A PARALLEL ALGORITHM OF CONSTRUCTING A VORONOI DIAGRAM ON HYPERCUBE CONNECTED COMPUTER NETWORKS

WENMAO CHAI

A Parallel Algorithm of Constructing A Voronoi Diagram on Hypercube Connected Computer Networks

By

Wenmao Chai

A thesis submitted to the School of Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland
St. John's Newfoundland Canada

Contents

List of Figures

List of Tables

Acknowledgment

I would like to take this opportunity to express my sincere thanks to my supervisor, Dr. Caoan Wang. He stimulated my interest in the field of computational geometry. His constant encouragement and guidance during the course of my study and research led to the completion of this thesis.

Last but not the least, I would like to give my thanks to my husband, Zhenpen Young, for his love and help throughout the composition of this thesis.

Abstract

Computational geometry is a branch of computer science concerned with the design and analysis of algorithms to solve geometric problems. The Voronoi diagram of a set S of n points (called sites) is a well known structure in computational geometry. In the Voronoi diagram, each point is surrounded by a convex polygon enclosing that territory which is closer to the surrounded point than to any other point in the set. Voronoi diagrams are useful in solving geometric problems such as proximity problems and the Euclidean minimum spanning tree problem. Voronoi diagrams also have applications in diverse areas like biology, visual perception, physics, and archeology.

There exist many methods to construct Voronoi diagrams on a single computer. Two of them are proposed by Shamos and Brown. In 1975, Shamos applied two-dimensional Voronoi diagrams to obtain elegant solutions in computational geometry, such as finding the nearest neighbor and construction of minimum spanning trees. Shamos described an $\mathcal{O}(n \log n)$ time sequential algorithm to construct the planar Voronoi diagram for a set of planar points. The strategy he used in the serial algorithm is divide-and-conquer. In 1979, Brown demonstrated an interesting linkage between the Voronoi diagram and the convex hull. He presented an $\mathcal{O}(n \log n)$ algorithm for constructing the Voronoi diagram, by transforming the problem of constructing a planar Voronoi diagram for an n-points set to the construction

of a convex hull of n points in 3-dimensional space *via* a geometric transformation known as inversion.

Due to the nature of some applications in which geometric problems arise, fast and even real-time algorithms are often required. Here, as in many other areas, parallelism seems to hold the greatest promise for major reduction in computation time. The idea is to use several processors which cooperate to solve a given problem simultaneously in a fraction of the time taken by a single processor. Therefore, it is not surprising that the interest in parallel algorithms for geometric problems has grown in recent years.

Based on Shamos's and Brown's methods, several parallel algorithms to construct Voronoi diagrams have been presented. Some of them are implemented on parallel random access machines. Some of them are run on processor networks such as mesh, cube-connected cycles, stars and pancakes. We will discuss them in the literature review. With the development of communication technique and concurrent programming, computer networks have become popular. The most popular processor interconnection topology today is undoubtedly the *hypercube*. Hypercubes have several advantages. First, the number of nodes in a hypercube grows exponentially with the number of connections per node, so that a small increase in the hardware at each node allows a large increase in the size of the computer. Second, the number of alternative paths between nodes increases with the size of the hypercube,

which helps relieve congestion. Third, efficient algorithms are known for routing messages between processors in a hypercube. Finally, and today most importantly, a large corpus of software and programming techniques exists for hypercube.

The hypercube is one of the most versatile and efficient networks yet discovered for parallel computation. In this thesis, a single instruction multiple data stream hypercube connected computer network is chosen as our parallel computation model. In a hypercube connected computer network, local computations as well as message exchanges are taken into consideration when analyzing the time taken by the processor networks to solve a problem. Based on Nassimi and Sahni's paper fundamental operations on hypercube connect computer networks are descriptively discussed. The corresponding programs are also given. Based on Brown's approach, a parallel algorithm to construct Voronoi diagrams are developed. Our algorithm runs in $\mathcal{O}(\log^3 n)$ time on an $\mathcal{O}(n)$-processor hypercube connected computer network. Our algorithm has several advantages. First, our algorithm is based on Brown's method which transforms the problem of construction of a planar a Voronoi diagram for an n-point set to construction of a convex hull of n points in three dimensional space. Compared with the parallel algorithms which are based on the divide-and-conquer approach used by Shamos, our algorithm can be used to solve two computational geometry problems: constructing 2−dimensional Voronoi diagrams and 3− dimensional convex hulls.

Second, comparing with Chow's methods which runs on a $\mathcal{O}(n)$ processors CCC (Cube-Connected Cycles) model has $\mathcal{O}(\log^4 n)$ time complexity, our algorithm has better time complexity. Third, comparing with Chang-Sung Jeong's algorithm which runs in $\mathcal{O}(\sqrt{n})$ on an $\sqrt{n} \times \sqrt{n}$ mesh, our parallel computation model is more general because most other popular networks can be easily mapped onto a hypercube.

Chapter 1

Introduction

Programming computers to process pictorial data efficiently has been an activity of growing importance over the last 40 years. These pictorial data may come from many sources. We distinguish two general classes:

1. Most often, the data are inherently pictorial, such as the images arising in medical, scientific, and industrial applications. Weather maps received from satellites in outer space are a good example.

2. Alternatively, the data are obtained when a mathematical model is used to solve a problem and the model relies on pictorial data. Examples here include computing the average of a set of data (represented as points in space) in the presence of outliers, computing the value of a function that satisfies a set of constraints, and so on.

Regardless of their sources, these computations may include the operations of identifying contours of objects, "noise" removal, feature enhancement, pattern recognition, detection of hidden lines, and obtaining intersections among various components. At the foundation of all these computations are problems of a geometric nature, that is, problems involving points, lines, polygons, and circles. *Computational geometry* is the branch of computer science concerned with designing efficient algorithms for solving geometric problems of inclusion, intersection, and proximity, to name but a few.

Until recently, these problems were solved using conventional *sequential* computers, computers whose design more or less follows the model proposed by John von Neumann and his team in the late 1940s. The model consists of a single processor capable of executing exactly one instruction of a program during each time unit. Computers built according to this paradigm have been able to perform at tremendous speeds. However, it seems today that this approach has been pushed as far as it will go, and that the simple laws of physics will stand in the way of further progress. For example, the speed of light imposes a limit that cannot be surpassed by any electronic device.

On the other hand, our appetite grows continually for ever more powerful computers capable of processing large amounts of data at great speeds. One solution to this predicament that has gained credibility and popularity is *parallel processing*. Here a computational problem to be solved is broken

into smaller parts that are solved simultaneously by the several processors of a *parallel computer*. The idea is a natural one, and the decreasing cost and size of electronic components have made it feasible. Lately, computer scientists have been busy building parallel computers and developing algorithms and software to solve problems on them. One area that has received its fair share of interest is the development of *parallel algorithms* for computational geometry.

The Voronoi diagram is a mathematical concept attributed to mathematician Voronoi [1]. Voronoi diagrams have been well studied in computational geometry since the work of Shamos [2] partly because of their applications in solving geometric problems such as proximity problems and the Euclidean Minimum spanning tree problem, as well as their applications in such diverse areas as biology, visual perception, physics, and archeology [3].

There exist many methods to construct Vorinoi diagram on a single computer. Two of them are proposed by Shamos and Brown. In 1975, Shamos applied two-dimensional Voronoi diagrams to obtain elegant solutions in computational geometry, such as finding the nearest neighbor and construction of minimum spanning trees. Shamos described an $\mathcal{O}(n \log n)$ time sequential algorithm to construct the planar Voronoi diagram for a set of planar points. The strategy he used in the serial algorithm is divide-and-

conquer. In 1979, Brown demonstrated an interesting linkage between the Voronoi diagram and the convex hull. He presented an $\mathcal{O}(n \log n)$ algorithm for constructing the Voronoi diagram, by transforming the problem of constructing a planar Voronoi diagram for an n-points set to the construction of a convex hull of n points in 3-dimensional space *via* a geometric transformation known as inversion.

In this thesis, the hypercube connected computer network is chosen as the parallel computation model. Based on Brown's approach [4] which transforms the problem of construction of a planar a Voronoi diagram for an n-point set to construction of a convex hull of n points in three dimensional space, a parallel algorithm to construct Voronoi diagrams will be developed. In this chapter, after we introduce parallel computation models, we will review parallel algorithms to construct Voronoi diagrams. Finally, the outline of the thesis will be given.

1.1 Parallel Computation Models

In order to review the parallel algorithms to construct Voronoi diagram, we first review some existing models of parallel computation in this section.

1.1.1 Parallel Random Access Machines

In parallel random access machine (PRAM), a common memory is used as a bulletin board and all data exchanges are executed through it. Any pair of processors can communicate through this shared memory in constant time. As shown in Fig. 1.1, an interconnection unit (IU) allows each processor to establish a path to each memory location for the purpose of reading or writing.

The processors operate synchronously and each step of a computation consists of three phases:

1. The *read* phase, in which the processors read data from memory;
2. The *compute* phase, in which arithmetic and logic operations are performed;
3. The *write* phase, in which the processors write data to memory.

Depending on whether two or more processors are allowed to read from and/or write to the same memory location simultaneously, three submodels of the PRAM are identified:

1. The exclusive-read exclusive-write (EREW) PRAM, where both read and write accesses by more than one processor to the same memory location are not allowed.

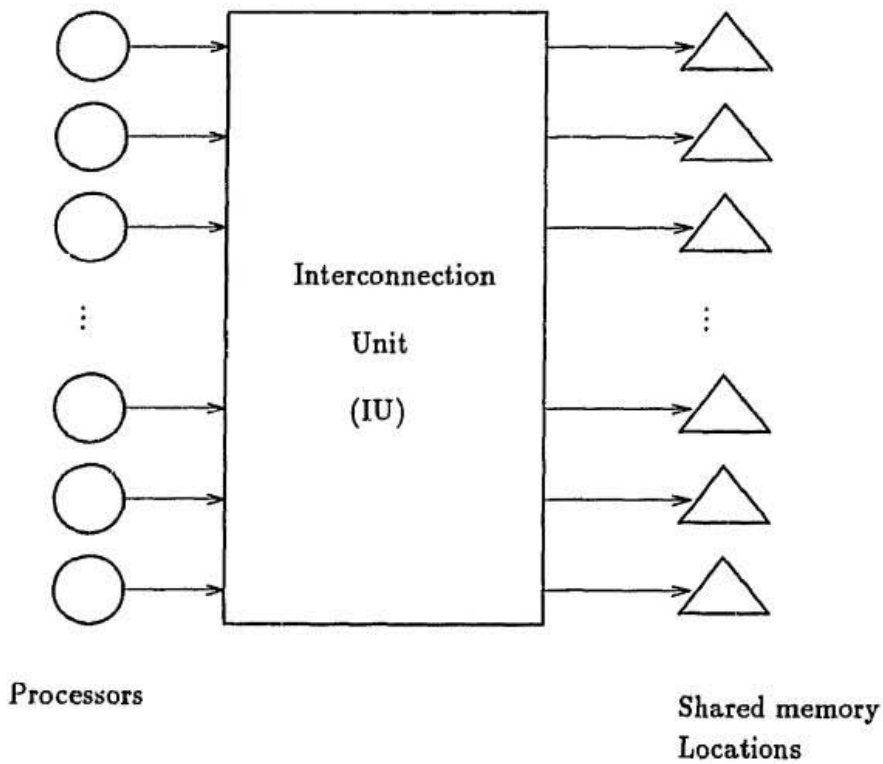


Figure 1.1: Parallel Random Access Machine

2. The concurrent-read exclusive-write (CREW) PRAM, where simultaneous reading from the same memory location is allowed, but not simultaneous writing.

3. The concurrent-read concurrent-write (CRCW) PRAM, where both forms of simultaneous access are allowed.

1.1.2 Processor Networks

In a processor network, an interconnected set of processors cooperate to solve a problem by performing local computations and exchanging messages. Several of the most widely used networks are outlined below.

1. **Mesh or Two-Dimensional Array**

   A two-dimensional network is obtained by arranging the n processors into an $m \times m$ array, where $m = \sqrt{n}$. The processor in row j and column k is denoted by (j, k), where $0 \leq j \leq m-1$ and $0 \leq k \leq m-1$. A two-way communication line links (j, k) to its neighbors $(j+1, k), (j-1, k), (j, k+1)$, and $(j, k-1)$. Processors on the boundary rows and columns have fewer than four neighbors and hence fewer connections. Such a network is also known as the *mesh* or the *mesh-connected computer* (MCC) model. Fig. 1.2 shows a mesh with $n = 16$ processors.

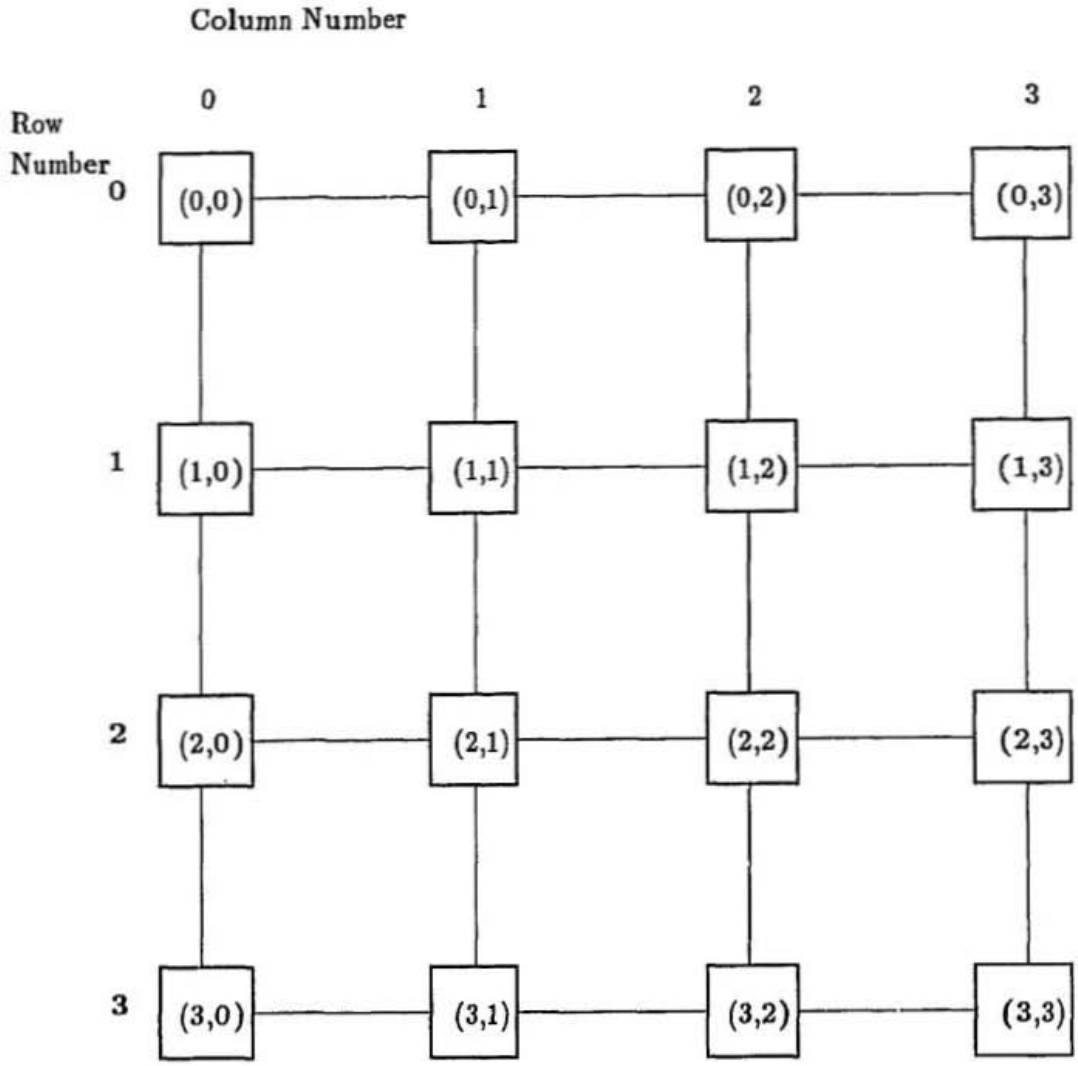


Figure 1.2: Mesh with $n = 16$ processors

2. **Cube-Connected Cycles**

To obtain a cube-connected cycles (CCC) network, we begin with a d-dimensional hypercube, then replace each of its 2^d corners with a cycle of d processors. Each processor in a cycle is connected to a processor in a neighboring cycle in the same dimension. See Fig. 1.3 for an example of a CCC network with $d = 3$ and $n = 2^d \cdot d = 24$ processors. In the figure, each processor has two indices i, j, where i is the processor order in cycle j.

3. **Stars and Pancakes**

Star and pancake are two interconnection networks with the property that for a given integer η, each processor corresponds to a distinct permutation of η symbols, say $\{1, 2, ..., \eta\}$. In other words, both networks connect $n = \eta!$ processors, and each processor is labeled with the permutation to which it corresponds. Thus, for $\eta = 4$, a processor may have the label 2134. In the *star network*, denoted by S_η, a processor v is connected to a processor u if and only if the label of u can be obtained from that of v by exchanging the first symbol with the i^{th} symbol, where $2 \leq i \leq \eta$. Thus for $\eta = 4$, if $v = 2134$ and $u = 3124$, u and v are connected by a two-way link in S_4, since 3124 and 2134 can be obtained from one another by exchanging the first and third symbols. Fig. 1.4 shows S_4.

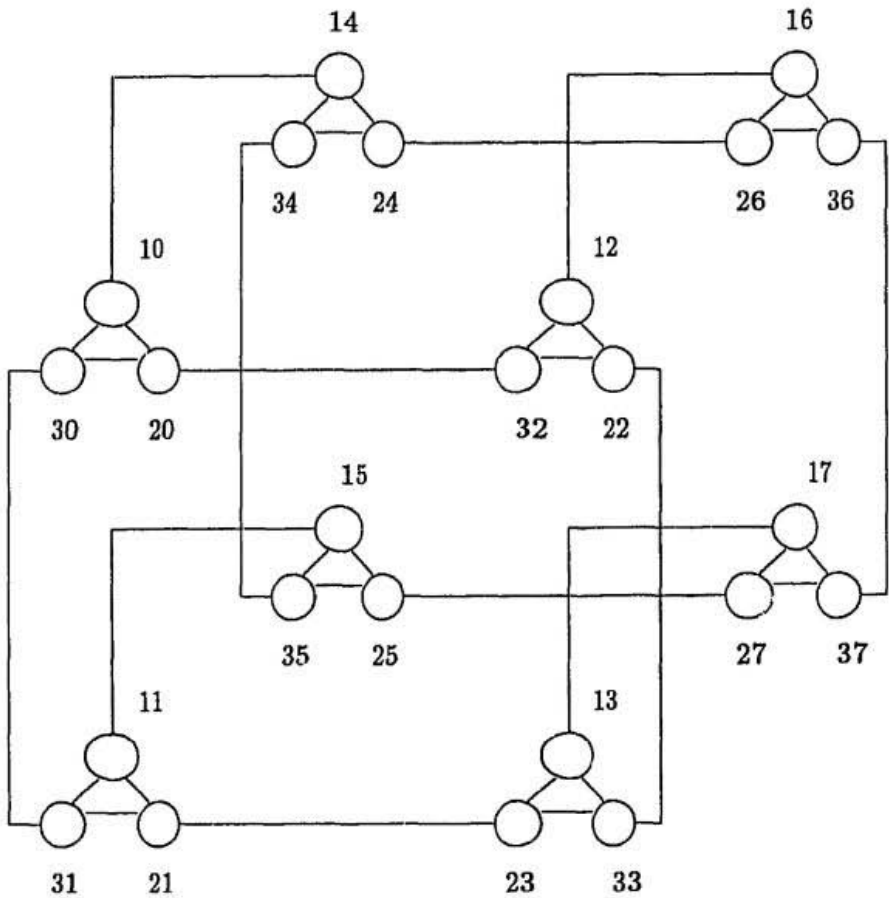


Figure 1.3: Cube-connected cycles network with $d = 3$ and $n = 24$

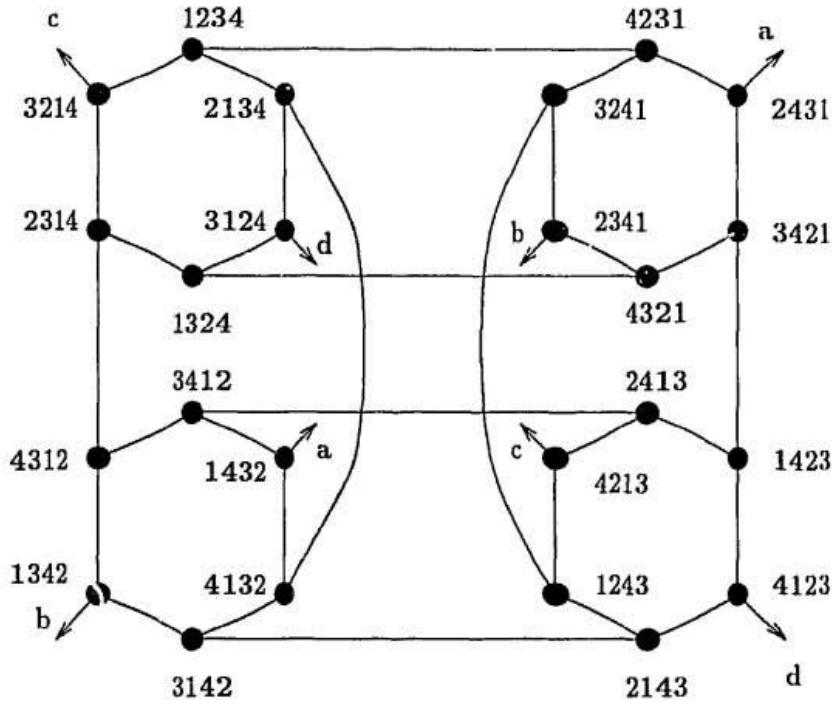


Figure 1.4: A 4-star

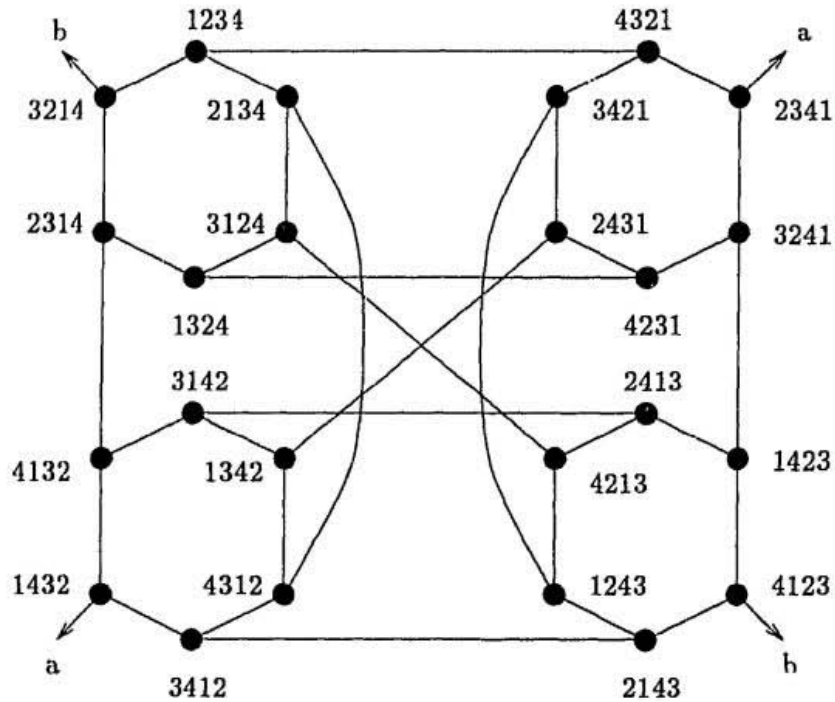


Figure 1.5: A 4-pancake

In the *pancake network*, denoted by P_η, a processor v is connected to a processor u if and only if the label of u can be obtained from that of v by flipping the first i symbols, where $2 \leq i \leq \eta$. Thus for $\eta = 4$, if $v = 2134$ and $u = 4312$, u and v are connected by a two-way link in P_4, since 4312 can be obtained from 2134 by flipping the four symbols, and vice versa. Fig. 1.5 shows P_4.

1.2 Literature Review of Parallel Algorithms for Constructing Voronoi Diagram

In 1980, Chow in her Ph.D thesis [5] proposed three parallel algorithms for constructing Voronoi diagrams. Her algorithms were based on Brown's method [4]. By using the inversion technique, she transformed the construction of the $2-d$ Voronoi diagram to the construction of the $3-d$ convex hull. The computation models she used were CREW PRAM and CCC parallel computer networks. On the CREW PRAM model, her algorithm runs in $\mathcal{O}(\log^3 n)$ time with $\mathcal{O}(n)$ processors. On the CCC model, one of her algorithms runs in $\mathcal{O}(\log^4 n)$ time and uses $\mathcal{O}(n)$ processors, and the other runs in $\mathcal{O}(k\log^3 n)$ time and uses $\mathcal{O}(n^{1+1/k})$ processors, where, $1 \leq k \leq \log n$.

In 1986, Mi Lu [6] presented a parallel algorithm to construct the Voronoi diagram of a set of planar points. The algorithm is based on Brown's approach [4] and has $\mathcal{O}(\sqrt{n}\log n)$ time complexity on $\mathcal{O}(\sqrt{n}\times\sqrt{n})$ MCC with constant storage per processors.

In 1988, Preilowski and Mumbeck [7] suggested a time-optimal parallel algorithm to compute Voronoi diagrams. The algorithm runs in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n^3)$ processors on a CREW PRAM. This result is time-optimal because the sorting problem can be reduced to the Voronoi diagram problem. The authors showed how their algorithm can compute all edges of the Voronoi

diagram in $\mathcal{O}(1)$ time with $\mathcal{O}(n^4)$ processors.

In 1988, Levcopoulos, Katajainen and Lingas [8] gave a parallel algorithm for computing the Voronoi diagram of a planar point set within a square window W. The algorithm uses multilevel bucketing and runs in $\mathcal{O}(\log n)$ average time on CRCW PRAM with $\mathcal{O}(n/\log n)$ processors when the points are drawn independently from a uniform distribution.

In 1989, Evans and Stojmenvoic [9] presented an $\mathcal{O}(\log^3 n)$ algorithm for constructing the Voronoi diagram of a set of n points on CREW PRAM. On the CRCW PRAM model the algorithm runs in $\mathcal{O}(\log^2 n)$ time. The algorithm uses Shamos's divide-and-conquer method [2].

In 1990, Chang-Sung Jeong [10] gave a parallel time optimal algorithm which runs in $\mathcal{O}(\sqrt{n})$ on an $\sqrt{n} \times \sqrt{n}$ mesh. The algorithm is based on the divide-and-conquer approach used by Shamos [2]. The set of points is sorted by x-coordinate and divided in half into two sets L and R by a vertical separating line l such that points in L are to the left of l and points in R are to the right of l. Recursively, the Voronoi diagram $Vor(L)$ and $Vor(R)$ are computed for the sets L and R, respectively. The two diagrams are then merged, resulting in $Vor(L \cup R)$. $Vor(L)$ is the Voronoi diagram of L, $Vor(R)$ is the Voronoi diagram of R and $Vor(L \cup R)$ is the Voronoi diagram of $L \cup R$.

In 1992, S.G Akl [11] presented parallel algorithms that compute the Voronoi Diagram of p points on the star and pancake network computers. For an n-star or n-pancake with $p = n!$ processors, given p planar points stored in the processors such that each processor holds one point and has a memory of constant size, the Voronoi diagram of these points can be found in $\mathcal{O}(n \log^2 n)$ time.

In this Chapter, we review parallel computation models and parallel algorithms of constructing Voronoi diagrams. The most popular processor interconnection topology today is undoubtedly the hypercube. The hypercube is one of the most versatile and efficient networks thus far discovered for parallel computation [12] because:

- In a hypercube, using d connections per processor, 2^d processor may be interconnected such that the maximum distance between any two processors is d. While linear array, tree, Mesh and Mesh of tree use a smaller number of connections per processor, the maximum distance between processors is larger.

- Most other popular networks are easily mapped into a hypercube. In particular, the n-node hypercube can simulate any $\mathcal{O}(n)$-node array, tree, or mesh of trees with only a small constant factor slowdown [12].

- Hypercube has the advantage of being a well studied network. Efficient algorithms are known for routing messages between processors in a

hypercube. A large corpus of software and programming techniques exist for hypercubes [13].

- A hypercube is completely symmetric. Every processor's interconnection pattern is like that of every other processor. Furthermore, a hypercube is completely decomposable into sub-hypercubes (i.e., hypercubes of smaller dimension). This property makes it relatively easy to implement recursive divide-and-conquer algorithms on the hypercube [14].

In a hypercube connected computer networks, local computations as well as message exchanges are taken into consideration when analyzing the time taken by the processor networks to solve a problem. When designing an algorithm for a processor network , the routing of messages from one processor to another is the responsibility of the algorithm designer. In Chapter 2, we will introduce the parallel computation model used in the thesis. Based on Nassimi and Sahni's paper [15], several fundamental operations on hypercube connect computer networks are described. The corresponding programs are also given. All those operations will be used in chapter 3. In Chapter 3, Based on Brown's method, we will develop a parallel algorithm to construct Voronoi diagrams. Our algorithm runs in $\mathcal{O}(\log^3 n)$ time on an $\mathcal{O}(n)$-processor hypercube connected computer network. Our algorithm is based on Brown's method which transforms the problem of construction of a planar a Voronoi diagram for an n-point set to construction of a convex hull of n points in three dimensional space. Comparing with the parallel algo-

rithms which are based on the divide-and-conquer approach used by Shamos, our algorithm can be used to solve two computational geometry problems: constructing 2−dimensional Voronoi diagram and 3− dimensional convex hull. Comparing with Chow's methods[5] which runs on a $\mathcal{O}(n)$ processors CCC (Cube-Connected Cycles) model has $\mathcal{O}(\log^4 n)$ time complexity, our algorithm has less time complexity. Comparing with Chang-Sung Jeong's algorithm[10] which runs in $\mathcal{O}(\sqrt{n})$ on an $\sqrt{n} \times \sqrt{n}$ mesh, our parallel computation model is more general. Most other popular networks can be easily mapped onto a hypercube[12].

Chapter 2

Hypercube Connected Computer Network and Its Fundamental Operations

In this chapter, the SIMD hypercube connected computer network will be defined. Some fundamental operations on it will be described and corresponding programs will also be given.

2.1 SIMD Hypercube Connected Computer Network

According to Michael J. Flnn's [16] taxonomy of computer architecture, parallel computers are divided into two categories, single instruction multiple data streams (SIMD) and multiple instruction multiple data streams

(MIMD). A block diagram for a SIMD computer is given in Figure 2.1.

As can be seen, an SIMD computer consists of n processing elements(PE's). The PE's are indexed 0 through $n-1$ and may be referenced as $PE(i)$. Each PE has its local memory. The PE's are synchronized and operate under the control program. PE's may be enabled or disabled so that the common instruction for any given time-unit is executed only on enabled PE's. This enabling and disabling of PE's can be done without the use of separate control lines for each PE as long as each PE knows its own index [17]. The PE's are connected together via an interconnection network. Different interconnection networks lead to different SIMD architectures. In this thesis, hypercube connect SIMD computers are considered.

Assume that $n = 2^d$ and let $i_{d-1} \cdots i_0$ be the binary representation of i for $i \in [0, n-1]$. Let $i^{(b)}$ be the number whose binary representation is $i_{d-1} \cdots i_{b+1} \overline{i_b} i_{b-1} \cdots i_0$, where $\overline{i_b}$ is the complement of i_b and $0 \leq b < d$. That is, $i^{(b)}$ is obtained by complementing the b'th bit of i's binary representation. In the hypercube model processor i is connected to processors $i^{(b)}$, $0 \leq b < d$. Fig. 2.2 shows an example of $n = 16$ processor hypercube.

The hypercube is an excellent (and popular) choice for the architecture of a multipurpose parallel machine. In this thesis, we choose the SIMD hypercube connected computer networks as our parallel computation model.

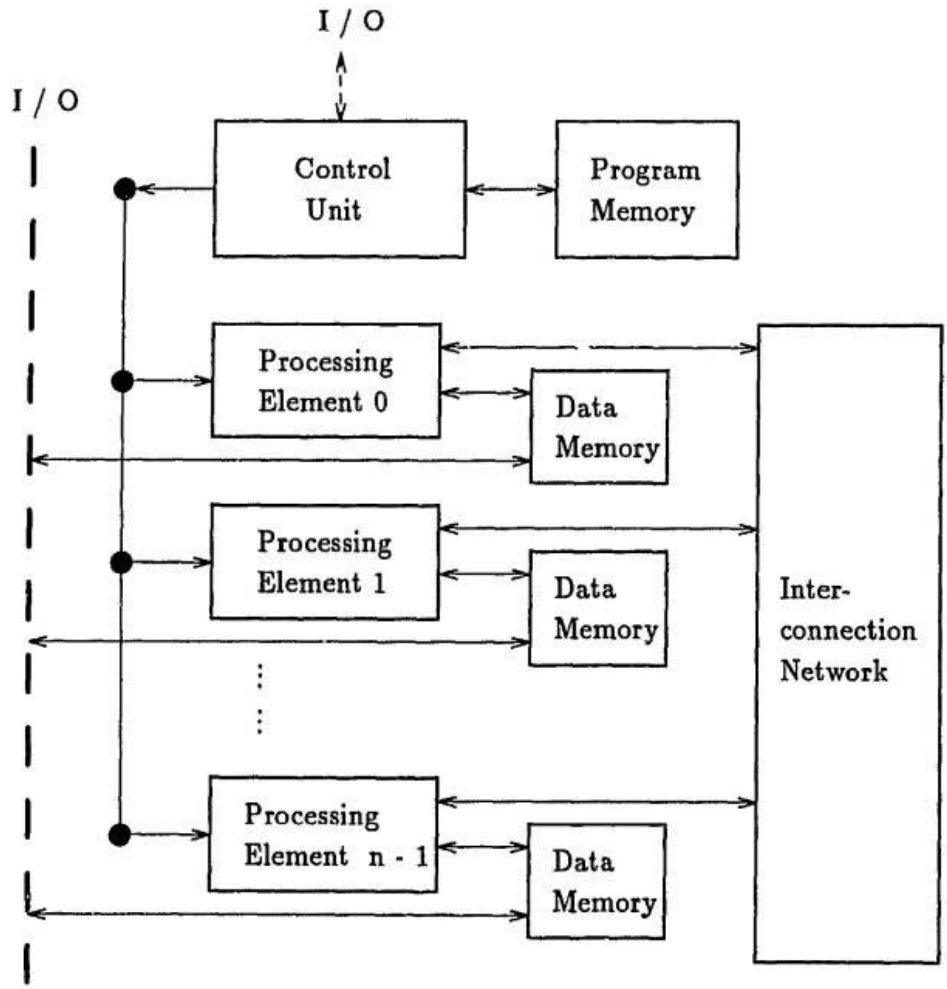


Figure 2.1: Block diagram of an SIMD computer

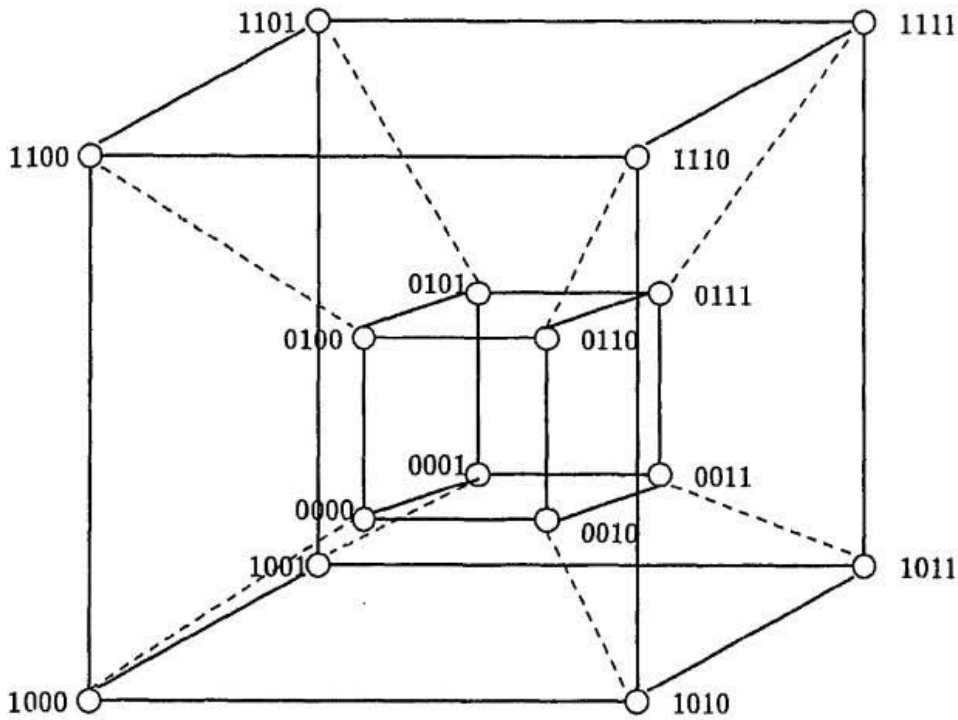


Figure 2.2: 16 processor hypercube

2.2 Fundamental Operations on Hypercube Connected Computer Network

In this section, based on Nassimi and Sahni's paper[15], several fundamental operations on hypercube connected computer networks are descriptively discussed. The derivations and the corresponding programs are also given after the description of each operation. All those operations will be used in chapter 3. Therefore, this section can be considered as the preparation for chapter 3. Only operations which are used will be discussed. Based on this, some very basic operations, such as data accumulation and consecutive sum on hypercube connected computer network, will not be mentioned in this section. For more details, [18] would be a good reference.

Before the fundamental operations on hypercube connected computer networks are introduced, some programming notation used is given as follows:

1. The notation $i^{(b)}$ is to represent the number that differs from i in exactly bit b. The square brackets ([ ]) are used to index an array and the parentheses ('( )') are used to index PEs. Thus $A[i]$ refers to the i'th element of array A and $A(i)$ refers to the A register of PE i. $A[j](i)$ refers to the j'th element of array A in PE i. The local memory in each PE holds data only (*i.e.*, no executable instructions). PEs need to be

able to perform only the basic arithmetic operations.

2. There are a separate program memory and a control unit. The control unit performs instruction sequencing, fetching, and decoding. In addition, instructions and masks are broadcasted by the control unit to the PEs for execution. An *instruction mask* is a boolean function used to select certain PEs to execute an instruction. For example, in the instruction

$$A(i) := A(i) + 1, (i_0 = 1)$$

$(i_0 = 1)$ is a mask that selects only those PEs whose index has bit 0 equal to 1.

3. Interprocessor assignments are denoted using the symbol " $\longleftarrow$", while intraprocessor assignments are denoted using the symbol " :=". Thus the assignment statement:

$$B(i^{(2)}) \leftarrow B(i), (i_2 = 0)$$

on a hypercube is executed only by the processors with bit 2 equal to 0. These processors transmit their B register data to the corresponding processors with bit 2 equal to 1.

4. A d-dimensional hypercube can be partitioned into windows of size 2^k processors each. Assume that this is done in such a way that the processor indices in each window differ only in their least significant k

bits. As a result, the processors on each window form a subhypercube of dimension k.

2.2.1 Maximum

Assume that a dimension d hypercube is partitioned into subhypercubes (or windows) of dimension k, where $P = 2^d$ and $W = 2^k$. If $l = iW + q$ $(0 \leq q \leq W)$ is a processor index, then processor l is the q'th processor in window i. This processor is to compute the maximum element of all elements, where, $0 \leq i < P/W$, $0 \leq q < W$.

The maximum relative to the whole size W window are obtained as below:

1. If a processor is in the left 2^{k-1} subwindow, then its maximum is unchanged.

2. The maximum of a processor in the right subwindow is its maximum when considered as a member of a 2^{k-1} window compared to the maximum of the A values in the left subwindow.

Table 2.1 gives an example maximum computation . The number of processors and the window size $W = 2^k$ are both 8. Line 0 gives the initial A values. The maximum in the current windows are stored in the S registers and the maximum of the A values of the processors in the current windows

are stored in the T registers. We begin with windows of size 1. The initial S and T values are given in lines 1 and 2, respectively. Next, the S and T values for windows of size 2 are obtained. These are given in lines 3 and 4. Line 5 and 6 give the S and T values when the window size is 4 and lines 9 and 10 give these values for the case when the window size is 8. The program in table 2.2 is the resulting procedure. Its time complexity is $\mathcal{O}(k)$.

2.2.2 Ranking

Associated with processor, i, in each size 2^k window of a hypercube is a flag *selected*(i) which is true iff this is a selected processor. The objective of ranking is to assign to each selected processor a *rank* such that *rank*(i) is the number of selected processors of the window with index less than i. Line 0 of Table 2.3 shows the selected processors in a window of size eight with an $*$. The ranks to be computed are shown in line 1.

The ranks of the selected processors in a window of size 2^k can be computed easily if we know the following information for the processors in each of the size 2^{k-1} subwindows that comprise the size 2^k window:

1. Rank of each selected processor in the 2^{k-1} subwindows
2. Total number of selected processors in each 2^{k-1} subwindow

If a processor is in the left 2^{k-1} subwindow then its rank in the 2^k

PE / line	0	1	2	3	4	5	6	7	
0	2	4	3	1	5	2	8	1	A
1	2	4	3	1	5	2	8	1	S
2	2	4	3	1	5	2	8	1	T
3	2	4	3	3	5	5	8	8	S
4	4	4	3	3	5	5	8	8	T
5	2	4	4	4	5	5	8	8	S
6	4	4	4	4	8	8	8	8	T
7	2	4	4	4	5	5	8	8	S
8	8	8	8	8	8	8	8	8	T

Table 2.1: Example to compute maximum in an SIMD hypercube

```
procedure SIMDMaximum(A, k, S);
{Compute the Maximum of A in windows of size 2^k}
begin
    {Initialize for size 1 windows}
    S(i) := A(i); T(i) := A(i);
    {compute for size 2^(b+1) windows}
    for b := 0 to k - 1 do
    begin
        B(i^(b)) ← T(i);
        S(i) := B(i), (S(i) < B(i) and i_b = 1);
        T(i) := B(i), (T(i) < B(i));
    end;
end; {of SIMDMaximum}
```

Table 2.2: Program for SIMD Maximum

window is the same as its rank in subwindow. If it is in the right subwindow, its rank is its rank in the subwindow plus the number of selected processors in the left subwindow. Line 2 of Table 2.3 shows the rank of each selected processor relative to subwindows of size 4. Line 3 shows the total number of selected processors in each subwindow.

Let $R(i)$ and $S(i)$, respectively, denote the rank of processor i (if it is a selected processor) and the number of selected processors in the current window that contains processor i. The strategy to count ranks in windows of size 2^k is to begin with R and S for windows of size one and then repeatedly double the window size until reaching a window size of 2^k. For windows of size one, it is given:

PE / line	0	1	2	3	4	5	6	7	
0		*	*		*		*	*	
1	∞	0	1	∞	2	∞	3	4	R
2	∞	0	1	∞	0	∞	1	2	R
3	2	2	2	2	3	3	3	3	S
4	0	0	0	0	0	0	0	0	R
5	0	1	1	0	1	0	1	1	S
6	0	0	0	0	0	0	0	1	R
7	1	1	1	1	1	1	2	2	S
8	0	0	1	1	0	0	1	2	R
9	2	2	2	2	3	3	3	3	S
10	0	0	1	1	2	2	3	4	R
11	5	5	5	5	5	5	5	5	S

Table 2.3: Example to compute ranks in an SIMD hypercube

$$R(i) = 0$$
$$S(i) = \begin{cases} 1 & \text{if } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

Lines 4 and 5 of Table 2.3 give the initial R and S values. Lines 6 and 7 give the values for windows of size 2. Lines 8 and 9 give these for windows of size 4, and lines 10 and 11 give them for a widow size of 8. The ranks for the processors that are not selected may now be set to ∞ to get the configuration of line 1. The procedure to compute ranks is given in Table 2.4. This procedure is due to Nassimi and Sahni [15] and its complexity is readily seen to be $\mathcal{O}(k)$.

2.2.3 Concentration

In a data concentration operation, it begins with one record, G, in each of the processors selected for this operation. The selected processors have been ranked and the rank information is in a field R of the record. Assume the window size is 2^k. The objective is to move the ranked records in each window to the processor whose position in the window equals the record rank. Line 0 of Table 2.5 gives an initial configuration for an SIMD eight processor window. The records are shown as pairs with the second entry in each pair being the rank. It is assumed that the processors that are not

```
procedure rank(k);
{Compute the rank of selected processors in windows of size 2^k}
{SIMD hypercube}
begin
    {Initialize for size 1 windows}
    R(i) := 0;
    if selected(i)
            then S(i) := 1
            else S(i) := 0;

    {Compute for size 2^(b+1) windows}
    for b := 0 to k - 1 do
    begin
        T(i^(b)) ← S(i);
        R(i) := R(i) + T(i), (i_b = 1);
        S(i) := S(i) + T(i);
    end;
    R(i) := ∞, (not selected(i));
end; {of rank}
```

Table 2.4: Program for SIMD ranking procedure

line \ PE	0	1	2	3	4	5	6	7
0	(-, ∞)	(B, 0)	(-, ∞)	(D, 1)	(E, 2)	(-, ∞)	(G, 3)	(H, 4)
1	(B, 0)	(D, 1)	(E, 2)	(G, 3)	(H, 4)	(-, ∞)	(-, ∞)	(-, ∞)
2	(B,0)	(-, ∞)	(-, ∞)	(D, 1)	(E, 2)	(-, ∞)	(H, 4)	(G, 3)
3	(B, 0)	(D, 1)	(-, ∞)	(-, ∞)	(H, 4)	(-, ∞)	(E, 2)	(G, 3)

Table 2.5: Example to concentrate in SIMD hypercube

selected for the concentration operation have a rank of ∞. The result of the concentration is shown in line 1. Let B, D, E, H be represented individal records.

Data concentration can be done in $\mathcal{O}(k)$ time by obtaining the agreement between the bits of the destination of a record and its present location in the order 0, 1, 2, $\cdots$, $k-1$ [15]. For example, let us seek agreement on bit 0. Examining the initial configuration (line 0), it can be seen that the destination and present location of records B, G and H disagree on bit 0. To obtain agreement, these records with the records in neighbor processors along bit 0 are exchanged. This gives the configuration of line 2. Examining the bit 1 of destination and present location in line 2, it can also be seen that records D, E and H have a disagreement. Exchanging these records

```
procedure concentrate (G, k);
{Concentrate records G in selected processors. 2^k is the window size}
{R is the rank field of a record}
begin
    for b := 0 to k − 1 do
    begin
        F(i^(b)) ← G(i);
        G(i) ← F(i), ((G(i).R ≠ ∞ and (G(i).R)_b ≠ i_b))
                    or (F(i).R ≠ ∞ and (F(i).R)_b ≠ i_b)));
    end;
end; {of concentrate}
```

Table 2.6: Program for procedure to concentrate records

with their neighbors along bit 1 yields line 3. Finally, let us examine bit 2 of the destination and present location of records in line 3 and determine that records E and G need to be exchanged with their neighbors along bit 2. This results in the desired final configuration of line 1.

2.2.4 Distribution

Data distribution is the inverse of data concentration. It begins with records in processors $0, \cdots, r$ of a hypercube window of size 2^k. Each record has a destination $D(i)$ associated with it. The destinations in each window are such that $D(0) < D(1) \cdots < D(r)$. The record that is initially in processor i of the window is to be routed to the $D(i)$'th processor of the window. Note that r may vary from window to window. Line 0 of Table 2.7 gives

line \ PE	0	1	2	3	4	5	6	7
0	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)
1	(-, ∞)	(-, ∞)	(-, ∞)	(A, 3)	(B, 4)	(-, ∞)	(-, ∞)	(C, 7)
2	(A, 3)	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	(B, 4)	(C, 7)	(-, ∞)
3	(-, ∞)	(-, ∞)	(A, 3)	(-, ∞)	(-, ∞)	(B, 4)	(C, 7)	(-, ∞)

Table 2.7: Example to distribute in an SIMD hypercube

an initial configuration for data distribution in an eight processor window of an SIMD hypercube. Each record is represented as a tuple with the second entry being the destination. Line 1 gives the result of the distribution.

Since data distribution is the inverse of data concentration, it can be carried out by running the concentration procedure in reverse [15]. The result is the program in Table 2.8. Lines 2, 3, and 1 of Table 2.7 give the configurations for the example following the iterations $b = 2$, 1, and 0, respectively, shown in Table 2.8.

2.2.5 Generalization

The initial configuration for a generalization is similar to that for a data distribution. It begins with records, G, in processors 0, $\cdots$, r of a

```
procedure distribute (G, k);
{Distribute records G. 2^k is the window size}
begin
    for b := k - 1 downto 0 do
    begin
        F(i^(b)) ← G(i);
        G(i) ← F(i), ((G(i).D ≠ ∞ and (G(i).D)_b ≠ i_b))
                    or (F(i).D ≠ ∞ and (F(i).D)_b ≠ i_b)));
    end;
end; {of distribute}
```

Table 2.8: Program for procedure to distribute records

hypercube window of size 2^k. Each record, $G(i)$, has a high destination $G(i).H$ associated with it, $0 \leq i \leq r$. The high destinations in each window are such that $G(0).H < G(1).H < \cdots G(r).H$. Let $G(-1).H = 0$. The record which is initially in processor i of the window is to be routed to processors $G(i-1).H$, $G(i-1).H+1$, $\cdots$, $G(i).H$ of the window, $0 \leq i \leq r$. Note that r may vary from window to window. Line 0 of Table 2.9 gives an initial configuration for data generalization in an eight processor window of an SIMD hypercube. Each record is represented as a tuple with the second entry being the high destination. Line 1 gives the result of the generalization.

Data generalization is done by repeatedly reducing the window size by half [15]. When the window size is halved, it should be ensured that all records needed in the reduced window are present in that window. Beginning with a window size of eight and line 0 of Table 2.9, each processor sends its

PE / line	0	1	2	3	4	5	6	7	
0	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	G
1	(A, 3)	(A, 3)	(A, 3)	(A, 3)	(B, 4)	(C, 7)	(C, 7)	(C, 7)	G
2	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	F
3	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	(-, ∞)	(B, 4)	(C, 7)	(-, ∞)	F
4	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	(-, ∞)	(B, 4)	(C, 7)	(-, ∞)	G
5	(C, 7)	(-, ∞)	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	(-, ∞)	(B, 4)	F
6	(C, 7)	(-, ∞)	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	(-, ∞)	(-, ∞)	F
7	(A, 3)	(B, 4)	(A, 3)	(B, 4)	(C, 7)	(B, 4)	(C, 7)	(-, ∞)	G
8	(B, 4)	(A, 3)	(B, 4)	(A, 3)	(B, 4)	(C, 7)	(-, ∞)	(C, 7)	F

Table 2.9: Example to generalize in an SIMD hypercube

record to its neighbor processor along bit 2. The neighbor processor receives the record in F. Line 2 shows the F values following the transfer. Next, some F's and G's are eliminated. This is done by comparing the high destination of a record with the lowest processor index in the size four window that contains the record. If the comparison succeeds, then, the record is not needed in the size four window. Applying this elimination criterion to line 0 results in the elimination of no G. However, when the criterion is applied to the F's of line 2, $F(4)$ is eliminated and the configuration of line 3 will be given. At this point each window of size four has all the records needed in that window. The records are, however, in both F and G. To consolidate the required records into the G's alone, the following consolidation criterion is used:

$$\text{replace } G(i) \text{ by } F(i) \text{ in case } F(i).H < G(i).H$$

i.e., of the two records in a PE, the one with smaller high destination survives. Applying the consolidation criterion to lines 0 and 3 results in line 4.

Next, records are transferred along bit 1. The F values following this transfer are given in line 5. Following the application of the elimination criterion, the F value of line 6 is obtained. The G values are unchanged. When the consolidation criterion is applied, the G values are as in line 7. Line 8 shows the F values following a transfer along bit 0. The elimination criterion results in line 1. Procedure *generalize* shown in Table 2.10 implements the generalization strategy just outlined.

```
procedure generalize (G, k);
{Generalize records G. 2^k is the window size}
begin
    for b := k − 1 downto 0 do
    begin
        {Transfer to neighboring window of size 2^b}
        F(i^(b)) ← G(i);

        {Elimination criterion}
        G(i).H := ∞, (G(i).H < i − i_{b−1..0});
        F(i).H := ∞, (F(i).H < i − i_{b−1..0});

        {Consolidation criterion}
        G(i) := F(i), (F(i).H < G(i).H);
    end;
end; {of generalize}
```

Table 2.10: Program for procedure to generalize records

For a natural number i, $i_{n..m}$ is the number obtained from i by flipping all the bits in the binary representation of i from the nth to the mth place.

2.2.6 Merging and Unmerging

Given two sorted sequences each stored in a hypercube of size $n/2$, their merging can be done in $\mathcal{O}(\log n)$ time. Merging two sorted sequences can be done by bitonic sort.

A *bitonic sequence* is a nonincreasing sequence of numbers followed by a nondecreasing sequence. Either (or both) of these may be empty. The sequence has the form $x_1 \geq x_2 \geq \cdots \geq x_k \leq x_{k+1} \leq \cdots \leq x_n$, for some k, $1 \leq k \leq n$. The sequences 10, 9, 9, 4, 5, 7, 9; 2, 3, 4, 5, 8; 7, 6, 4, 3, 1; and 11, 2, 5, 6, 8, 9 are examples of bitonic sequences.

A *bitonic sort* is a process which sorts a bitonic sequence into either nonincreasing or nondecreasing order. Suppose we are given two sorted sequences $v_1 \leq v_2 \leq \cdots \leq v_l$ and $w_1 \leq w_2 \leq \cdots \leq w_m$. First reverse one of sequences. This can be done in $\mathcal{O}(\log n)$ time by doing the *reversing* procedure.

Suppose we have a revising sequence $\{v_1, v_2, \cdots, v_l\}$, we get sequence $\{v_l, v_{l-1}, \cdots, v_2, v_1\}$. By concatenating two sequences to obtain the bitonic sequence $v_l \geq v_{l-1} \geq \cdots \geq v_1$? $w_1 \leq w_2 \leq \cdots \leq w_m = x_1 \geq x_2 \geq \cdots \geq x_k$

```
procedure SIMDRevising(A, k);
{The sequence is stored in window of size 2^k } {Revising all the elements in
windows of size 2^k}
begin
    for b := 0 to k − 2 do
        A(j^(i)) ← A(j)
end; {of SIMDRevising}
```

Table 2.11: Program for SIMD Reversing

$\leq x_{k+1} \leq \cdots \leq x_n$ where $n = l + m$. The resulting bitonic sequence is then sorted using a bitonic sort to obtain the desired merged sequence. So, for example, if it is desired to merge the sequences (2, 8, 20, 24) and (1, 9, 10, 11, 12, 13, 30), the bitonic sequence (24, 20, 8, 2, 1, 9, 10, 11, 12, 13, 30) should be first created.

Batcher's bitonic sort [19] is ideally suitable for implementation on a hypercube computer. Batcher's algorithm to sort the bitonic sequence $x_1, \cdots, x_n$ into nondecreasing order is given in Table 2.12.

Example: Consider the bitonic sequence (24, 20, 8, 2, 1, 9, 10, 11, 12, 13, 30). Suppose we wish to sort this into nondecreasing order. The odd sequence is (24, 8, 1, 10, 12, 30) and the even sequence is (20, 2, 9, 11, 13). Sorting these, the sequences (1, 8, 10, 12, 24, 30) and (2, 9, 11, 13, 20) are obtained. Putting the sorted odd and even parts together, the sequence (1, 2, 8, 9, 10, 11, 12, 13, 24, 20, 30) is given. After performing the $\lfloor n/2 \rfloor$ compare/exchanges of step 3, the sorted sequence (1, 2, 8, 9, 10, 11, 12, 13,

Step 1:	[Sort odd subsequence] If $n > 2$ then recursively sort the odd bitonic subsequence $x_1, x_3, x_5, \cdots$ into nondecreasing order
Step 2:	[Sort even subsequence] If $n > 2$ then recursively sort the even bitonic subsequence $x_2, x_4, x_6, \cdots$ into nondecreasing order
Step 3:	[Compare/exchange] Compare the pairs of elements x_i and x_{i+1} for i odd and exchange them in case $x_i > x_{i+1}$

Table 2.12: Bitonic sort into nondecreasing order

20, 24, 30) is obtained.□

When n is a power of 2, the recursion in Table 2.12 can be unfolded to obtain the comparing/exchanging algorithm of the program in Table 2.13. In each iteration of the while loop, each sequence element is paired with exactly one other sequence element that is a distance d from it. The pairs are formed from left to right. To obtain a nondecreasing sequence each comparing/exchanging causes the smaller element of the pair to move to the left position. If a nonincreasing sequence is desired the smaller element is moved to the right. Table 2.14 shows an eight element bitonic merge that results in a nondecreasing sequence and Table 2.15 gives an example that results in a nonincreasing sequence. The examples assume the elements to be sorted are stored in processors of a hypercube with one element per processor. As can be seen, the elements that form each of the pairs for the comparing/exchanging operation are in processors that are hypercube neighbors. Hence each iter-

```
procedure BitonicSort(n);
{Sort the bitonic sequence x_1, ..., x_n}
{n is a power of 2}
begin
    d = n/2;
    while d > 0 do
    begin
        compare/exchange elements d apart
        d = d/2;
    end;
end; {of BitonicSort}
```

Table 2.13: Iterative bitonic sort for n, a power of 2

ation of the while loop of the program in Table 2.13 takes $\mathcal{O}(1)$ time on a hypercube. The total time to sort an n element bitonic sequence is therefore $\mathcal{O}(\log n)$.

Given a sorted sequence of elements so that part of the elements belong to a set A (thus the remaining belong to $\overline{A}$) and each element knows the corresponding rank in A or $\overline{A}$, permute the sequence to return each A and $\overline{A}$. This can be done by running the merging algorithm in reverse order Therefore, Unmerging can be done in $\mathcal{O}(\log n)$ time.

2.2.7 Sorting

To sort n elements using bitonic sort, we begin with sorted sequences of size one. Adjacent pairs of these form bitonic sequences that are sorted (in parallel) to obtain sorted sequences of size two. The sorting is done such

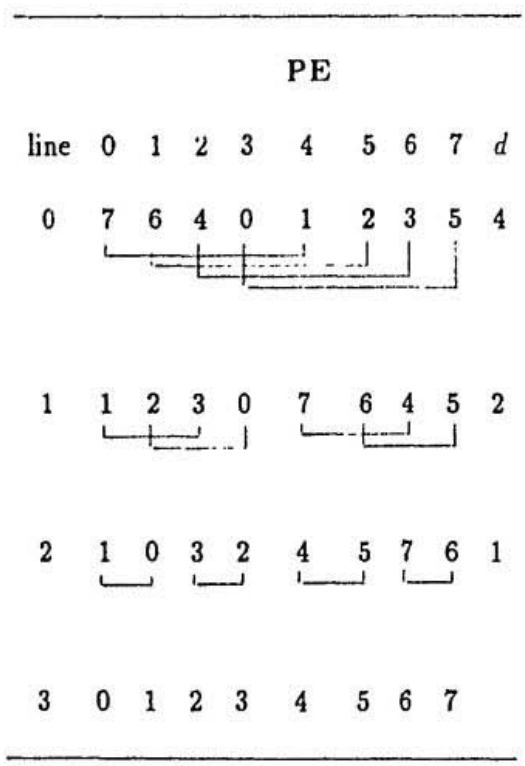


Table 2.14: Power of 2 bitonic sort (nondecreasing order)

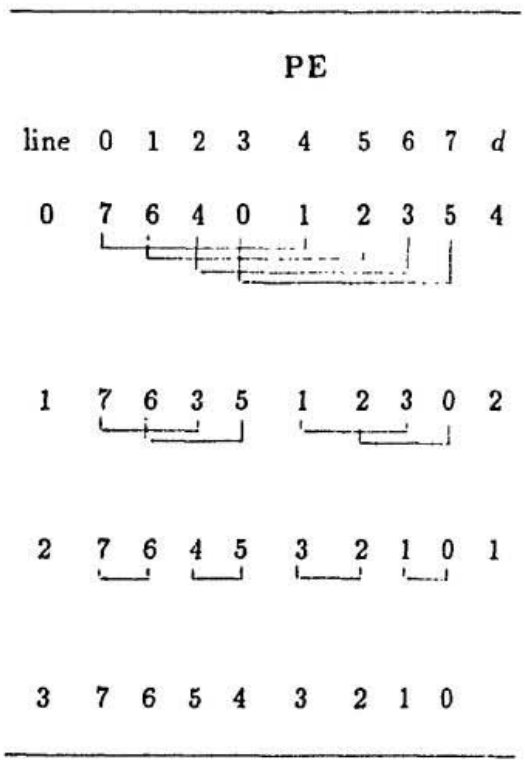


Table 2.15: Power of 2 bitonic sort (nonincreasing order)

that the size two sequences are alternately nonincreasing and nondecreasing sequences (i.e, the first, third, fifth, ... , sequences are nonincreasing and the remainder are nondecreasing). Consequently every pair of adjacent size two sequences forms a bitonic sequence of size four which can be sorted using bitonic sort. The size four sequences are also sorted alternately into nonincreasing and nondecreasing order. Continuing in this way, we can obtain a sorted sequence of size n after $\log n$ bitonic sorting steps. Note that if the sorted sequence is to be in nondecreasing order, then the last bitonic sort step should sort the first and only resulting sequence into this order. The total time for the sorting is $\mathcal{O}(\log^2 n)$.

Example: Sorting following sequence

c n m f h a p d g j l k b e i o

into nondecreasing order and that a < b < $\cdots$ < o < p. The pairs (c n), (m f), (h a), (p d), (g j), (l k), (b e) and (i o) are bitonic sequences that are sorted by using bitonic sort to obtain the sequence:

n c f m h a d p j g k l e b i o

Note that the odd pairs were sorted into nonincreasing order while the even ones were sorted into nondecreasing order. Next let us consider the adjacent sequences of length four. These are (n c f m), (h a d p), (j g k l) and (e b i o). Since each is a bitonic sequence, it may be sorted using bitonic sort. The result is:

n m f c a d h p l k j g b e i o

Once again the odd sequences are sorted into nonincreasing order while the even ones are sorted into nondecreasing order. Two bitonic sequences of length eight, (n m f c a d h p) and (l k j g b e i o), are given. Sorting them will give the sequence:

p n m h f d c a b e g i j k l o

Sorting it into nondecreasing order results in the sequence:

a b c d e f g h i j k l m n o p

□

2.2.8 Random Access Read

In a random access read (RAR), some of the processors of the hypercube wish to read data from other processors of the hypercube. Let $\Lambda(i)$ be the PE from which processor i wishes to get data. The data to be obtained is $D(\Lambda(i))$. In case PE i does not wish to read data from any other PE, then $\Lambda(i) = \infty$. Line 0 of Table 2.16 gives the Λ values for an example RAR in an eight processor hypercube. Note that in an RAR, several processors may read from the same PE. An RAR can be done in $\mathcal{O}(\log^2 n)$ time in an n processor hypercube using the algorithm of the program shown in Table 2.17 [15].

	PE								
step	0	1	2	3	4	5	6	7	
0	4	3	∞	1	7	7	∞	3	A
1	(4,0,t)	(3,1,t)	(∞,2,t)	(1,3,t)	(7,4,t)	(7,5,t)	(∞,6,t)	(3,7,t)	
2	(1,3,t)	(3,1,f)	(3,7,t)	(4,0,t)	(7,4,f)	(7,5,t)	(∞,2,f)	(∞,6,t)	sort
3	0		1	2		3			rank
4	(0,1,0)		(1,3,2)	(2,4,3)		(3,7,5)			
5	(0,1,0)	(1,3,2)	(2,4,3)	(3,7,5)					concentrate
6	(0,1)	(1,3)	(2,4)	(3,7)					
7		(0,1)		(1,3)	(2,4)			(3,7)	distribute
8		(0,D(1))		(1,D(3))	(2,D(4))			(3,D(7))	
9	(0,D(1))	(1,D(3))	(2,D(4))	(3,D(7))					concentrate
10	(0,D(1))	(2,D(3))	(3,D(4))	(5,D(7))					
11	(0,D(1))	(2,D(3))	(2,D(3))	(3,D(4))	(5,D(7))	(5,D(7))			generalize
12	(3,D(1))	(1,D(3))	(7,D(3))	(0,D(4))	(4,D(7))	(5,D(7))	(2,-)	(6,-)	
13	(0,D(4))	(1,D(3))	(2,-)	(3,D(1))	(4,D(7))	(5,D(7))	(6,-)	(7,D(3))	sort

Table 2.16: Example of a random access read

Step 1: Each processor a creates a triple $(A(a), a, flag)$ where $flag$ is a Boolean entity that is initially true.

Step 2: [Sort] Sort the triples into nondecreasing order of the read address $A(a)$. Triples with the same read address are in nondecreasing order of the PE index a. Furthermore, during the sort, the $flag$ entry of a triple is set to false in case there is a triple to its right with the same read address.

Step 4: [Rank] Processor with triples whose first component $\neq \infty$ and whose third component (*i.e.*, $flag$) is true are ranked.

Step 4: Each processor b that has a triple $(A(a), a$,true) with $A(a) \neq \infty$ creates a triple of the form $(R(b), A(a), b)$ where $R(b)$ is the rank computed in the preceding step.

Step 5: [Concentrate] The triples just created are concentrated.

Step 6: Each processor c that has a concentrated triple $(\Gamma(b), A(a), b)$ creates a tuple of the form $(c, A(a))$. Note that since $c = R(b)$ this tuple is just the first two components of the triple.

Step 7: [Distribute] The tuples are distributed using the second component as the destination address.

Step 8: Each processor $A(a)$ that receives a tuple $(c, A(a))$ creates the tuple $(c, D(A(a)))$.

Step 9: [Concentrate] The tuples created in the preceding step are concentrated using the first component as the rank.

Step 10:	Each processor c that received a tuple $(c, D(A(a)))$ in the last step also has a triple of the form $(R(b), A(a), b)$ that it received in Step 5 (notice that $c = R(b)$). Using this triple and the tuple received in Step 9 it creates the triple $(b, D(A(a)))$.
Step 11:	[Generalize] The tuples $(b, D(A(a)))$ are generalized using the first component as the high destination.
Step 12:	Each processor that received a tuple $(b, D(A(a)))$ in Step 11 also has a triple $(A(a), a, flag)$ that it at obtained as a result of the sort of Step 2. Using information from the tuple and the triple at creates a new tuple $(a, D(A(a)))$. Processors that it did not receive a tuple use the triple they received in Step 2 and form the tuple $(a, -)$.
Step 13:	[Sort] The newly created tuples of Step 12 are sorted by their first component.

Table 2.17: Algorithm for a random access read

Consider the example of Table 2.16. In Step 1, each processor creates a triple with the first component being the index of the processor from which it wants to read data; the second component is its own index; and the third component is a flag that is initially true(t) for all triples. Then, in Step 2 the triples are sorted on the first component. Triples that have the same first component are in increasing order of their second component. Within each sequence of triples that have the same first component only the last one has a true flag. The flag for the remaining triples is false. The first components of the triples with a true flag give all the distinct processors from which data is to be read.

Processors 0, 2, 3, and 5 are ranked in Step 3. Since the highest rank is three, data is to read from only four distinct processors. In Step 4, the ranked processors create triples of the form $(R(b), A(a), b)$. The triples are then concentrated. Processors 0 through 3 receive the concentrated triples and form tuples of the form $(c, A(a))$. Because of the sort of Step 2, the second components of these tuples are in ascending order. Hence, they can be routed to the processors given by the second component using a data distribution as in Step 7. The destination processors of these tuples are the distinct processors whose data is to be read. These destination processors create, in Step 8, tuples of the form $(c, D(A(a)))$ where c is the index of the processor that originated the tuple it received. These tuples are concentrated in Step 9 using the first component as the rank.

In Step 10, the receiving processors (*i.e.*, 0 through 3) use the triples received in Step 5 and the tuples received in Step 9 to create tuples of the form $(b, D(A(a)))$. The first component is the index of the processor that originated the triple received in Step 5. Since the triples received in Step 5 are the result of a concentration, the first component of the newly formed tuples are in ascending order. The tuples are therefore ready for generalization using the first component as the high index. This is done in Step 11. After this generalization we have the right number of copies of each data. For example, two processors (4 and 5) wanted to read from processor 7 and we now have two copies of $D(7)$. Comparing the triples of Step 2 and the tuples of Step 11, we see that the second component of the triples tells us where the data in the tuples is to be routed to. In Step 12 we create tuples that contain the destination processor and the data. Since the destination addresses are not in ascending order the tuples cannot be routed to their destination processors using a distribute. Rather, they must be sorted by destination.

2.2.9 Random Access Write

A random access write (RAW) is like a random access read except that processors wish to write to other processors rather than to read from them. A random access write uses many of the basic steps used by a random access read. It is, however, quite a bit simpler. Line 0 of Table 2.18 gives the

index $A(i)$ of the processor to which processor i wants to write its data $D(i)$. $A(i) = \infty$ when processor i is not to write to another processor. Observe that it is possible for several processors to have the same write address A. When this happens, it is said that the RAW has collisions. It is possible to formulate several strategies to handle collisions. Three of these are:

1. **Arbitrary RAW** of all the processors that attempt to write to the same processor exactly one succeeds. Any of these writing processors may succeed.

2. **Highest/lowest RAW** of all the processors that attempt to write to the same processor the one with the highest (lowest) index succeeds.

3. **Combining RAW** all the processors succeed in getting their data to the target processors.

Consider the example of line 0 of Table 2.18. In an arbitrary RAW any one of $D(0)$, $D(2)$ and $D(7)$ will get to processor 3. One of $D(1)$ and $D(5)$ will get to processor 0. And $D(3)$ and $D(4)$ will get to processors 4 and 6, respectively. In a highest RAW $D(7)$, $D(5)$, $D(3)$, and $D(4)$, respectively, get to processors 3, 0, 4, and 6, respectively. In a lowest RAW $D(0)$, $D(1)$, $D(3)$ and $D(4)$ get to processors 3,0,4,and 6, respectively. In a combining RAW $D(0)$, $D(2)$, and $D(7)$ all get to processor 3. Both $D(1)$ and $D(5)$ get to processor 0. And $D(3)$ and $D(4)$ get to processors 4 and 6, respectively.

	PE								
step	0	1	2	3	4	5	6	7	
0	3	0	3	4	6	0	∞	3	A
1	$(3, D(0), t)$	$(0, D(1), t)$	$(3, D(2), t)$	$(4, D(3), t)$	$(6, D(4), t)$	$(0, D(5), t)$	$(\infty, D(6), t)$	$(3, D(7), t)$	
2	$(0, D(1), f)$	$(0, D(5), t)$	$(3, D(0), f)$	$(3, D(2), f)$	$(3, D(7), t)$	$(4, D(3), t)$	$(6, D(4), t)$	$(\infty, D(6), t)$	sort
3		0			1	2	3		rank
4		$(0, D(5), 0)$			$(3, D(7), 1)$	$(4, D(3), 2)$	$(6, D(4), 3)$		
5	$(0, D(5), 0)$	$(3, D(7), 1)$	$(4, D(3), 2)$	$(6, D(4), 3)$					concentrate
6	$(0, D(5), 0)$			$(3, D(7), 1)$	$(4, D(3), 2)$		$(6, D(4), 3)$		distribute

Table 2.18: Example of an arbitrary random access write

The steps involved in an arbitrary RAW are given in Table 2.19 [15]. Let us go through the example in Table 2.18. Each processor first creates triples whose first component is the index, $A(a)$, of the processor to which it is to write. Its second component is the data, $D(a)$, to be written and the third component is true. The triples are then sorted on the first component. During this sort the flag entry of a triple is changed to false in case there is a triple with the same write address to its right. Only the triples with a true flag are involved in the remainder of the algorithm. Notice that for each distinct write address there will be exactly one triple with a true flag. The processors that have a triple with a true flag are ranked (Step 3) and these processors create new triples whose first and second components are the same as in the old triples but whose third component is the rank. The triples are then concentrated using this rank information. Since the triples are in ascending order of the write addresses (first component) they may be routed to these processors using a data distribute operation. Note that for Step 6 the third component (*i.e.*, rank) of each triple may be dropped before the distribute begins.

The complexity of a random access write is determined by the sort step which takes $\mathcal{O}(\log^2 n)$ time where n is the number of processors.

A highest (lowest) RAW can be done by modifying the program in Table 2.19 slightly. Step 1 creates 4-tuples instead of triples. The fourth

Step 1:	Each processor a creates a triple $(A(a), D(a), flag)$ where $flag$ is a Boolean entity that is initially true.
Step 2:	[Sort] Sort the triples into nondecreasing order of the write address $A(a)$. Ties are broken arbitrarily and during the sort the $flag$ entry of a triple is set to false in case there is a triple to its right with the same write address.
Step 3:	[Rank] Processors with triples whose first component is not ∞ and whose third component (*i.e.*, *flag*) is true are ranked.
Step 4:	Each processor b that has a triple $(A(a), D(a), \text{true})$ with $A(a) \neq \infty$ creates a triple of the form $(A(a), D(a), R(b))$ where $R(b)$ is the rank computed in the preceding step.
Step 5:	[Concentrate] the triples just created are concentrated.
Step 6:	[Distribute] the concentrated triples are distributed using the first component as the destination address.

Table 2.19: Algorithm for an arbitrary random access write

component is the index of the originating processor. In the sort step (Step 2) ties are broken by the fourth component in such a way that the right most 4-tuple in any sequence with the same write address is the 4-tuple to be succeeded (*i.e.*, highest of lowest fourth component in the sequence). Following this the fourth component may be dropped from each 4-tuple. The remaining steps are unchanged.

The steps for a combining RAW are also similar to those in the program shown in Table 2.19. When the ranking of Step 3 is done, a version of procedure *rank* (Table 2.14) is used, which does not contain the last line ($R(i) := \infty$,(**not** *selected*(i))). As a result processor 0 (Table 2.18) has a rank of 0 and processors 2 and 3 have a rank of 1. During the concentration step (Step 5) more than one triple will try to get to the same processor. Procedure *concentrate* (Table 2.14) is modified to combine together triples that have the same rank. These modifications do not change the asymptotic complexity of the RAW unless the combining operation increases the triple size (as in a concatenate). In case d data values are to reach the same destination, the complexity is $\mathcal{O}(\log^2 n + d \log n)$.

2.2.10 Precede

Given two sorted lists $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_n)$, we define the predecessor of a_i as the least element b_j in B previous to a_i in the sorted list of A and B, *i.e.* $b_j \leq a_i \leq b_{j+1}$. If there is no such b_j, we assume its

predecessor is the least element b_m in B. The Precede operation computes, for each element a_i in A, its predecessor in B. The Precede operation can be performed in the following way: For each element in list B, we assign a flag. Merging two sorted lists A and B, we can get a sorted list C. Performing rank operation, for each element a_i, we can get its predecessor in B. The Precede operation can be done in $\mathcal{O}(\log n)$ time.

2.2.11 Summary

In this subsection, we provide a summary. In the following, M and W are powers of 2. They represent the size (*i.e.*, the number of processors) in a subhypercube. Unless otherwise stated, the size of the full hypercube is denoted by P.

1. Maximum

   Task: This works on each dimension k, $k = \log W$, subhypercube of an SIMD hypercube. The hypercube PE $l = iW + q$, $0 \leq q < W$ is the q'th PE in the i'th dimension k subhypercube. This PE computes, in its S register, the maximum of the A register values of the 0'th though q'th PEs in its subhypercube, $0 \leq q < W, 0 \leq i < w$, where w is the number of subhypercubes of dimension k.

   Complexity: $\mathcal{O}(k)$.

2. Ranking

Task: Rank the selected processors in each size 2^k window of the SIMD hypercube.

Complexity: $\mathcal{O}(k)$.

3. Concentration

Task: Let $G(i).R$ be the rank of each selected processor i in the window of size 2^k that it is contained in. For each selected PE, i, the record $G(i)$ is sent to the $G(i).R$'th PE in the size 2^k window that contains PE i.

Complexity: $\mathcal{O}(k)$.

4. Distribution

Task: This is the inverse of a concentration.

Complexity: $\mathcal{O}(k)$.

5. Generalization

Task: Each record $G(i)$ has a high destination $G(i).H$. The high destinations in each size 2^k window are in ascending order. Assume that $G(-1).H = 0$. The record initially in processor i is routed to processors $G(i-1).H$ through $G(i).H$ of the window provided that $G(i).H \neq \infty$. If $G(i).H = \infty$, then the record is ignored. The procedure as written assumes a PE ordering that corresponds to that generally used for SIMD hypercubes.

Complexity: $\mathcal{O}(k)$.

6. **Merging and Unmerging**

   Task: Given two sorted sequences A and B, merging is a process that sorts them into either nonincreasing or nondecreasing order. Unmerging is the inverse of merging.

   Complexity: $\mathcal{O}(\log n)$. n is the total number of elements in sequences A and B.

7. **Sorting**

   Task: Given an element per processor, after sorting, the elements are kept in either nonincreasing or nondecreasing order.

   Complexity: $\mathcal{O}(\log^2 n)$. n is the total number of elements.

8. **Random Access Read**

   Task: Each PE in an n processor hypercube reads the A register data of some other PE in the hypercube.

   Complexity: $\mathcal{O}(\log^2 n)$

9. **Random Access Write**

   Task: Each PE in an n processor hypercube sends its A register data to the A register of some other PE in the hypercube.

   Complexity: $\mathcal{O}(\log^2 n)$.

10. **Precede**

Task: Given two sorted lists $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_n)$, we define the predecessor of a_i as the least element b_j in B previous to a_i in the sorted list of A and B, *i.e.* $b_j \leq a_i \leq b_{j+1}$. The Precede operation computes, for each element a_i in A, its predecessor in B.

Complexity: $\mathcal{O}(\log n)$

Chapter 3

A Parallel Algorithm for Constructing Voronoi Diagrams on a Hypercube Connected Computer Network

In this chapter, a parallel algorithm for constructing Voronoi diagrams on a hypercube connected computer will be introduced. Firstly, Section 3.1 gives the formal definition and important features of Voronoi diagram which are essential for understanding the algorithm. Secondly, Section 3.2 describes two major tools used in the algorithm — the convex hull and the inversion transform. Finally, the parallel algorithm for constructing Voronoi diagrams on a hypercube connected computer is discussed in Section 3.3.

3.1 Definition and Features of Voronoi Diagram

A Voronoi diagram (also called a Thiessen diagram) of a set S of n points is a well known structure which makes explicit some proximity information about S. More formally, given two points $p_i \in S$ and $p_j \in S$, define $H(p_i, p_j)$ as the half-plane containing p_i and bounded by the perpendicular bisector of p_i and p_j. Let the intersection of $n-1$ half-planes be $V(i) = \bigcup_{i \neq j} H(p_i, p_j)$, $j = 1, ..., n$, the Voronoi polygon of point p_i. $V(i)$ is a convex polygon with at most $n-1$ sides such that any point in $V(i)$ is closer to p_i than to any other point in S. The set of n Voronoi polygons defines the Voronoi diagram of S, $Vor(S)$. Some Voronoi polygons may be unbounded. Vertices of the Voronoi polygons are called Voronoi vertices and edges of the polygons are called Voronoi edges. Figure 3.1 is an example of the Voronoi Diagram.

Some important properties of the planar Voronoi diagram are given below:

1. Voronoi vertices are the center of circles defined through three points of S. These circles contain no other point of S.

2. A Voronoi polygon $V(i)$ is unbounded if and only if p_i is a point on the convex hull of S.

3. The straight-line dual of the Voronoi diagram is a triangulation of S

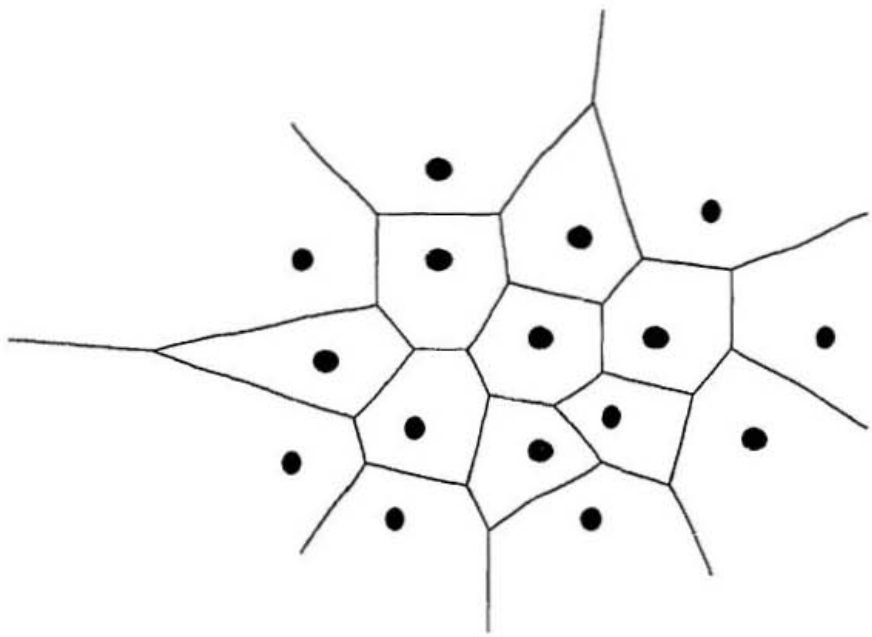


Figure 3.1: Voronoi diagram

called the *Delaunay Triangulation*. This triangulation of S has the property that the minimum angle of its triangles is maximum over all triangulations of S.

4. The Voronoi diagram of a set S of n points has $\mathcal{O}(n)$ vertices and $\mathcal{O}(n)$ edges by Euler's relation, namely $v-e+f=2$, where v, e, and f denote the number of vertices, edges, and regions of a planar subdivision.

3.2 Convex Hull and Inversion Transform

3.2.1 Convex Hull

The convex hull of a set of n points is defined as the smallest convex set which contains all of the points. In the plane, this is a convex polygon of at most n sides. In the three dimensional space, it is a convex polyhedron. A convex polyhedron is specified completely by its edges and faces. It is a crucial observation that the set of the edges of a convex polyhedron forms a planar graph: if we exclude degeneracies, it forms a triangulation, that is, each convex face is a triangle and has three adjacent faces. The convex hull has only $\mathcal{O}(n)$ faces F_i and edges e_{ij} because a planar graph of $n > 2$ vertices has at most $2n-4$ regions(faces) and at most $3n-6$ edges [20].

3.2.2 Inversion Transform

Given an inversion center P_0 and an inversion radius r, we can transform point Q to point Q' by inversion, where $\vec{P_0Q'}$ is in the same direction as $\vec{P_0Q}$ and $| \vec{P_0Q'} | = r^2 / | \vec{P_0Q} |$. The inversion has the following properties:

1. An inversion transforms a plane which does not pass through the inversion center to a sphere which passes through the inversion center, and *vice versa*.

2. The interior of the sphere corresponds to one of the half spaces bounded by the plane and the exterior of the sphere corresponds to the other half space.

3. The inversion is involutory. *i.e.* application of inversion twice yields the original point.

3.3 A Parallel algorithm to construct a Voronoi Diagram

In 1979, Brown demonstrated an interesting linkage between two dimension Voronoi diagrams and three dimension convex hulls. He presented a $\mathcal{O}(n \log n)$ sequential algorithm to construct the Voronoi diagram, by transforming the problem of constructing a planar Voronoi diagram for n points set to the construction of the convex hull of n points in 3-dimensional space *via*

a geometric transformation known as inversion. Based on Brown's method, a parallel algorithm for constructing Voronoi Diagram on hypercube connected computers will be introduced in the following sections.

The parallel algorithm is divided into four steps.

1. Perform the inversion for each of the points in the plane and get a new set of points in 3-dimensional space.
2. Construct the 3-dimensional convex hull for the new set of points.
3. Perform the inversion for each convex face, obtain a set of spheres which intersect the xy-plane to form a set of circles, determine the Voronoi vertices.
4. Construct the Voronoi diagram

Lemma 3.1 *Let n points be distributed on d-dimension a hypercube, where, $n = 2^d$. One point per PE. Inversion for each point can be done in constant time.*

Proof: Let S be a set of n planar points located in the xy-plane of 3-space. Pick a point in 3-space, say P_0, for simplicity, (0,0,1), as the inversion center. Choose $r = 1$ as the inversion radius. Perform the inversion for each point in S, we get a new set of n points S'. Let (x_p, y_p) be coordinates of the points in S and (x_s, y_s, z_s) be the coordinates of the points in S'. Due to the

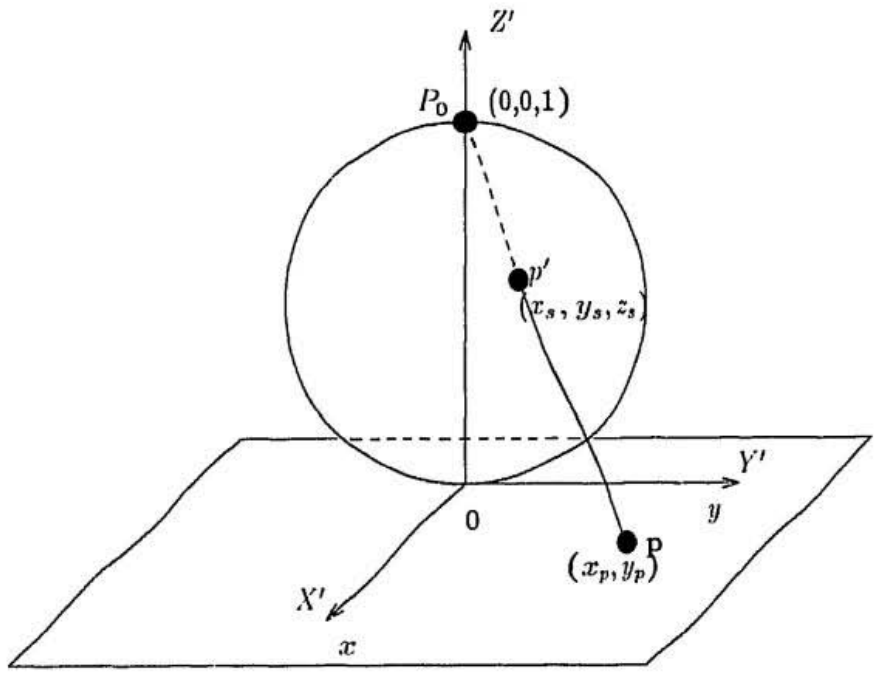


Figure 3.2: Inversion

property of the inversion, all points of the xy-plane are mapped to a sphere, say O, with P_0 at the apex. See Figure 3.2.

The equation of the line segment P_0p is :

$$\frac{x}{x_p} = \frac{y}{y_p} = \frac{z-1}{-1} \tag{3.1}$$

The distance between P_0 and p, $\mid \vec{P_0p} \mid$, is :

$$\sqrt{x_p^2 + y_p^2 + 1}$$

The distance between P_0 and p', $\mid \vec{P_0p'} \mid$, is :

$$\sqrt{x_s^2 + y_s^2 + (z_s - 1)^2}$$

The inversion radius r is 1. So we can get:

$$| \vec{P_0 p} | = r^2 / | \vec{P_0 p'} |$$

$$\sqrt{x_p^2 + y_p^2 + 1} \bullet \sqrt{x_s^2 + y_s^2 + (z_s - 1)^2} = 1 \tag{3.2}$$

$\vec{P_0 p}$ is in the same line as $\vec{P_0 p'}$. From Eq 3.1, we can get:

$$\frac{x_s}{x_p} = \frac{y_s}{y_p} = \frac{z_s - 1}{-1} \tag{3.3}$$

From Eq 3.2, Eq 3.3, we can get:

$$x_s = \frac{x_p}{(x_p^2 + y_p^2 + 1)}$$

$$y_s = \frac{y_p}{(x_p^2 + y_p^2 + 1)}$$

$$z_s = \frac{x_p^2 + y_p^2}{(x_p^2 + y_p^2 + 1)}$$

Therefore, inversion for each point can be done in constant time.

Lemma 3.2 *Constructing a* $3-d$ *convex hull can be done in* $\mathcal{O}(\log^3 n)$ *time on an* n*-processor hypercube connected computer network*

Proof: The algorithm will be described later in section 3.3.1.

Construct the convex hull of the points in S'. All n of the points of S' will be on the convex hull because inversion about P_0 maps all points of the xy plane to a sphere with P_0 at the apex. The convex hull has $\mathcal{O}(n)$ faces F_i. Each face F_i of the convex hull determines a plane in the 3-dimensional space. If we exclude degeneracies, each convex face is a triangle and has three adjacent faces. Each convex face F_i can be represented by three points through which it passes. Each convex face is stored in a PE. The PE's index and PEs in which its adjacent faces are present are known by the PE.

Lemma 3.3 *All the Voronoi vertices can be determined in constant time.*

Proof: In order to determine the Voronoi vertices, each PE which contains a face of convex hull performs the "reinversion". Invert faces of convex hull (with respect to the center of the inversion P_0 and radius r)and obtain $\mathcal{O}(n)$ spheres which intersect the xy plane in $\mathcal{O}(n)$ circles. The equation of line segment P_0p' is :

$$\frac{x}{x_s} = \frac{y}{y_s} = \frac{z-1}{z_s-1} \tag{3.4}$$

Because p is on the line Pp' and p is in the xy-plane.

$$\frac{x_p}{x_s} = \frac{y_p}{y_s} = \frac{z_p - 1}{z_s - 1} \tag{3.5}$$

$$z_p = 0. \tag{3.6}$$

From Eq 3.5 and Eq 3.6, we can get:

$$x_p = \frac{x_s}{(1 - z_s)}$$

$$y_p = \frac{y_s}{(1 - z_s)}$$

Find the center of the circle of the three points corresponding to each face F_i. Let c_i denote the center of this circle. Each face F_i of the convex hull associates with a half-space H_i which contains the convex hull and whose boundary plane is face F_i. If half-space H_i contains P_0, then c_i is a Voronoi vertex and set flag v_i to 1. Otherwise, set $v_i = 0$. (H_i does not contain P_0, the c_i is a farthest Voronoi vertex. We do not consider this situation in the thesis. For more information about farthest Voronoi diagrams, please see Brown's paper [21].

It should be explained why the centers of the circles may be the Voronoi vertices. In order to prove this, it is sufficient to show that these circles each pass through three of the n points of S and do not contain any of the other $n-3$ points in the interior. Because the inversion is involution, each of these

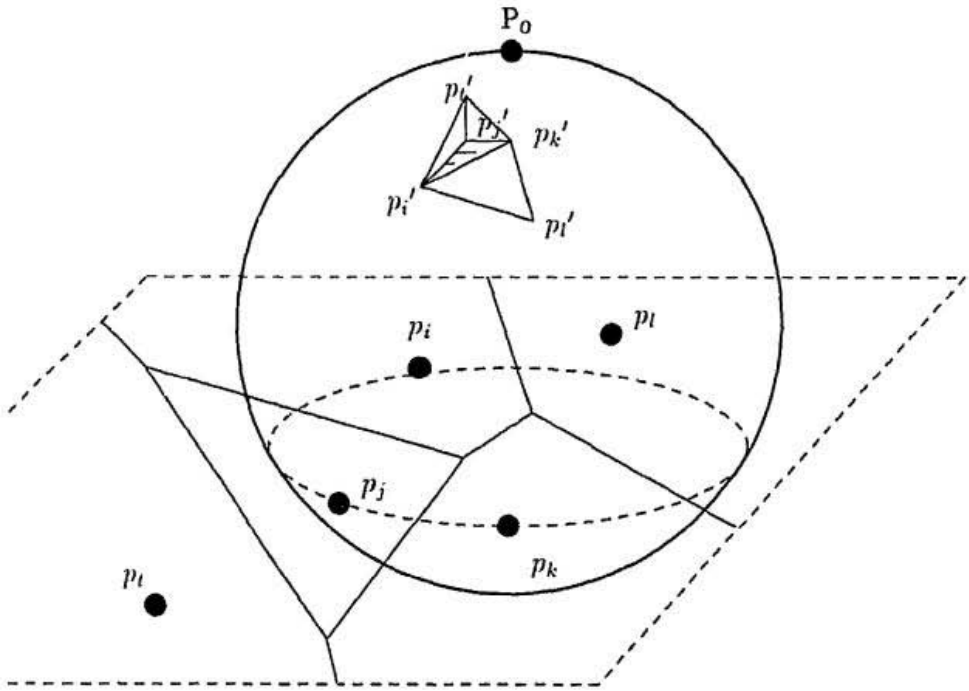


Figure 3.3: Relation between 3-d convex hull and 2-d Voronoi diagram

circles passes through three of the n points of S. If the circle passing through points p_i, p_j, and p_k of S contains another point $p_l \in S$ in its interior, then the convex hull of the transformed points S' will not contain a face $p_i'p_j'p_k'$ because of the presence of point p_l'. Therefore, each of the $\mathcal{O}(n)$ circles passes through three of n points of S and do not contain any of the other $n-3$ points in the interior. See Figure 3.3. Therefore, all the Voronoi vertices can be determined in constant time.

Lemma 3.4. *All the Voronoi edges can be constructed in* $\mathcal{O}(\log^2 n)$ *time*

Proof: Each PE containing c_i, perform RAR to get c_j. Face F_i and F_j are adjacent faces.

- If $v_i = 1$ and $v_j = 1$, then v_iv_j is a Voronoi edge.
- If $v_i = 1$ and $v_j = 0$, then v_iv_j is a Voronoi ray starting at v_i in the direction of v_iv_j.
- If $v_i = 0$ and $v_j = 1$, then v_jv_i is a Voronoi ray starting at v_j in the direction of v_jv_i.

Each face has at most three adjacent faces. Constructing Voronoi edges can be done in $\mathcal{O}(\log^2 n)$ time because of the RAR operation.

From Lemma 3.1, Lemma 3.2, Lemma 3.3 and Lemma 3.4, we can get:

Theorem 3.1 *Constructing a Voronoi diagram on an* n *– processor hypercube connected computer network can be done in* $\mathcal{O}(log^3 n)$ *time.*

The flow chart of the algorithm is shown in Figure 3.4.

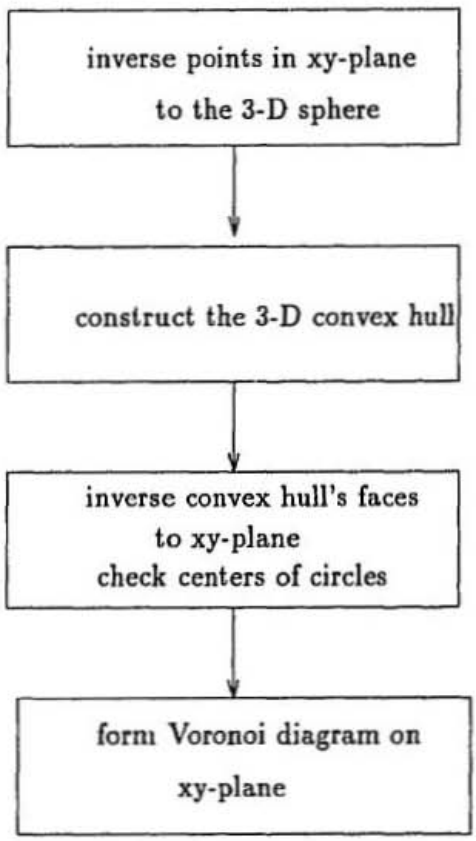


Figure 3.4: The algorithm of constructing a Voronoi Diagram

Procedure $CH(S)$;
begin $S_1 := \{p_1, \cdots, p_{\lfloor n/2 \rfloor}\}$;
$S_2 := \{p_{\lfloor n/2 \rfloor}, \cdots, p_n\}$;
$P_1 := CH(S_1)$; $P_2 := CH(S_2)$;
$P := MERGE(P_1, P_2)$;
return P
end

Table 3.1: The Algorithm of Constructing 3-Dimensional Space Convex Hull

3.3.1 Parallel algorithm for constructing the 3-d convex hull

It is clear that one of the main steps for constructing a Voronoi diagram is to construct a 3-dimensional space convex hull. In this subsection, based on Preparata-Hong[22] method, a parallel algorithm to construct a 3-dimensional space convex hull will be discussed.

Let $S = \{p_1, p_2, ..., p_n\}$ be a set of n points in 3-dimensional space. The convex hull of S is denoted as $CH(S)$. Recursively divide the set $S = \{p_1, p_2, ..., p_n\}$ into two subsets. $S_1 = \{p_1, p_2, ..., p_{\lfloor n/2 \rfloor}\}$ and $S_2 = \{p_{\lfloor n/2 \rfloor}..., p_n\}$. Let P_1 be $CH(S_1)$ and P_2 be $CH(S_2)$. Merge P_1 and P_2, we can get P. The recursive algorithm shown in Table 3.1. Clearly the merge function is the crucial component of the algorithm.

Let P_1 P_2 be two polyhedrons to be merged. If a face of P_1 or P_2 is

still a face of the merged polyhedron P then, the face is an external face, otherwise the face is an internal face. Let C_{P_1} be the circuit of P_1 and C_{P_2} be the circuit of P_2. C_{P_1} contains edges of P_1, which are shared by an internal face and an external face. C_{P_2} contains edges of P_2 , which are shared by an internal and an external face.

Lemma 3.5: *To determine each face of convex hull P_1 (or P_2) is an external face or an internal face can be done in $\mathcal{O}(\log^2 n)$ time.*

Proof: The algorithm will be discussed in section 3.3.2.

Lemma 3.6: *Construction of the circuits C_{P_1} and C_{P_2} for polyhedron P_1 and P_2 can be done in constant time.*

Proof: For each external face of both polyhedron P_1 or P_2, check its adjacent faces. If the adjacent face is an internal face, the edge induced by two faces is an edge of the circuit. Each face has at most three adjancent faces. Therefore, constructing a circuit can be done in constant time.

Lemma 3.7 : *Adding new faces of convex hull $CH(S_1, S_2)$ along two circuits can be done in $\mathcal{O}(\log^2 n)$ time.*

Proof: The algorithm will be discussed in section 3.3.5.

Lemma 3.8 : *Removal of the internal faces of P_1 and P_2 can be done in constant time.*

Proof: For each internal face, check its adjacent faces. Remove the edges bounding two internal faces. This step can be done in constant time.

From Lemma 3.5, Lemma 3.6, Lemma 3.7 and Lemma 3.8 , it can be concluded that:

Theorem 3.2 *The merge of P_1 and P_2 takes $\mathcal{O}(\log^2 n)$ time.*

Theorem 3.3 *Constructing a 3-dimensional space convex hull can be done in $\mathcal{O}(\log^3 n)$ time on an $n-$processor hypercube connected computer network.*

Proof: The algorithm used to construct a 3-dimensional space convex hull is shown in Table 3.1. If the "merging" of two convex hulls with at most n vertices in total, *i.e.*, the construction of the convex hull of their union, can be done in at most $M(n)$ operations, an upper bound to the number $T(n)$ of operations used by the recursive algorithm is given by the equation $T(n) = 2T(n/2) + M(n)$. It has been shown (Theorem 3.2) that $M(n)$ is $\mathcal{O}(\log^2 n)$. Therefore, $T(n)$ is $\mathcal{O}(\log^3 n)$ time complexity because of $\mathcal{O}(\log n)$ recursive calls.

The flow chart of the merge procedure is shown in Figure 3.5.

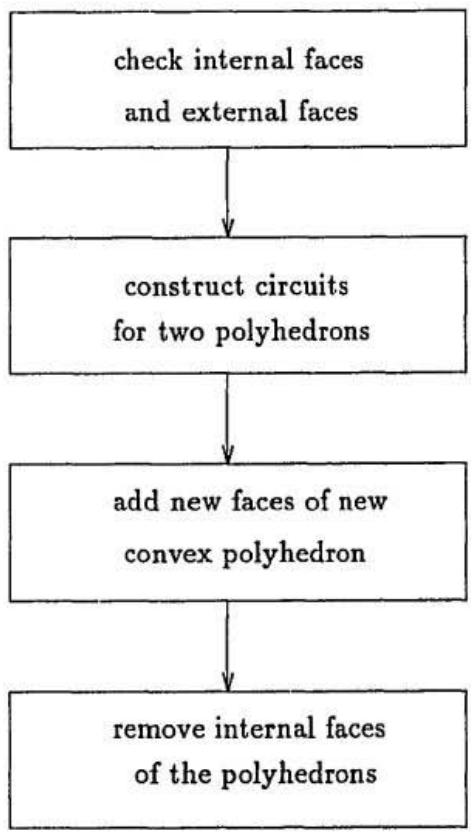


Figure 3.5: The algorithm of a three dimension convex hull merge

3.3.2 A parallel algorithm to test external faces and internal faces

The first step in the 3-dimensional space convex hull merge is to determine the external faces and internal faces. In this subsection, we will discuss the algorithm of testing external faces and internal faces.

When considering a convex polyhedron, each convex polyhedron face F_i is represented by an equation $\alpha_i x + \beta_i y + \gamma_i z + \delta = 0$ with normal vector $< a_i, b_i, c_i >$ pointing away from the polyhedron, where

$$a_i = \frac{\alpha_i}{\sqrt{\alpha_i^2 + \beta_i^2 + \gamma_i^2}},$$

$$b_i = \frac{\beta_i}{\sqrt{\alpha_i^2 + \beta_i^2 + \gamma_i^2}},$$

$$c_i = \frac{\gamma_i}{\sqrt{\alpha_i^2 + \beta_i^2 + \gamma_i^2}}.$$

The convex angle formed by faces F_i and F_j with normal vector $< a_i, b_i, c_i >$ and $< a_j, b_j, c_j >$ is $\cos^{-1} < a_i, b_i, c_i > \cdot < a_j, b_j, c_j >$ which is $\cos^{-1}(a_i a_j + b_i b_j + c_i c_j)$. In the range $0 \leq \theta \leq \pi$, the function $\cos\theta$ decreases from 1 to -1; the inverse function $\cos^{-1}$ also decreases as θ increases. The distance between two points (a_i, b_i, c_i) and (a_j, b_j, c_j) is $\sqrt{2(1 - (a_i a_j + b_i b_j + c_i c_j))}$,

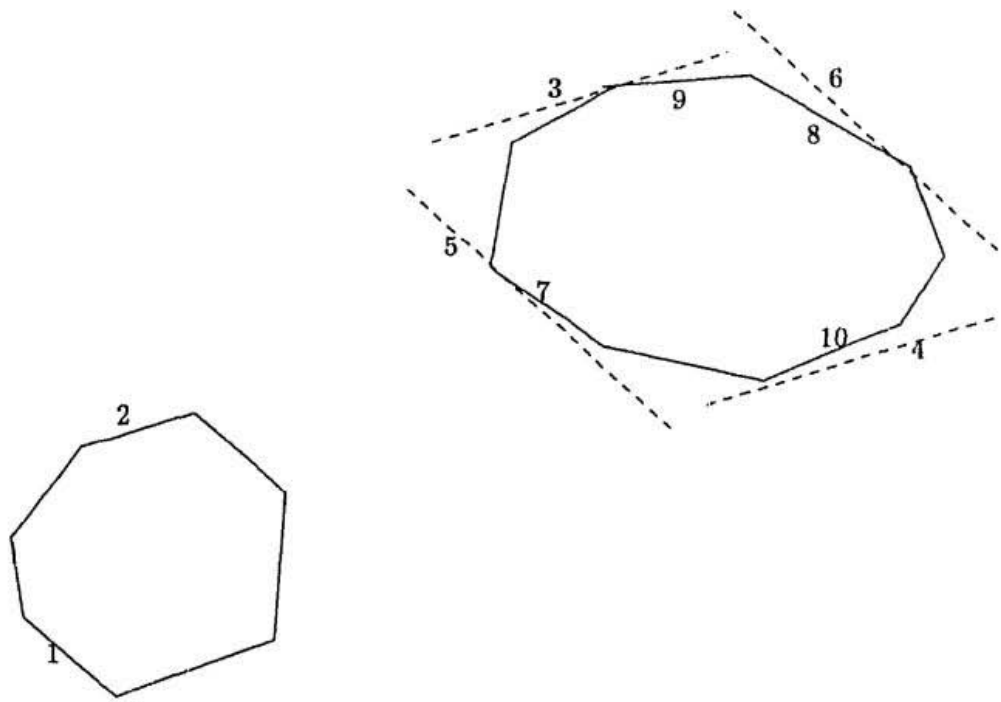
since ${a_i}^2 + {b_i}^2 + {c_i}^2 = {a_j}^2 + {b_j}^2 + {c_j}^2 = 1$. Therefore, $\cos^{-1}(a_i a_j + b_i b_j + c_i c_j)$ decreases as $\sqrt{2(1 - (a_i a_j + b_i b_j + c_i c_j)}$ decreases. Now we can conclude that:

Theorem 3.4 *The convex angle formed by face F_i with normal vector $< a_i, b_i, c_i >$ with face F_j with normal vector $< a_j, b_j, c_j >$ decreases as the distance between points (a_i, b_i, c_i) and (a_j, b_j, c_j) decreases.*

Consider the half-space bounded by the face $F_{P_1}(i)$; we denote the half-space that contains polyhedron P_1 by $H(P_1, i)$. Face $F_{P_1}(i)$ belongs to convex polyhedron P which is merged by P_1 and P_2, if P_2 lies in the half-space $H(P_1, i)$. For each face $F_{P_1}(i)$, there exist two planes which are parallel to $F_{P_1}(i)$ and support polyhedron P_2, denoted as $PL'_{P_1}(i)$ and $PL''_{P_1}(i)$. For $PL'_{P_1}(i)$ and $PL''_{P_1}(i)$, there exist two faces of polyhedron P_2 which intersect at point of tangency with $PL'_{P_1}(i)$ and $PL''_{P_1}(i)$ and form smallest angles with $PL'_{P_1}(i)$ and $PL''_{P_1}(i)$, denoted as $F'_{P_2}(i)$ and $F''_{P_2}(i)$. See Figure 3.6.
Due to convexity, $F_{P_1}(i)$ is an external face if $F'_{P_2}(i)$ and $F''_{P_2}(i)$ are in the half-space $H(P_1, i)$, otherwise, $F_{P_1}(i)$ is an internal face. Thus the key point of testing $F_{P_1}(i)$ to be an external face is to find $F'_{P_2}(i)$ and $F''_{P_2}(i)$. We have the following theorem:

Theorem 3.5 *$F_{P_1}(i)$ is an external face if $F'_{P_2}(i)$ and $F''_{P_2}(i)$ are in the half-space $H(P_1, i)$, otherwise, $F_{P_1}(i)$ is an internal face.*

The algorithm to test the external and internal faces for polyhedron P_1



1— $F_{P_1}(i)$

2— $F_{P_1}(j)$

3— $PL'_{P_2}(j)$

4— $PL''_{P_2}(j)$

5— $PL'_{P_2}(i)$

6— $PL''_{P_2}(i)$

7— $F'_{P_2}(i)$

8— $F''_{P_2}(i)$

9— $F'_{P_2}(j)$

10— $F''_{P_2}(j)$

Figure 3.6: The two dimensional analogy

is described as follows. The algorithm for polyhedron P_2 is similar.

Lemma 3.9: *Each PE containing $F_{P_2}(j)$ does a transformation. This can be done in constant time.*

Proof: Each face $F_{P_2}(j)$ will be transformed into a point $p_{P_2}(j)$ on the surface of the unit sphere, where the coordinates of $p_{P_2}(j)$ is (a_j, b_j, c_j) , and $< a_j, b_j, c_j >$ is the normal vector, pointing away from P_2 of face $F_{P_2}(j)$.

A spherical Voronoi diagram of an n-points set on a sphere is a partition of the surface of the sphere into n regions: the region j for point $p_{P_2}(j)$ is the locus of points on the surface of the sphere which are closer to $p_{P_2}(j)$ than to any other $n-1$ points.

Lemma 3.10: *Using $\{p_{P_2}(1), p_{P_2}(2), ...\}$ as site points, construct a spherical Voronoi diagram on the unit sphere. This can be done in $\mathcal{O}(\log^2 n)$ time*

Proof: The algorithm will be presented in section 3.3.3.

Lemma 3.11 : *Transform each face $F_{P_1}(i)$ with normal vector $< a_i, b_i, c_i >$ into two points, $p'_{P_1}(i)$ and $p''_{P_1}(i)$. This can be done in constant time for each PE.*

Proof : For each face of P_1, $F_{P_1}(i)$, there are two normal vectors $< a_i, b_i, c_i >$ and $< -a_i, -b_i, -c_i >$. These two vectors can be transformed

to two points on the unit sphere.

Lemma 3.12 : *For each i, determine the nearest neighbors $p_{P_2}(i')$ and $p_{P_2}(i'')$ of the points $p_{P_1}(i')$ and $p_{P_1}(i'')$. This can be done in $\mathcal{O}(\log^2 n)$ time.*

Proof: Locate $p_{P_1}(i')$ and $p_{P_1}(i'')$ on a sphere Voronoi diagram to find site points $p_{P_2}(i')$ and $p_{P_2}(i'')$. Due to the property of a spherical Voronoi diagram, $p_{P_2}(i')$ and $p_{P_2}(i'')$ are nearest neighbors of $p_{P_1}(i')$ and $p_{P_1}(i'')$. Locating points on a spherical Voronoi diagram needs $\mathcal{O}(\log^2 n)$ time. The algorithm will be discussed in section 3.3.4.

Lemma 3.13 : *For each i, check that both $F'_{P_2}(i)$ and $F''_{P_2}(i)$, are in $H(P_1, i)$. if* **true** *then $F_{P_1}(i)$ is an external face, otherwise $F_{P_1}(i)$ is an internal face. This can be done in constant time.*

Proof: $p_{P_2}(i')$ represents one face of P_2, say $F'_{P_2}(i)$. $p_{P_2}(i'')$ represents one face of P_2, say $F''_{P_2}(i)$. $p_{P_1}(i')$ and $p_{P_1}(i'')$ represent $PL'_{P_1}(i)$ and $PL''_{P_1}(i)$ which are parallel to face $F_{P_1}(i)$ and support polyhedron P_2. $p_{P_2}(i')$ and $p_{P_2}(i'')$ are nearest neighbors of the points $p_{P_1}(i')$ and $p_{P_1}(i'')$. Due to Theorem 3.4, $F'_{P_2}(i)$ forms the smallest convex angle with $PL'_{P_1}(i)$ and $F''_{P_2}(i)$ forms the smallest convex angle with $PL''_{P_1}(i)$. Due to Theorem 3.5, $F_{P_1}(i)$ is an external face if $F'_{P_2}(i)$ and $F''_{P_2}(i)$ are in the half-space $H(P_1, i)$, otherwise, $F_{P_1}(i)$ is an internal face. Check that $F'_{P_2}(i)$ and $F''_{P_2}(i)$ are in the half-space

$H(P_1, i)$ can be done in constant time.

Theorem 3.6 : *The parallel algorithm for testing the external face and internal face can be done in* $\mathcal{O}(\log^2 n)$ *time.*

Proof: The correctness and the time complexity follow from Lemma 3.9, Lemma 3.10, Lemma 3.11, Lemma 3.12 and Lemma 3.13.

The flow chart of the algorithms is shown in Figure 3.7.

3.3.3 A parallel algorithm to construct a spherical Voronoi diagram

In order to distinguish external faces and internal faces, a spherical Voronoi diagram on the unit sphere should be constructed. Based on the method proposed by Brown[21], a parallel algorithm to construct a spherical Voronoi diagram on a unit sphere is described here.

Given a set of points $(p_1, p_2, ..., p_n)$, every point p_i is on the unit sphere. For each point p_i on the unit sphere, there is a plane PL_i tangent to the sphere at point p_i. Let H_i be the half-space bounded by PL_i which contains the entire sphere. The intersection of n half-spaces H_i form a convex body say, C. The spherical Voronoi diagram is now obtained by a simple projection of the edges of this polyhedron to the surface of the sphere. The projection is a "radial" projection: the projection of a point p is the point

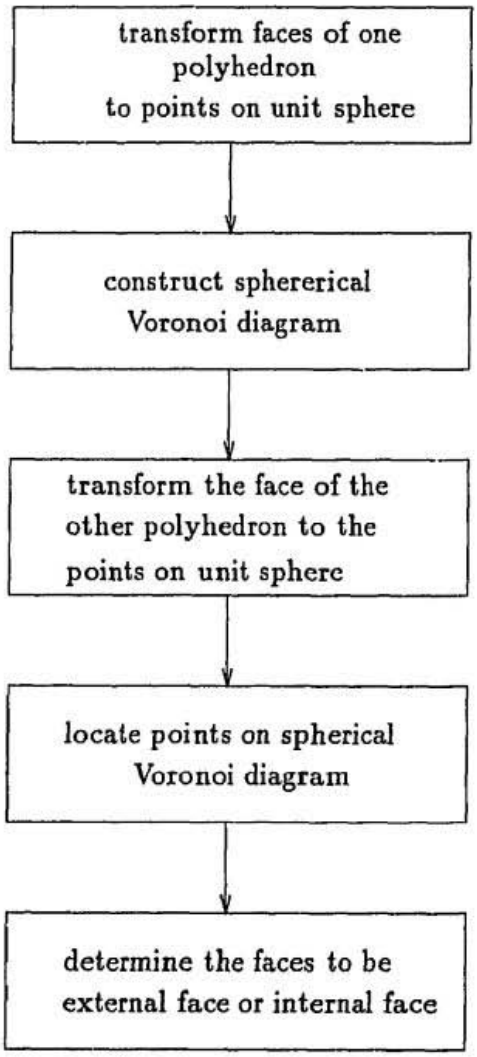


Figure 3.7: The algorithm of external and internal face test

where a line segment connecting the center of the sphere and point p intersects the sphere. This projection maps edges of the polyhedron to arcs of great circles on the sphere. The vertices of the polyhedron are mapped to spherical Voronoi points and the face of the polyhedron are mapped to spherical Voronoi regions.

Lemma 3.14 :*All the planes PL_i can be found in constant time.*

Proof: For each PE containing $F_{P_2}(i)$ with normal vector $< a_i, b_i, c_i >$, find a corresponding point p_i, (a_i, b_i, c_i), on unit sphere. Then, the plane which is tangent to the unit sphere at p_i, say PL_i, can be obtained. The equation of the plane PL_i is $a_i x + b_i y + c_i z = 1$. This needs constant time for each PE.

Lemma 3.15 : *All the edges and vertices of the convex body C can be found in $\mathcal{O}(\log^2 n)$ time.*

Proof: In order to find all the edges and vertices of C, each PE containing $F_{P_2}(i)$ does a RAR operation from the PE which contains $F_{P_2}(j)$. $F_{P_2}(i)$ and $F_{P_2}(j)$ are adjacent faces. The plane PL_i is obtained from $F_{P_2}(i)$ and the plane PL_j is obtained from $F_{P_2}(j)$. An edge of C is the intersection of PL_i and PL_j.

$$\begin{cases} a_i x + b_i y + c_i z = 1 \\ a_j x + b_j y + c_j z = 1 \end{cases} \tag{3.7}$$

By examining all the faces adjacent to $F_{P_2}(i)$ in the convex hull P_2, we can obtain the vertices of C. Because each convex polyhedron face is a triangle, each face has at most three adjacent faces. Therefore, all the edges and vertices of the convex body C can be found in $\mathcal{O}(\log^2 n)$ time.

Lemma 3.16 : *All the vertices and edges of a spherical Voronoi diagram can be found in constant time.*

Proof : Connect the vertices of C with the center of a sphere intersecting the unit sphere. These are the vertices of a spherical Voronoi diagram. By connecting these Voronoi vertices the arcs of great circles on the sphere are determined. These are the edges of the spherical Voronoi diagram. This step can be done in constant time for each PE.

Based on Lemma 3.14, Lemma 3.15 and Lemma 3.16, it is concluded that:

Theorem 3.7 *Construction of a spherical Voronoi diagram can be done in $\mathcal{O}(\log^2 n)$ time.*

The flow chart of the algorithm is shown in Figure 3.8.

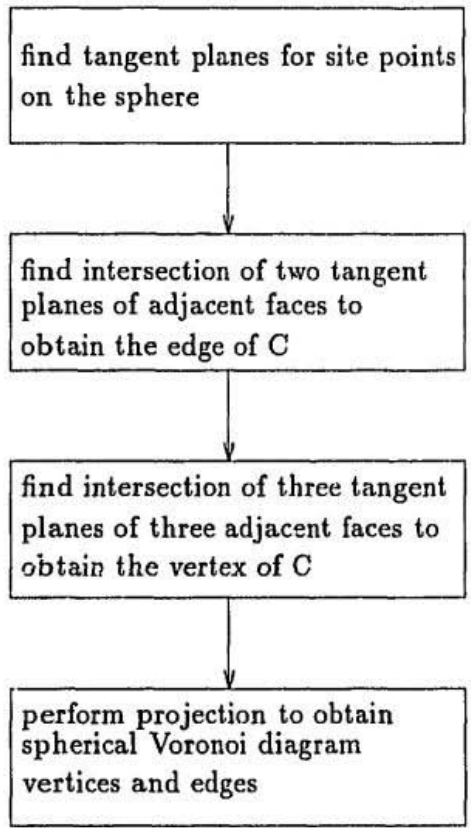


Figure 3.8: The algorithm for constructing spherical Voronoi Diagram

3.3.4 A parallel algorithm to locate points on a spherical Voronoi diagram

This algorithm is based on the chain method described by Lee and Preparata [23]. From a computational viewpoint, any solution to the point location problem should include two steps: the preprocessing step and the search step. The preprocessing step constructs the data structure postulated by the search algorithm. The search step locates the query points in the subdivision.

Since the points are located on the spherical Voronoi diagram, the data structure postulated by the search algorithm is the spherical Voronoi diagram. For an efficient search, the first thing to be done is to get a representation of the spherical Voronoi diagram. It is clear that the Voronoi diagram is composed by a set of monotone chains which are generated at different levels of the merging step, when Voronoi diagram is being constructed by a divide-and-conquer method. Each chain has its own level and index (the rank of the chain in the chains of given level). Chains may share common edges. If an edge e belongs to more than one chain, it then belongs to all members of a set of consecutive chains. We assign e to hierarchically the highest chain to which e belongs.

Theorem 3.8 : *Marking the levels and index for spherical Voronoi edges can be done in* $\mathcal{O}(\log^2 n)$ *time.*

Proof: Sort the site points of a spherical Voronoi diagram by $\theta_i = x_i/\sqrt{x_i^2 + y_i^2}$. For each site point, a binary index is given. Each PE which contains an edge e_i of the spherical Voronoi diagram does an RAR to get the index of the pair of points it is associated with. "bit exclusive or" of two points, say Ψ, is then obtained. The level of the edge e_i can be obtained by $l_i = \lfloor log\Psi \rfloor$. The index of the edge e_i is $\lfloor (2\text{'s } complement\ (2^{l_i}) - 2^{l_i}) \wedge (index\ of\ e_i\text{'s } associated\ point) \rfloor / 2^{l_i+1}$. For example, if e_i is associated with points 0010 and 0101, "bit exclusive or" of 0010 and 0101 is 0111, $\lfloor \log 0111 \rfloor = (2)_{10}$, so e_i is of level 2. $2's complement(2^2) - 2^2 = 2$'s $complement(0100)$ -0100 = 1100 - 0100 = 1000. (1000 $\wedge$ 0010) (or 1000 $\wedge$ 0101) = 0; so e_i is indexed as 0 in the chains of level 2. Therefore, marking the levels for spherical Voronoi arcs can be done in $\mathcal{O}(\log^2 n)$ time because of the sorting operation and the RAR operation. See Figure 3.9.

Now the search algorithm is discussed. All edges are sorted by their level as the primary key, their index as the secondary and the y-coordinate of the endpoint of the edge as the ternary key (endpoint with less y-coordinate between two endpoints of a chosen edge). All query points are sorted by their y-coordinates. For each query point, two flags are assigned, denoted as $L(k)$ and $R(k)$, which are represented by the edges on the left and right side of the query point. Initially, all query points are assigned the highest level $\lceil logn \rceil$ and index 0, $L(k) \leftarrow -\infty$, $R(k) \leftarrow \infty$. Then, for each level i, from $i = \lceil logn \rceil$ to $i = 0$, the following operations are performed :

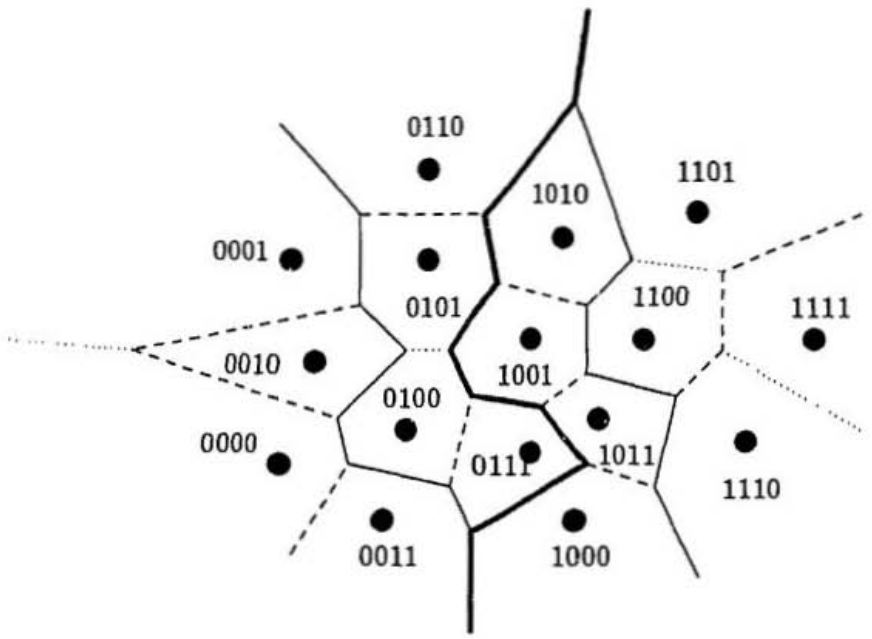


Figure 3.9: Representation of chain

1. All query points have assigned the same level, equal to i. Merge the query points and the set of edges. This step can be done in $\mathcal{O}(\log n)$ time because of the merging operation.

2. Generalize operation is performed to find, for each query point p_k, the corresponding edge e the p_k should be discriminated against. That is, the y-coordinate of p_k is between y-coordinates of endpoints of e. This step can be done in $\mathcal{O}(\log n)$ time because of the generalize operation.

3. Let e be the corresponding edge for query point p_k. Depending on which side of e the query point p_k lies, compare e with either $L(k)$ or $R(k)$.

   - If p_k is on the left side of e and e is to the right of $L(k)$, update $L(k)$ with e.
   - If p_k is on the right side of e and e is to the left of $R(k)$, update $R(k)$ with e.

   This step can be done in constant time for each PE.

4. If $L(k)$ and $R(k)$ are bounding the same region, *i.e.* they have the same associate point, the query point p_k is located. This step can be done in constant time for each PE.

5. For unlocated points, p_k calculates the index of the chain at next level it should be discriminated. The index is $[(2\text{'s } complement\ (2^{i-1}) - 2^{i-1}) \wedge$

(index of e's associated point)]/2^i. This step can be done in constant time for each PE.

6. Unmerge edges and query points (using former indices of query points). This step can be done in $\mathcal{O}(\log n)$ time because of unmerge operation.

7. Perform concentrate operation for the unlocate query points with answer "left" of corresponding edge in step 2. Unlocate query point with answer "right" will be also concentrated. This step can be done in $\mathcal{O}(\log n)$ because of the concentrate operation.

8. Since both subsets of unlocate query points are sorted by the new indices after concentrating. Merging "left" and "right" unlocate query points by their new indices. Give next level to all unlocated query points. This step can be done in $\mathcal{O}(logn)$ time because of the merging operation.

Theorem 3.9 : *Locating the points on a spherical Voronoi diagram can be done in* $\mathcal{O}(\log^2 n)$ *time.*

Proof: From the discussion above, it is clear that the preprocess algorithm can be done in $\mathcal{O}(\log^2 n)$ time. (Due to Theorem 3.8). The search algorithm needs $\mathcal{O}(\log^2 n)$ time because each step in the search algorithm takes $\mathcal{O}(\log n)$ time and there are $\log n$ iterations. Therefore, locating points on a spherical Voronoi diagram can be done in $\mathcal{O}(\log^2 n)$ time.

The flow chart of the search algorithm is shown in Figure 3.10.

3.3.5 A parallel algorithm to add new faces to the convex polyhedron

P_1 and P_2 are two convex polyhedrons to be merged. P is a convex polyhedron which is obtained by merging P_1 and P_2. New faces are the faces which do not belong to P_1 and P_2 but belong to P. If we exclude degeneracies, each convex polyhedron face is a triangle. C_{P1} is the circuit of P_1 and C_{P2} is the circuit of P_2. The new face is determined by an edge of C_{P1} and a node of C_{P2} or by an edge of C_{P2} and a node of C_{P1}.

In order to add new faces, the first thing to be done is to order the edges in C_{P_1} and C_{P_2}.

Lemma 3.18: *Ordering the edges in C_{P_1} and C_{P_2} can be done in $\mathcal{O}(\log^2 n)$ time.*

Proof: The observer P_2 is defined as an observer placed at any point of P_2 and oriented like the negative z-axis, and observer P_1 as an observer placed at any point of P_1 and oriented like the positive z-axis. The edges in C_{P1} are numbered in ascending order so that they form a clockwise sequence for an observe P_2. And the edges in C_{P2} are numbered in ascending order so that they form a counterclockwise sequence for an observe P_1. Both sequences are started at the vertices with the largest y-coordinates in

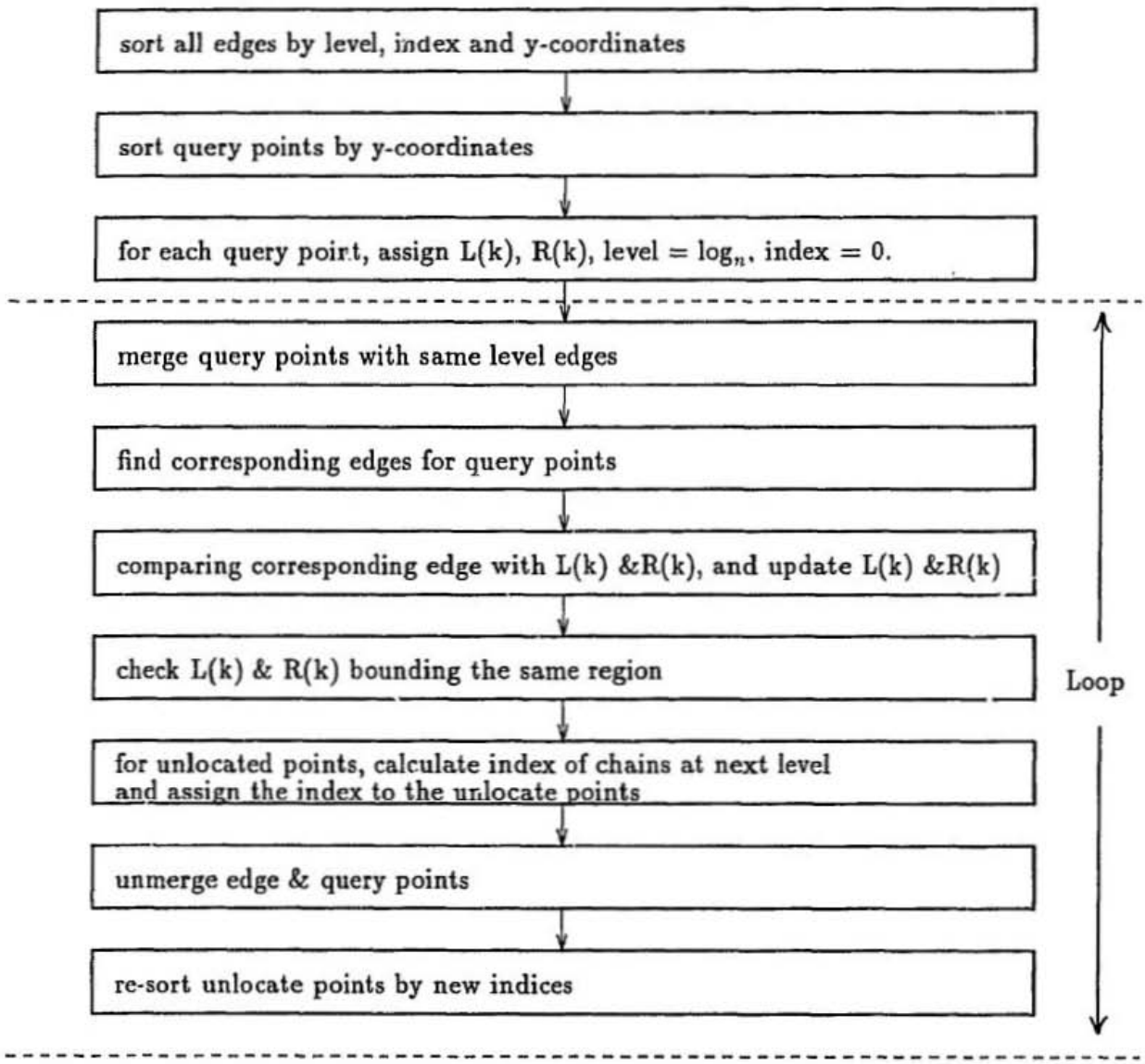


Figure 3.10: The search algorithm for points location

C_{P_1} and C_{P_2}. Let $C_{P_1}(i)[v_1]$ and $C_{P_2}(j)[v_1]$ be the vertices at which edges $C_{P_1}(i)$ and $C_{P_2}(j)$ originate respectively. Then $(C_{P_1}(0)[v_1], C_{P_1}(1)[v_1], ...)$ and $(C_{P_2}(0)[v_1], C_{P_2}(1)[v_1], ...)$ are the sequences of vertices of C_{P_1} and C_{P_2} respectively. Due to convexity, the convex angle formed by $(C_{P_1}(0)[v_1], C_{P_1}(i)[v_1])$ and $(C_{P_1}(0)[v_1], C_{P_1}(j)[v_1])$ is clockwise for an observe P_2, where $i < j$; the convex angle formed by $(C_{P_2}(0)[v_1], C_{P_2}(i)[v_1])$ and $(C_{P_2}(0)[v_1], C_{P_2}(j)[v_1])$ is counterclockwise for an observer P_1, where $i < j$. Therefore, edges in C_{P_1} can be ordered and so can those in C_{P_2}. This can be done by using sorting algorithm which takes $\mathcal{O}(\log^2 n)$ time.

Lemma 3.19 : *For each edge in $C_{P_1}(i)$, find a node of $C_{P_2}(j)[v_1]$ such that the plane determined by $C_{P_1}(i)$ and the node $C_{P_2}(j)[v_1]$ is new face. The same procedure is carried out for the edges in C_{P_2}. This can be done in $\mathcal{O}(\log^2 n)$ time.*

Proof: The proof is for each edge in $C_{P_1}(i)$ finding a node of $C_{P_2}(j)[v_1]$. The proof for each edge in $C_{P_2}(i)$, finding a node of $C_{P_1}(j)[v_1]$ is same.

If the plane determined by the edge $C_{P_1}(i)$ and the node of $C_{P_2}(j)[v_1]$ is the new face of P then the convex angle formed by the plane determined by $C_{P_1}(i)$ and $C_{P_2}(j)[v_1]$ and the face bounded by $C_{P_1}(i)$ which belong to P is maximum. Let $\theta_{P_1}(i,j)$ be an angle measure associated with edge $C_{P_1}(i)$ and vertex $C_{P_2}(j)[v_1]$, as the convex angle formed by the plane determined by $C_{P_1}(i)$ and $C_{P_2}(j)[v_1]$ and the face bounded by $C_{P_1}(i)$, which belong to

P. In an analogous manner, $\theta_{P_2}(j,i)$ is defined as the convex angle formed by the plane determined by $C_{P_2}(j)$ and $C_{P_1}(i)[v_1]$ and the face bounded by $C_{P_2}(j)$, which belongs to P. Let $j^{(i)}$ be the smallest index such that $\theta_{P_1}(i,j^{(i)})$ is a maximum among all $\theta_{P_1}(i,j)$, $0 \leq j \mid C_{P_2} \mid$, let $i^{(j)}$ be the largest index such that $\theta_{P_2}(j,i^{(i)})$ is a maximum among all $\theta_{P_2}(j,i)$, $0 \leq i <\mid C_{P_1} \mid$. For a particular i, $j^{(i)}$ can be determined by performing a maximum operation which takes $\mathcal{O}(\log \mid C_{P_2} \mid)$ time. It is observed that $(j^{(0)}, j^{(1)}, ...)$ and $(i^{(0)}, i^{(1)}, ...)$ are nondecreasing sequences. Firstly, $j^{(|C_{P_2}|/2)}$ can be found; then in parallel $j^{(|C_{P_2}|/4)}$ in the intervals $[0, j^{(|C_{P_2}|/2)}]$ and $[j^{(|C_{P_2}|/2)}, \mid C_{P_2} \mid -1]$ can be found respectively, and so on. It is straightforward to see that it takes $\log \mid C_{P_2} \mid$ iterations to obtain all $j^{(i)}$s. Therefore, for each edge in $C_{P_1}(i)$, finding a node of $C_{P_2}(j)[v_1]$ such that the plane determined by $C_{P_1}(i)$ and the node $C_{P_2}(j)[v_1]$ is new face can be done in $\mathcal{O}(\log^2 n)$ time.

Connect $C_{P_1}(i)[v_1]$ with $C_{P_2}(j^{(i)})[v_1]$ and connect $C_{P_1}(i)[v_2]$ with $C_{P_2}(j^{(i)})[v_1]$ to get a new face determined by $C_{P_1}(i)$ and $C_{P_2}(j^{(i)})[v_1]$. $C_{P_2}(j)[v_1]$ is connected with $C_{P_1}(i^{(j)})[v_1]$ and $C_{P_2}(j)[v_2]$ is connected with $C_{P_1}(i^{(j)})[v_1]$ to get a new face determined by $C_{P_2}(j)$ and $C_{P_1}(i^{(j)})[v_1]$.

From the above discussion, we get the following Theorem:

Theorem 3.10. : *The algorithm of adding new face takes* $\mathcal{O}(\log^2 n)$ *time.*

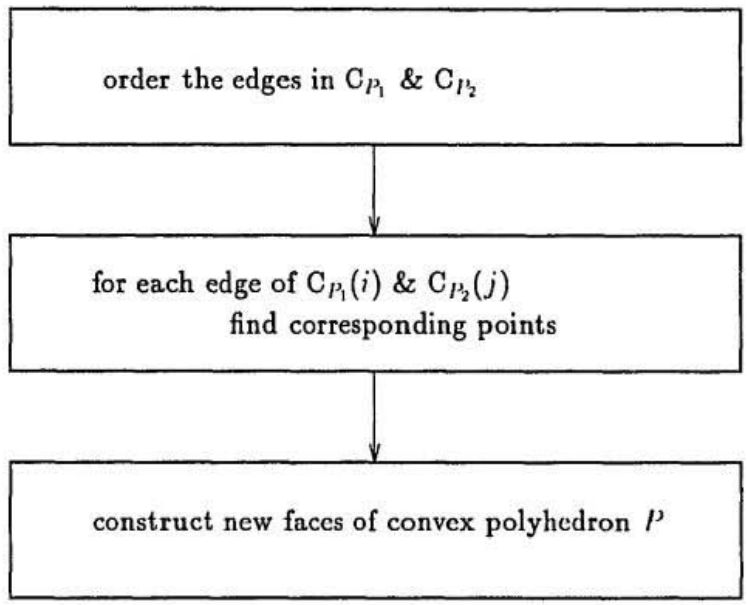


Figure 3.11: The algorithm for adding new faces of convex polyhedron

Proof : The correctness follows from Lemma 18 and Lemma 19.

The flow chart of the algorithm is shown in Figure 3.11.

3.4 Summary

In this Chapter, a parallel algorithm for constructing a Voronoi diagram on hypercube connected computer networks is developed. The algorithm is based on Brown's method [4] and consists of four steps. The details of the

algorithm are shown in section 3.3. One of the main steps of this algorithm is to construct a 3-dimensional convex hull. Based on Preparata-Hong's method [22], a parallel algorithm to construct a 3-dimensional convex hull is discussed in section 3.3.1. The strategy used in the algorithm is divide-and-conquer. Recursively divide the point set into two subsets and then merge two sub-convex-polyhedron. The crucial component of the algorithm is the merge function. The merge function is divided into four steps. To determine the convex polyhedron face is an external face or an internal face; to construct the circuit for each polyhedron which is merged; to add new faces; and to remove internal faces. The algorithm to determine the external face and the internal face is described in section 3.3.2. The algorithm to add new faces of convex polyhedron is discussed in section 3.3.5. In order to determine the external face and the internal face , the algorithm to construct a spherical Voronoi diagram and the algorithm of points location are used. The algorithm to construct a spherical Voronoi diagram is based on Brown's method [21] and is discussed in section 3.3.3. The points location algorithm is based on Preparata and Lee's method [23] and is described in section 3.3.4.

Let the algorithm to construct a Voronoi diagram be Algorithm_VD. Let the algorithm to construct a 3-dimensional space convex hull be Algorithm_CH. Let the algorithm to determine the external face and the internal face be Algorithm_EI. Let the algorithm to construct a spherical Voronoi diagram be Algorithm_SVD. Let the algorithm of points location be Al-

gorithm_LP. Let the algorithm to add new faces of polyhedron be Algorithm_AN. The time complexity of these algorithms are listed in the following table.

Name	Number of Processors	Time Complexity	Location
Algorithm_VD	$\mathcal{O}(n)$	$\mathcal{O}(\log^3 n)$	Section 3.3
Algorithm_CH	$\mathcal{O}(n)$	$\mathcal{O}(\log^3 n)$	Section 3.3.1
Algorithm_EI	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n)$	Section 3.3.2
Algorithm_SVD	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n)$	Section 3.3.3
Algorithm_LP	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n)$	Section 3.3.4
Algorithm_AN	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n)$	Section 3.3.5

From the discussion above, it is clear that our algorithm runs $\mathcal{O}(n)$ processors hypercube connected computer network and needs $\mathcal{O}(\log^3 n)$ time.

Chapter 4

Conclusion and Discussion

A Voronoi diagram is a useful data structure in computation geometry. In this thesis, SIMD hypercube connected computer network are chosen as the parallel computation model. In chapter 2, the fundamental operations on the hypercube connected computer network are discussed. In chapter 3, based on Brown's method, a parallel algorithm to construct a Voronoi diagram is developed. Our algorithm runs in $\mathcal{O}(\log^3 n)$ time on an $\mathcal{O}(n)$-processor hypercube connected computer network. Our algorithm is based on Brown's method which transforms the problem of construction of a planar a Voronoi diagram for an n-point set to construction of a convex hull of n points in three dimensional space. Comparing with the parallel algorithms which are based on the divide-and-conquer approach used by Shamos, our algorithm can be used to solve two computational geometry problems: constructing 2−dimensional Voronoi diagram and 3− dimensional convex hull. Comparing with Chow's methods which runs on a $\mathcal{O}(n)$ processors

CCC (Cube-Connected Cycles) model has $\mathcal{O}(\log^4 n)$ time complexity, our algorithm has less time complexity. Comparing with Chang-Sung Jeong's algorithm which runs in $\mathcal{O}(\sqrt{n})$ on an $\sqrt{n} \times \sqrt{n}$ mesh, our parallel computation model is more general. Most other popular networks can be easily mapped onto a hypercube. Next we will discuss some extension and future work.

4.1 Parallel Algorithm to Construct a Voronoi Diagram in $L_1(L_\infty)$ on a hypercube connected computer network

It is known that there are many methods to construct a Voronoi diagram on a single computer. Two of them were proposed by Shamos in 1975 and by Brown in 1979. Many parallel algorithms were suggested based on these two methods. Among those parallel algorithms, some are implemented on processor networks, some on shared-memory machines. The Voronoi diagrams for other metrics have also been studied by several researchers.

Given two points q_i and q_j in the plane R^2 with coordinates (x_i, y_i) and (x_j, y_j), respectively, the distance between q_i and q_j in the L_p metric is defined as $d_p(q_i, q_j) = (\mid x_i - x_j \mid^p + \mid y_i - y_j \mid^p)^{1/p}$ for $p = 1, 2, \ldots$ and $d_\infty(q_i, q_j) = max(\mid x_i - x_j \mid, \mid y_i - y_j \mid)$. The plane in which the L_p metric is

the distance measure is denoted by R_p^2. The bisector $B_p(q_i, q_j)$ of two points q_i and q_j is the locus of points equidistant from q_i and q_j, *i.e.* $B_p(q_i, q_j) = \{r \mid r \in R_p^2, d_p(r, q_i) = d_p(r, q_j)\}$. The locus of points closer to q_i than to q_j, denoted by $h_p(q_i, q_j)$, is one of the halfplanes containing q_i that is determined by the bisector $B_p(q_i, q_j)$, *i.e.* $h_p(q_i, q_j) = \{r \mid d_p(r, q_i) \leq d_p(r, q_j)\}$. Given a set S of points $q_1, q_2, ..., q_n$, the locus of points closer to q_i than to any other points, denoted by $V_S^p(q_i)$, is called the Voronoi region or polygon associated with q_i in the L_p metric and is thus given by $V_S^p(q_i) = \bigcap_{i \neq j} h_p(q_i, q_j)$, the intersection of all the halfplanes containing q_i. The entire set of Voronoi polygons partitions the plane into n regions and is referred to as the Voronoi diagram $V_p(S)$ for the set S in R_p^2. Figure 4.1 is an example of the Voronoi Diagram in L_1 metric.

In 1991, Chang-Sung Jeong [24] gave an $\mathcal{O}(\sqrt{n})$ parallel algorithm on $\sqrt{n} \times \sqrt{n}$ mesh-connected computer to construct a Voronoi diagram in $L_1(L_\infty)$ metric for a set of n points in the Cartesian plane.

An $\mathcal{O}(\log^3 n)$ algorithm to construct a Voronoi diagram of a set of n planar points in $L_1(L_\infty)$ metric on hypercube connected computer network can be obtained by using Jeong's [24] method and the fundamental operations discussed in Chapter 2.

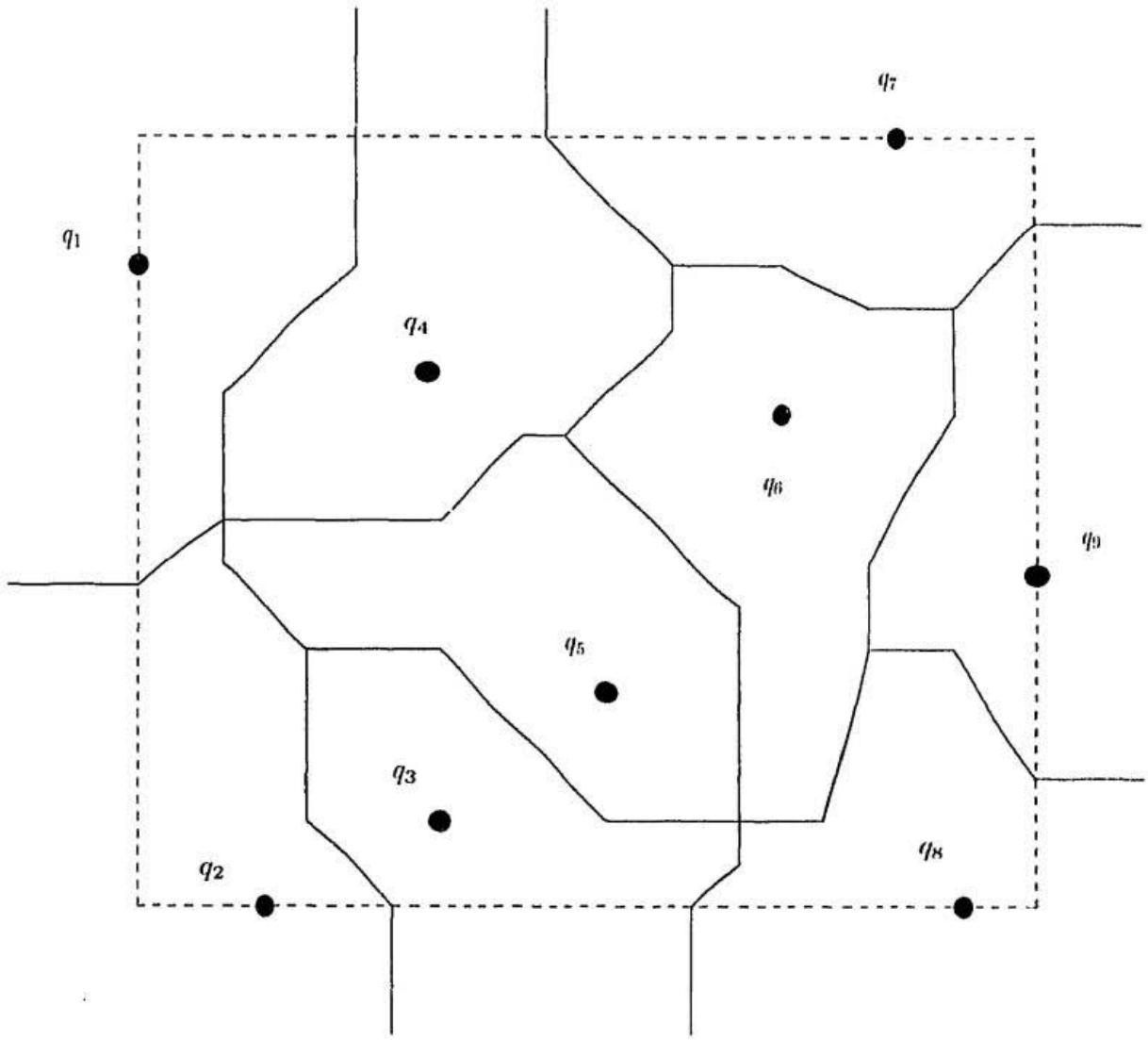


Figure 4.1: Voronoi diagram in L_1 metric

4.2 Optimal Parallel Algorithms to Construct Voronoi Diagrams

If $p(n)$ is the processor complexity, $t(n)$ is the parallel time complexity, and $seq(n)$ is the time complexity of the best known sequential algorithm for the problem under consideration, then $t(n) * p(n) = O(seq(n))$. If the product $t(n) * p(n)$ achieves the sequential lower bound for the problem, then we say the algorithm is optimal. Computing a Voronoi diagram of a set of points in the plane with a single processor has an $\Omega(n \log n)$ lower bound. In 1990, Guha gave a parallel algorithm for the rectilinear Voronoi diagram [25]. The algorithm runs in $\mathcal{O}(\log^2 n)$ time and uses $\mathcal{O}(n/\log n)$ processors. The computation model is CREW PRAM. In the same year, Wee and Chaiken presented a parallel L_1 metric Voronoi diagram algorithm [26]. The computation model is CREW PRAM and the algorithm runs in $\mathcal{O}(\log n)$ time and uses $\mathcal{O}(n)$ processors. Both algorithms are cost optimal in view of the $\Omega(n \log n)$ sequential lower bound for this problem. No optimal parallel algorithms to construct Voronoi diagram of a set of points on an interconnection processors network have been found to date.

There are two reasons. The first reason is, in an interconnection processors network, local computations as well as message exchanges are taken into consideration when analyzing the time taken by a processor network to solve a problem. For example, RAR and RAW operations take constant

time in a sequential computer but RAR and RAW operations take $\log^2 n$ time on hypercube connected computer networks. The second reason is that the memory is no longer shared, but instead, distributed among processors. This prevents the implementation of complex data structures.

For future work, one possible solution is to develop more efficient algorithms for interprocessor message routing. The other possible solution is to implement complex data structures on interconnection processor networks. The most important and powerful feature of the PRAM is the common memory shared by the processors. Not only does the shared memory serve as a communication medium for the processors, but it allows a direct implementation of complex data structures, in a manner very similar to the way they are implemented on the memory of a sequential computer. Therefore, implementing data structures on processor networks is a worthwhile endeavor that deserves to be pursued.

Bibliography

[1] G. Voronoi, Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxiéme Mmoire: Recherches sur les parallélo`edres primitifs, *Journal für die Reine und Angewandte Mathematik*, 134, 1908, 198-287.

[2] M.I.Shamos, Geometric complexity, *Proceedings of the Seventh ACM Symposium on Theory of Computing*, Albuquerque, New Mexico, May 1975, 224-233.

[3] S.Saxena, P.C.P.Bhatt, and V.C. Prasad, Efficient VlSI parallel algorithm for Delaunay triangulation on orthogonal tree network in two and three dimensions, *IEEE Transactions on Computers*, Vol. C-39, No.3, March 1990, 400-404.

[4] K.Q.Brown, Voronoi diagram from convex hulls, *Information Processing Letters*, Vol.9, 1979, 223-228.

[5] A.L.Chow, Parallel algorithms for geometric problems , Ph.D. thesis, University of Illnois at Urbana-Champaign, 1980.

[6] M.Lu, Constructing the Voronoi diagram on a mesh-connected computer, *Proceedings of the 1986International Conference on Parallel Processing*, St.Charles, Illinois, August 1986, 806-811.

[7] W.Preilowski and W.Mumbeck, A time-optimal parallel algorithm for the computing of Voronoi diagrams, Lecture Notes in Computer Science, No.344, J. van Leeuwen(Editor), Springer-Verlag, Berlin, June 1988, 424-433.

[8] C.Levcopoulos, J.Katajainen, and A.Lingas, An optimal expected-time parallel algorithm for Voronoi diagrams, *Lecture Notes in Computer Science*, No.318, Springer-Verlag, Berlin, 1988, 190-198.

[9] D.J.Evans and I.Stojmenovic, On parallel computation of Voronoi diagrams, *Parallel Computing*, Vol.12,1989, 121-125.

[10] C.-S.Jeong, and D.T.Lee, Parallel geometric algorithms on a mesh-connected computer, *Algorithmica*, Vol.5, No.2, 1990,155-178.

[11] S.G.Akl, K.Qiu, I.Stojmenovic, Computing the Voronoi diagram on the star and pancake interconnection networks, *Proceedings of the Fourth Canadian Conference on Computational Geometry*, St.John's, Newfoundland, August, 1992, 353-358.

[12] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays · Tree · Hypercubes*, Morgan Kaufman, San Mateo, California, 1991.

[13] Arthur Trew, Greg Wilson, " Past, Parallel, Parallel", Springer-Verlag, Berlin Heidelberg, 1991.

[14] Q.F.Stout Supporting divide-and-conauer algorithms for image processing *Journal of Parallel and Distributed Computing* Vol.4, pp95-115.

[15] Nassimi, D., Sahni, S., "Data broadcasting in SIMD computers", *IEEE Trans. Computer*, vol.c-30, no.2, pp.101-107, Feb. 1981.

[16] M. J. Flynn, Very high-speed computing systems *Proceedings of the IEEE* 54, Dec., pp. 1901-1909, 1966.

[17] D.Stevenson, Programming the ILLIAC IV, Carnegie-Mellon Unit. Pittsburgh, PA, Tech. Rep., Nov.1975.

[18] Ranka, S., Sahni, S., "Hypercube Algorithms with applications to image processing and pattern recognition", Springer-Verlag, New York, 1990.

[19] Knuth, D., "The art of computer programming: Sorting and searching", vol.3, Addison Wesley, NY, 1973.

[20] F. Harary, Graph Theory Addison-Wesley, Reading, MA, 1969.

[21] K.Q., Geometric transforms for fast geometric algorithms, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.

[22] F.P. Preparata and S.J. Hong, "Convex hulls of finite sets of points in two and three dimensions" *Comm. ACM*2(20), 87-93 (Feb.1977).

[23] D.T.LEE and F.P.PREPARATA, "Location of a point in a planar subdivision and its applications" SIAM J. COMPUT. Vol.6, No.3, September 1977.

[24] C.-S.Jeong, Parallel Voronoi diagram in $L_1(L_\infty)$ metric on a mesh-connected computer, *Parallel Computing*, Vol.17, No.2/3, June 1991, 241-252.

[25] S.Guha, An optimal parallel algorithm for the rectilinear Voronoi diagram, *Proceedings of the Twenty-Eighth Annual Allerton Conference on Communication, Control and Computing,* Monticello, Illinois, October 1990, 789-807.

[26] Y.C. Wee and S.Chaiken, An optimal parallel L_1-metric Voronoi diagram algorithm, *Proceedings of the Second Canadian Conference on Computational Geometry,* Ottawa, Ontario, August 1990, 60-65.