAM | $O(\log^3 N \log \log N)$ |
| [ACGOY85] | CREW PRAM | $O(\log^3 N)$ |
| [C80] | cube-connected-cycles | $O(\log^4 N)$ |
| [L86] | mesh-connected | $O(N^{0.5} \log N)$ |
| [S88] | hypercube | $O(\log^3 N)$ |
| [LZ92] | mesh-of-trees | $O(\log^2 N)$ |

It remains an open problem whether or not the geometric inversion approach introduced in [B79] can be used to develop a general and efficient parallel algorithm for constructing a $(k+1)$-dimensional convex hull. If so, we may use this convex hull to construct a $k$-dimensional Voronoi diagram.

The two-dimensional Voronoi diagram is one of the most useful data structures in computational geometry. It can be directly or indirectly used to solve *the nearest neighbor, all nearest-neighbors, convex hull, Delaunay triangulation, Euclidean minimum spanning tree*, and the *traveling salesman problem* easily and efficiently. Besides, it also has many important applications in robotics, image processing and graph theory ([PS88], [S78]).

(3) *The Congruent Patterns*

In chapter 4, we presented two parallel algorithms for reporting all the patterns (cycles) in a planar straight-line graph G which are congruent to a given test pattern P (a simple polygon), respectively. This algorithm runs on a mesh of trees parallel machine, with $n \times n$ processors. It takes the optimal time complexity $O(log\ n)$ for the test pattern P with distinct edges, otherwise for general case, the optimal time complexity is $O(k \log n)$, where $n$ is the edge number of G. A comparison of the time complexity of our algorithms with other parallel algorithms is given in Table 5.3.

Table 5.3 Time Complexity for Congruent Pattern Algorithms

| Algorithm | Model | Time Complexity | Dimension |
|---|---|---|---|
| [S84] | sequential | $O(N \log N)$ | $d = 3$ and $k = N$ |
| [A87] | sequential | $O(N \log N)$ | $d = 3$ and $k = N$ |
| [SLY90] | CREW PRAM | $O(N^2)$ | $d = 2$ and $k < N$ |
| [JKB92] | CREW PRAM | $O(log\ N \log \log N)$ * | $d = 2$ and $k < N$ |
| ours | mesh-of-trees | $O(log\ N)$ * | $d = 2$ and $k < N$ |
| [JKB92] | CREW PRAM | $O(k \log N \log \log N)$ ** | $d = 2$ and $k < N$ |
| ours | mesh-of-trees | $O(k \log N)$ ** | $d = 2$ and $k < N$ |

* : Each edge in the test pattern P has unique length.
** : Some edges in the test pattern P have not unique length (i.e. the general case).
$k$ and $N$ are the edge number of P and G, respectively.

We wander that it is possible to develop another efficient MOT algorithm to report all patterns in a given graph G which are similar to a given test polygon P. Besides, it remains an open problem whether or not our optimal algorithms can be generalized to find $d$-dimensional congruent polyhedra, for $d \geq 3$. Finding congruent

and similar patterns arises frequently in computer-aided design (CAD), image analysis, pattern recognition, and computer vision.

## 5.2 Future Work

In the near future, we plan to continue our research on the *visibility* problems, such as (1) computing the *visibility polygon* from a point inside or outside a given simple polygon, (2) computing the *visibility graph* of a segment set, or (3) Removing hidden surfaces from a three-dimensional image. All the above problems will be solved on a mesh of trees. The visibility problems have many applications in computer graphics, robotics, computer vision, and graph theory.

(1) *Computing Visibility Polygon*

Given a simple polygon $P$ with $n$ vertices and a viewpoint $z$ in the plane, find all points on the boundary of $P$ that are *visible* from $z$. All the visible boundary points form the *visibility polygon* (see Figure 5.1). For this problem, Gindy and Avis presented an $O(n)$ time sequential algorithm for finding the visibility polygon from a point inside $P$ [GA81].

In [L83], Lee developed another sequential algorithm for finding the visibility polygon of a point inside or outside $P$. His algorithm also runs in $O(n)$ time, but it uses only one stack instead of three as required by [GA81]. In [AC91], Atallah and Chen presented a parallel algorithm for computing the visible portions of a simple polygonal chain with $n$ vertices from a point in the plane. Their algorithm runs in $O(\log n)$ time using $O(n / \log n)$ processors in the EREW PRAM computational model, and hence is asymptotically optimal.

In [CG85], Chazelle and Guibas used *geometric duality* on the two-sided plane to solve various visibility and intersection problems about a simple polygon $P$ with $n$ vertices. Among their results are a simple $O(n \log n)$ algorithm for computing the illuminated subpolygon of $P$ from a luminous side, an $O(log\ n)$ algorithm for determining which side of $P$ is first hit by a bullet fired from a point in a certain direction, and an $O(n \log n)$ algorithm for computing the first intersection of $P$ with a light ray from the luminous edge. All of the above are sequential algorithms.

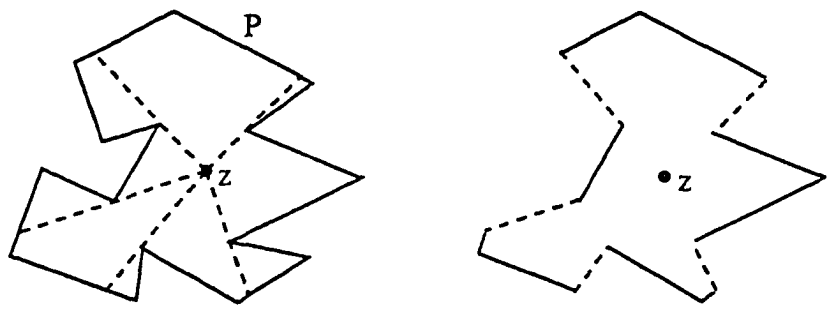


Fig. 5.1 A simple polygon P and the visibility polygon of viewpoint z.

For our research, we will try to use geometric duality approach to find the visibility polygon from a point inside or outside $P$. A reason is that geometric duality is a simpler way for solving this problem. The only problem is that it is generally not very easy to figure out the main idea of this approach.

(2) *Computing Visibility Graph*

Given a set $S$ of $n$ nonintersecting line segments in the plane, its visibility graph $G_S$ is the undirected graph which has the endpoints of the line segments in $S$ as nodes and in which two nodes (points) are adjacent whenever they *see* each other (the line segments in $S$ are regarded as nontransparent obstacles, see Figure 5.2). For this

problem, Welzl [W85a] and Edelsbrunner [E87] have presented sequential algorithms to find $G_S$ in $O(n^2)$ time, respectively.

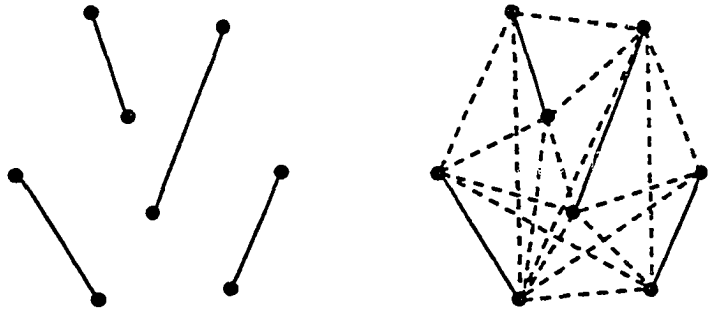

Fig. 5.2 A set of segments and its visibility graph.

In [H87], Hershberger described an $O(m + n \log \log n)$ sequential algorithm to solve this problem, where $m$ is the number of edges of $G_S$. Because $m$ can be as small as $O(n)$, the algorithm improved Welzl's algorithm. In [OW88], Overmars and Welzl presented an $O(m \log n)$ sequential algorithm for finding sparse $G_S$, where $m$ is the edge number of $G_S$. For our research, we will try to develop a logarithmic time algorithm to find $G_S$ on a mesh of trees.

(3) *Removing Hidden Surfaces*

For this problem, Reif and Sen presented a sequential algorithm for removing hidden surfaces in a three-dimensional terrain map [RS88]. This algorithm runs in $O((k + n) \log n \log \log n)$ time. In the same paper, they also presented a parallel algorithm which runs in time $O(log^4(k + n))$ in a CREW PRAM model. In [B88], Bern presented a sequential algorithm for removing hidden surfaces of a set of $n$ rectangles with sides parallel to the $x$ and $y$ axes, with viewpoint at $z = \infty$ (that is, an orthographic projection). Their algorithm runs in time $O(n \log n \log \log n + k \log n)$,

where $k$ is the number of output line segments. In [MNU90], Mehlhorn, Naher, and Uhrig solved the same problem as in [B88]. Their algorithm runs in time $O(n \log n + k \log(n^2 / k))$ (see Figure 5.3).

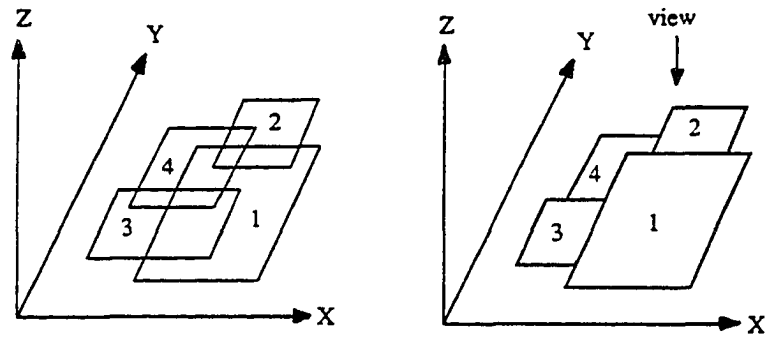


Fig. 5.3 Removing hidden-surfaces of rectangular objects.

In [BO90], Berg and Overmars presented a new and efficient sequential algorithm for computing the visibility map of a set of axis-parallel polyhedra (that is, polyhedra with their faces and edges parallel to the coordinate axes) as seen from a given view-point. For non-intersecting polyhedra with $n$ edges in total, the algorithm runs in time $O((k + n) \log n)$, where $k$ is the complexity of the visibility map. The algorithm can be extended to deal with intersecting polyhedra with only a slight increase in the time bound. For our research, we plan to develop an optimal parallel algorithm to solve a similar problem as in [B88] and [MNU90] on a mesh of trees.

## Bibliography


[A87] M. D. Atkinson, "An Optimal Algorithm for Geometrical Congruence", *Journal of Algorithms*, Vol. 8, 1987, pp159-172.

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

[AC91] M. J. Atallah and D. Z. Chen, "An Optimal Parallel Algorithm for the Visibility of a Simple Polygon from a Point", *Journal of the Association for Computing Machinery*, Vol. 38, No. 3, July 1991, pp516-533.

[ACG85] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap "Parallel Computational Geometry", *IEEE Symposium on Foundations of Computer Science*, 1985, pp468-477.

[ACG87] M. J. Atallah, R. Cole, M. T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms", *IEEE Symposium on Foundations of Computer Science*, October 1987, pp151-160.

[AG85] M. Atallah and M. Goodrich, "Efficient Parallel Solutions to Geometric Problems", *IEEE Int'l Conf. on Parallel Processing*, 1985, pp411-417.

[AG86] M. Atallah and M. Goodrich, "Efficient Plane Sweeping in Parallel", *ACM Symposium on Computational Geometry*, 1986, pp216-225.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Bell Telephone Lab. Inc., 1974.

[AU87] T. Asano and H. Umeo, "Systolic Algorithms for Computing the Visibility Polygon and Triangulation of a Polygon Region", *Parallel Algorithms and Architectures*, ed. Albrecht, Jung, and Mehlhorn, 1987, pp77-85.

[B79] K. Q. Brown, "Voronoi Diagrams from Convex Hulls", *Information Processing Letters*, Vol. 9, No. 5, Dec. 1979, pp223-228.

[B88] M. Bern, "Hidden Surface Removal for Rectangles", *ACM Symposium on Computational Geometry*, 1988, pp183-192.

[BO90] M. Berg and M. H. Overmars, "Hidden Surface Removal for Axis-Parallel Polyhedra", *IEEE Symposium on Foundations of Computer Science*, 1990, pp252-261.

[C80] A. L. Chow, *Parallel Algorithms for Geometric Problems*, Ph. D. Thesis, University of Illinois, 1980.

[C84] B. Chazelle, "Computational Geometry on a Systolic Chip", *IEEE Transactions on Computers*, Vol. c-33, No. 9, 1984, pp774-785.

[C90] D. Z. Chen, "Efficient Geometric Algorithms in the EREW-PRAM", Department of Computer Science, Purdue University, 1990.

[CG85] B. Chazelle and L. J. Guibas, "Visibility and Intersection Problems in Plane Geometry", *ACM Symposium on Computational Geometry*, 1985, pp135-145.

[D72] J.C. Dodge, "Euclidean Geometry and Transformations", Addison-Wesley, Reading, MA, 1972.

[D86] F. Dehne, "$O(n^{1/2})$ Algorithms for the Maximal Elements and ECDF Searching Problem on a Mesh-Connected Parallel Computer", *Information Processing Letters*, May 1986, pp303-306.

[DS88] F. Dehne and J. Sack, "A Survey of Parallel Computational Geometry Algorithms", *Parcella 1988*, (Fourth International Workshop on Parallel Processing by Cellular Automata and Arrays), ed. Wolf, Legend, and Schendel, pp73-88.

[E88] H. Edelsbrunner, *Algorithms for Combinatorial Geometry*, Springer-Verlag, 1988, pp275-278.

[GA81] H. E. Gindy and D. Avis, "A Linear Algorithm for Computing the Visibility Polygon from a Point", *Journal of Algorithms*, Vol.2, 1981, pp186-197.

[H87] J. Hershberger, "Finding the Visibility Graph of a Simple Polygon Time Proportional to its Size", *ACM Symposium on Computational Geometry*, 1987, pp11-20.

[JKB92] C. S. Jeong, S. M. Kim and S. B. Baek, "Parallel Computation of Congruent Regions on SIMD Machines", to appear in the *International Conference on Parallel Processing*, August 1992.

[KE86] V. Kumar and M. Eshaghian, "Parallel Geometric Algorithms for Digitized Pictures on Mesh of Trees", *International Conference on Parallel Processing* , 1986, pp270-273.

[KLP75] H. Kung, F. Luccio, and F. Preparata, "On Finding the Maxima of a Set of Vectors", *Journal of the ACM*, Vol.22, No.4, October 1975, pp469-476.

[L83] D. T. Lee, "Visibility of a Simple Polygon", *Computer Vision, Graphics, and Image Processing*, Vol.22, 1983, pp207-221.

[L86] M. Lu, "Constructing the Voronoi Diagram on a Mesh-Connected Computer", *IEEE International Conference on Parallel Processing*, 1986, pp806-811.

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

[LP84] D. Lee and F. Preparata, "Computational Geometry - A Survey", *IEEE Transactions on Computers*, December 1984, pp1072-1101.

[LZ92] F. Lee and S. Q. Zheng, "Constructing Voronoi Diagram of a Point Set on Mesh of Trees", to appear in the *IEEE International Conference on Parallel Processing*, 1992.

[MI87] A. Moitra and S. Iyengar, *Parallel Algorithms for a Class of Computational Problems - A Survey*, Academic Press, Inc., 1987.

[MM87] R. Miller and S. Miller, "Using Hypercube Multiprocessors to Determine Geometric Properties of Digitized Pictures", *IEEE International Conference on Parallel Processing*, 1987, pp638-640.

[MNU90] K. Mehlhorn, S. Naher, and C. Uhrig, "Hidden Line Elimination for Iso-Oriented Rectangles", *Information Processing Letters*, 1987, pp137-143.

[MS87] R. Miller and Q. Stout, "Mesh Computer Algorithms for Line Segments and Simple Polygons", *IEEE International Conference on Parallel Processing*, 1987, pp282-285.

[OW88] M. H. Overmars and E. Welzl, "New Methods for Computing Visibility Graphs", *ACM Symposium on Computational Geometry*, 1988, pp164-171.

[PH77] F. P. Preparata and S. J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", *Communications of the ACM*, Feb. 1977.

[PS88] F. P. Preparata and M. I. Shamos, *Computational Geometry : An Introduction*, Revised Edition, Springer-Verlag, 1988.

[Q87] M. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.

[RS87] J. H. Reif and S. Sen, "Optimal Randomized Parallel Algorithms for Computational Geometry", *International Conference on Parallel Processing*, 1987, pp270-277.

[RS88] J. H. Reif and S. Sen. "An Efficient Output-Sensitive Hidden-Surface Removal Algorithm and Its Parallelization", *ACM Symposium on Computational Geometry* , 1988, pp193-200.

[S78] M. I. Shamos, *Computational Geometry*, Ph. D. Thesis, Yale University, Department of Computer Science, May, 1978.

[S84] K. Sugihara, "An $n \log n$ Algorithm for Determining the Congruity of Polyhedra", *Journal of Computer and System Sciences*, Academic Press, Inc.,Vol.29, 1984, pp36-47.

[S88a] R. Sedgewick, *Algorithms*, 2nd Edition, Addison-Wesley, Inc., 1988.

[S88b] I. Stojmenovic, "Computational Geometry on a Hypercube", *IEEE International Conference on Parallel Processing*, Vol. 3, 1988, pp100-103.

[SLY90] Z. C. Shih, R. C. T. Lee and S. N. Yang, "A Parallel Algorithm for Finding Congruent Regions", *Parallel Computing 13*, 1990, pp135-142.

[U84] J. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Inc., 1984, pp158-160.

[VDP89] S. Vandewalle, J. De Keyser, and R. Piessens, "The Numerical Solution of Elliptic Partial Differential Equations on a Hypercube Multiprocessor", *Scientific Computing on Supercomputers*, ed. Devreese and Van Camp, 1989, pp69-73.

[W85a] E. Welzl, "Constructing the Visibility Graph for $n$-Line Segments in $O(n^2)$ Time", *Information Processing Letters*, Vol.20, No.4, 1985, pp167-171.

[W85b] S. H. Whitesides, "Computational Geometry and Motion Planning", *Computational Geometry*, ed. G. Toussaint, North-Holland, 1985, pp377-427.

[W85c] D. Wood, "An Isothetic View of Computational Geometry", *Computational Geometry*, ed. G. Toussaint, North-Holland, 1985, pp429-459.

[ZL91] S. Q. Zheng and F. Lee, "A Hypercube Algorithm for the Maxima Problem", *The 29th Annual Allerton Conference on Communication, Control, and Computing*, 1991.

# Vita

Fenglien Lee was born in Changhua, Taiwan, on May 17, 1952. He received a B. Ed. degree in Industrial Education from National Taiwan Normal University on July. 1980. From August 1980 to July 1982, he taught at Taipei Municipal Vocational Industrial High School, Taiwan. Then, he entered the graduate program in computer science at Indiana University - Bloomington and received an M. S. degree on May, 1984. From August 1984 to July 1987, he taught, as an instructor, at National Taiwan Normal University. On August 1987, he joined the Ph. D. program in computer science at Louisiana State University. Since June 1988 he has been a teaching assistant at Computer Science Department, LSU.

Mr. Lee's current research interests are parallel and distributed computing, computer architecture, operating systems, database, and image processing. He is a student member of the ACM and a member of the IEEE Computer Society.

# DOCTORAL EXAMINATION AND DISSERTATION REPORT

**Candidate:** Fenglien Lee

**Major Field:** Computer Science

**Title of Dissertation:** Parallel Geometric Algorithms

**Approved:**

Major Professor and Chairman

Dean of the Graduate School

**EXAMINING COMMITTEE:**

**Date of Examination:**

5/7/92