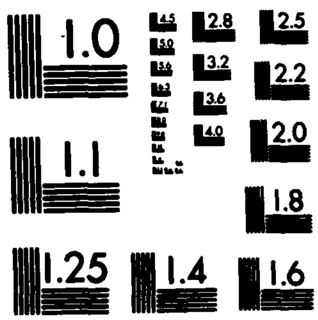
AD-A124 353 PARALLEL ALGORITHMS FOR GEOMETRIC PROBLEMS(U) ILLINOIS UNIV AT URBANA APPLIED COMPUTATION THEORY GROUP A L CHOW DEC 81 ACT-30 N00014-79-C-0424 1/2

UNCLASSIFIED F/G 12/1 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

We shall develop algorithms for solving the rectangle intersection problem on the SMM and on the CCC. As two intermediate steps in our approach, we shall study the problems of reporting intersecting pairs of horizontal

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A124353	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PARALLEL ALGORITHMS FOR GEOMETRIC PROBLEMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER R-927;(ACT-30);UILU-ENG 81-2258
7. AUTHOR(s) Anita L. Chow		8. CONTRACT OR GRANT NUMBER(s) MCS 78-13642 N00014-79-C-0424
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois, 1101 W. Springfield Ave. Urbana, IL 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS National Science Foundation; Joint Services Electronics Program		12. REPORT DATE December 1981
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel computation; shared-memory-machine, cube, cube-connected-cycles; geometric algorithms; convex hulls; geometry of rectangles; planar point location; Voronoi diagrams

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The existence of parallel computing systems and the important applications of geometric solutions have motivated our study on the design and analysis algorithms for solving geometric problems on two parallel computing systems: the Shared Memory Machine (SSM) and the Cube-Connected-Cycles (CCC). The validity of the first SMM resides in uncovering the inherent data-dependence of the problems, while that of the CCC, which complies with the VLSI technological constraints, is the development of practical parallel algorithms. It is shown that solutions to geometric problems can be organized to reveal a large (over)

DD FORM 1 JAN 73 1473

UNCLASSIFIED

20. (Continued)

amount of parallelism, which can be exploited to substantially reduce the computation time. Precisely, using the SMM with a number of processors and memory units linear in the problem size, algorithms are developed to solve problems of reporting intersection of N rectangles in time $O((\log N)^2+k)$, where k is the maximum number of intersections per rectangle, intersection of N rectangles in time $O((\log N)^2)$, planar point location in time $O((\log N)^2 \log\log N)$, finding the two-dimensional convex hull of N points in time $O((\log N)^2)$, the three-dimensional convex hull of N points in time $O((\log N)^3 \log\log N)$, and constructing the planar Voronoi diagram of N points in time $O((\log N)^3 \log\log N)$. Using the CCC with a number of processors linear in the problem size, the parallel algorithms developed for all of these problems, except reporting intersection of rectangles and constructing the two-dimensional convex hull, have time complexity increased only by a factor of $\log N/\log\log N$ with respect to that on the SMM. The algorithms for reporting intersection of rectangles and for constructing the two-dimensional convex hull on the CCC have the same time complexity as that on the SMM. With an increase in the number of processors of the CCC to $N^{1+\alpha}$ $(0 < \alpha \leq 1)$, all of these problems can be solved with algorithms of time complexity improved by a factor of $1/(\alpha \log N)$ with respect to that on the CCC with N processors.

-a-

PARALLEL ALGORITHMS FOR GEOMETRIC PROBLEMS

Anita Liu Chow
Department of Computer Science
University of Illinois at Urbana-Champaign, 1980

ABSTRACT

The existence of parallel computing systems and the important applications of geometric solutions have motivated our study on the design and analysis algorithms for solving geometric problems on two parallel computing systems: the Shared Memory Machine (SMM) and the Cube-Connected-Cycles (CCC). The validity of the first SMM resides in uncovering the inherent data-dependence of the problems, while that of the CCC, which complies with the VLSI technological constraints, is the development of practical parallel algorithms. It is shown that solutions to geometric problems can be organized to reveal a large amount of parallelism, which can be exploited to substantially reduce the computation time. Precisely, using the SMM with a number of processors and memory units linear in the problem size, algorithms are developed to solve problems of reporting intersection of N rectangles in time $O((\log N)^2+k)$, where k is the maximum number of intersections per rectangle,

intersection of N rectangles in time $O((\log N)^2)$, planar point location in time $O((\log N)^2 \log\log N)$, finding the two-dimensional convex hull of N points in time $O((\log N)^2)$, the three-dimensional convex hull of N points in time $O((\log N)^3 \log\log N)$, and constructing the planar Voronoi diagram of N points in time $O((\log N)^3 \log\log N)$. Using the CCC with a number of processors linear in the problem size, the parallel algorithms developed for all of these problems, except reporting intersection of rectangles and constructing the two-dimensional convex hull,

have time complexity increased only by a factor of $\log N/\log\log N$ with

respect to that on the SMM. The algorithms for reporting intersection of rectangles and for constructing the two-dimensional convex hull on the CCC have the same time complexity as that on the SMM. With an increase in the number of processors of the CCC to $N^{1+\alpha}$ $(0 < \alpha \leq 1)$, all of these problems can be solved with algorithms of time complexity improved by a factor of $1/(\alpha \log N)$ with respect to that on the CCC with N processors.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The existence of parallel computers [5,10,15,32,38] has motivated the development of parallel algorithms for solving many problems. These problems include both numerical and non-numerical problems like matrix problems [11,14,36], polynomial evaluation [24,25], arithmetic computation [23], graph problems [3,12,17,33], and sorting [16,29,37]. A recent development in applied computation theory has been the solution of geometric problems by a uniprocessor system [6,8,20,27,34]. It is illustrated in [34] that geometric problems are frequently encountered in operation research, pattern recognition, computer graphics, and statistics.

The topic of this thesis is the study of the solution of geometric problems by parallel computing systems. We shall design and analyze parallel algorithms with references to two systems: the shared memory machine [26] and the cube-connected-cycles [31]. The validity of the first model resides in uncovering the inherent data-dependence of given problems, while that of the second is the development of practical algorithms.

1.1 Parallel Computing Systems

A meaningful study of the design and analysis of parallel algorithms requires a precise model of computation. In this section, we shall describe two systems which are adopted in this thesis.

1.1.1 The Shared Memory Machine (SMM)

Several workers have designed and analyzed efficient parallel algorithms with reference to a shared memory machine [3,10,14,16,17,29, 33,37]. In this model (refer to Figure 1), the processors can communicate with each other through memory. Each processor is capable of performing

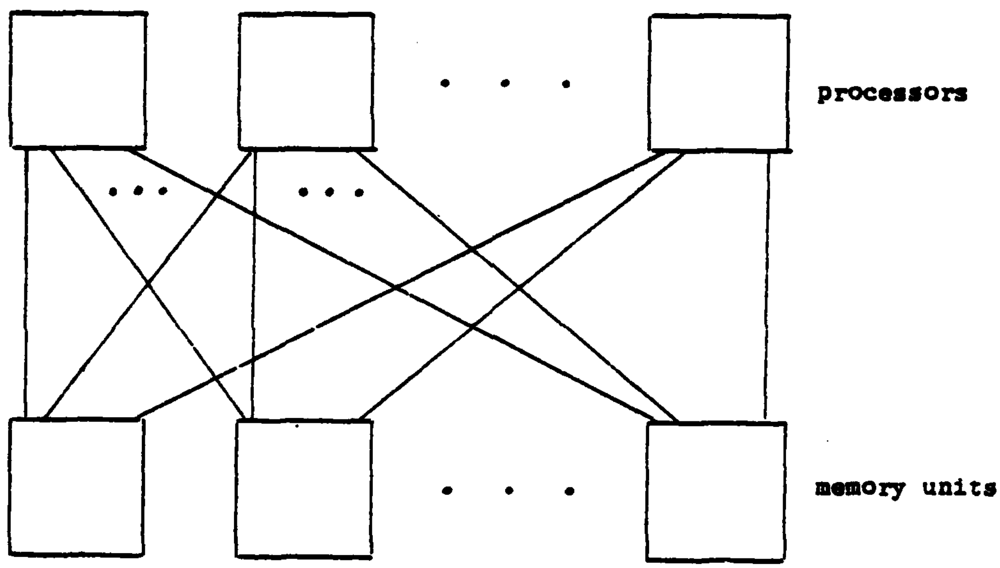


Figure 1. The Shared Memory Machine.

arithmetic operations, boolean operations, comparisons and, possibly, the calculations of trigonometric functions in unit time. The main memory consists of a number of parallel memory units, each of which contains a sufficient number of words. It takes constant time to transmit data from any processor to any memory unit and vice versa. Processors are allowed to simultaneously read from, but not write in, the same word. However, two processors are not permitted to read or write into different words of the same memory unit. (This situation is referred to as memory conflicts.)

We shall assume that the processors are indexed 0 through n-1 and the memory units are indexed from 0 through m-1. Arrays A(0:m-1) of elements A(0),...,A(m-1) are stored systematically in the main memory such that A(i) is in memory unit i.

1.1.2 The Cube Machine (CM) and the Cube-Connected-Cycles (CCC)

In these models there is no shared memory. Each processor has a private RAM memory. Each processor, as in the SMM, is capable of performing arithmetic operations, boolean operations, comparisons and calculating trigonometric functions in unit time.

Assume that $n = 2^k$ and let $BIT_j(a)$ be the $(j+1)^{th}$ least significant bit in the binary expansion of a. In the Cube Machine, the processors are interconnected as a k dimensional cube, that is, processor i is connected to processors $i + (1-2BIT_j(i))2^j$, $0 \leq j < k$. Data may be transmitted from one processor to another only via this interconnection pattern.

Processor i can be identified by a pair of integers (ℓ,p) such that $\ell \cdot 2^r + p = i$ where r is the smallest integer for $r + 2^r \geq k$. In the cube-connected-cycles, which was recently proposed by Preparata and Vuillemin [28], processor (ℓ,p) is connected to processor $(\ell,(p+1) \bmod 2^r)$. $(\ell,(p-1) \bmod 2^r)$ and $\ell_t(1 - 2BIT_p(\ell))2^p,p)$, (refer to Figure 2). The geometric structure underlying the interconnection of the processors is that of a k-dimensional cube, but the CCC requires only three connections per processor. Once again, data transmission from processor to processor is possible only via the available connections.

The development of algorithms with reference to the CCC, unlike that on the SMM which considers only the data-dependence, concerns also the data-movement. Moreover, this machine complies with the present technological constraints of VLSI design [22]. It is shown that the CCC is remarkably suited for implementing efficient algorithms such as Radix-2 Fast Fourier Transform, Bitonic Sorting, etc.

Algorithms for some interesting problems - such as bitonic merge and cyclic shift - perform a sequence of basic operations on data which are successively $2^{k-1}, 2^{k-2}, \ldots, 2^0 = 1$ locations apart. This class of algorithms is referred to as DESCEND class [31]. The dual class ASCEND consists of algorithms which perform a sequence of basic operations on data that are successively $1 = 2^0, 2^1, \ldots, 2^{k-1}$ locations apart. Algorithms in DESCEND class are of the form:

```
for i ← k-1 downto 0 do
    foreach j, 0 ≤ j < 2^k do
        if BIT_i(j) = 0 then OPER(A(j),A(j + 2^i));
```

where OPER(A(j), $A(j+2^i)$) is some basic operation on the operands A(i)

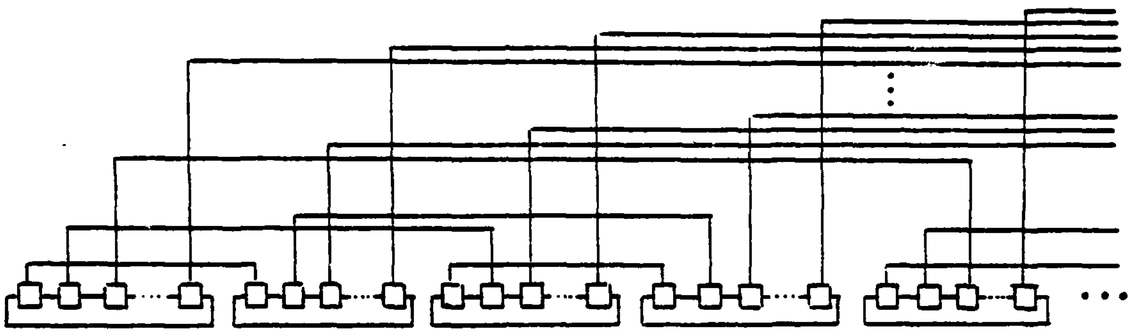


Figure 2. The Cube-Connected-Cycles.

and $A(j+2^i)$. ASCEND differs from DESCEND only in the control loop. The control loop of ASCEND is: <u>for</u> $i \leftarrow 0$ <u>to</u> k-1 <u>do</u>. In both cases, the number of parallel steps on the CM is clearly k. In [28], Preparata and Vuillemin show that algorithms in both classes can be implemented on the CCC in k parallel steps. They also show that other problems (such as <u>permutation</u>, <u>shuffle</u>, <u>unshuffle</u>, <u>bit reversal</u>, <u>odd-even merge</u>, <u>Fast-Fourier-Transform</u>, <u>convolution</u>, <u>matrix transposition</u>) having programs consisting of short sequence of algorithms in the DESCEND or ASCEND classes run in O(k) parallel steps on the CCC. There are also applications - such as <u>bitonic sort</u>, <u>odd-even sort</u>, and calculations of <u>symmetric functions</u> - for which the combining step of the two results of a recursive call is itself an algorithm in the DESCEND or ASCEND class. These algorithms run in $O((\log n)^2)$ parallel steps on the CCC.

1.2 Class of Problems Considered

In this paper, parallel algorithms are presented for several geometric problems, based on the parallel computing systems described in Section 1.1. The geometric problems which are considered here are the following.

We first consider a subproblem of the intersection problems. Given a set of N rectangles with their sides parallel to the coordinate axes, we want to report any pair of rectangles which intersect. Apart from being interesting in its own right, this problem has an important application in VLSI circuitry design rule checking [4,19]. Bentley and Wood [7] recently investigated this problem for a uniprocessor system and developed an $O(N\log N+k)$[(1)] time algorithm for reporting all k such intersecting pairs.

[(1)] All logarithms in this thesis are to the base 2.

We shall develop algorithms for solving the rectangle intersection problem on the SMM and on the CCC. As two intermediate steps in our approach, we shall study the problems of reporting intersecting pairs of horizontal and vertical line segments and of two dimensional range searching. The latter problem is also important in its own right and has applications in the database systems.

The second problem to be studied is an inclusion problem. Given a planar graph embedded in the plane as a straight line graph G [21] with N vertices and a set of M points, for each of these M points, we have to find the region of the planar subdivision induced by G which contains it. In short, we shall refer to this problem as planar point location. This problem is quite important in computational geometry. Indeed, point location is a crucial step in our three-dimensional convex hull algorithms to be developed. The most recent and practical sequential result is due to Preparata [28]. This algorithm runs in time O(MlogN) on a data structure which can be constructed in time O(NlogN).

The next two problems to be investigated are two-dimensional and three-dimensional convex hulls. Given a set S of N points, the convex hull CH(S) of S is the intersection of all convex sets containing S. The convex hull CH(S) is a convex polyhedral region. Chapter 3 of [34] demonstrates the importance of the convex hull problems, which arise in statistics, numerical analysis, and image processing, as well as in many other fields. Preparata and Hong [30] show that the convex hulls of sets of points in both two dimensions or three dimensions can be determined serially with O(NlogN) operations.

The last problem is the construction of the Voronoi diagram for a set of N points in the plane. A Voronoi diagram is a partition of the plane into N polygonal regions, each of which is associated with a given point and is the locus of points closer to the given point than to any other point. This problem arises in clustering analysis [13] and in the context of several closest-point problems [35]. While optimal O(NlogN) serial algorithms exist, we shall consider the construction of Voronoi diagrams on the SMM and on the CCC.

We shall develop algorithms for the above problems on the SMM with a number of processors linear in the problem size and on the cube machine with numbers of processors both linear and superlinear in the problem size. The algorithms that we developed for the cube machine are ASCEND and DESCEND programs, therefore they can be implemented on the CCC without significantly increasing the time complexity.

1.3 Outline of Thesis

In the next chapter we develop some basic tools which will be used in later chapters. Each of the next five chapters is devoted to a problem described in Section 1.2. Each chapter consists of three main algorithms: the first for the SMM and the second for the CCC, both with a number of processors linear in the problem size; the last one for the CCC with a number of processors superlinear in the problem size.

Chapter 3 is on intersection of rectangles. Chapter 4 is on planar point location. Chapters 5 and 6 are on convex hulls in two dimensions and three dimensions respectively. Chapter 7 is on the construction of Voronoi diagrams. In Chapter 8 conclusions are drawn.

CHAPTER 2

BASIC ALGORITHMS

In this thesis, parallel algorithms are sought for various geometric problems. The strategy used to develop an algorithm for a given problem is to devise a technique which reduces the solution of the problem to the solution of a sequence of problems for which efficient parallel algorithms can be developed. In anticipation of later use, we develop some basic parallel algorithms.

2.1 On the SMM with N Processors

We shall discuss the problem of data extraction and the $O((\log N)^2)$ time solution for finding the minimum or maximum of a set of N numbers.

2.1.1 Data Extraction

We consider the following extraction problem. Given an ordered array A(0:N-1) and an associated array t(0:N-1) of tags, we want to move elements A(i), with t(i) = 1, to consecutive memory units in a stable fashion, i.e., preserving the original order.

We first determine the rank R(i) of element A(i), which is the number of elements preceding it and with tags being set to 1. Then elements with tags equal to 1 are moved to consecutive memory units defined by their ranks. We use Nassimi's ranking algorithm: The algorithm is best described recursively. Divide a 2^k element set into two halves, each containing 2^{k-1} consecutive elements. Let R(i) be the rank of A(i) in the 2^{k-1}-set. Let S(i) be the total number of elements in the 2^{k-1}-set containing A(i) with tags equal to one. Then the rank of an element in a 2^k-set is R(i) if $BIT_{k-1}(i)$ equals to 0 (note that $BIT_{k-1}(i) = 0$ for the left 2^{k-1}-set of a 2^k-set) and $R(i) + S(i-2^{k-1})$ if $BIT_{k-1}(i)$ equals to 1. (Note that $S(i-2^{k-1})$ is constant for all terms of the left 2^{k-1}-set.) Unfolding the recursion yields the iterative procedure RANK:

```
procedure RANK(A,t,R):

    /* determine R(i) = number of A(j) for which t(j) = 1 and j < i */
    begin
        foreach i, 0 ≤ i < N do
            begin R(i) ← 0
                if t(i) = 1 then S(i) ← 1 else S(i) ← 0
            end
        for k ← 0 to logN-1 do
            foreach i, 0 ≤ i < N do
                begin T(i + (1-2BIT_k(i))2^k) ← S(i)
                    if BIT_k(i) = 1 then R(i) ← R(i)+T(i)
                    S(i) ← S(i)+T(i)
                end
    end
```

It is easy to see that procedure RANK runs in time O(logN) on a SMM with N processors and N memories. We are now able to describe the entire procedure EXTRACT1. ($|A|$ is the number of elements with tag = 1).

```
procedure EXTRACT1 (A,t):

    /* extract elements A(i) with t(i) = 1 and move them to consecutive
       memory units beginning at unit 0 */
    begin

        /* determine the rank R(i) of each element A(i) */
        call RANK(A,t,R)

        /* route A(i) to R(i) */
        foreach i, 0 ≤ i < N do
            begin T(i) ← A(i)
                if t(i) = 1 then A(R(i)) ← T(i)
            end

        /* determine |A| and fill the right end of A with null */
        if t(N-1) = 0 then |A| ← R(N-1) else |A| ← R(n-1)+1
        foreach i, |A| ≤ i < N do A(i) ← null
    end
```

The time complexity of EXTRACT1 is mainly determined by the first step which calls procedure RANK. Therefore, procedure EXTRACT1 runs in time O(logN) on a SMM with N processors and N memories.

Theorem 2.1. A selected subset of an ordered array A(0:N-1) of elements can be moved to consecutive memory units in a stable fashion in time O(logN) on a SMM with N processors and N memory units.

2.1.2 Finding the Minimum (Maximum) of N Numbers

We now review a well-known O(logN) time algorithm for finding the minimum of a set S of N numbers: we first partition S into two subsets S_1 and S_2 of equal size. We then find the minima m_1 of S_1 and m_2 of S_2 simultaneously. The minimum of S is the smaller number between m_1 and m_2. It can be written as follows.

```
function MINIMUM (S)

    /* returns the minimum of S */
    begin foreach i, 0 ≤ i < N do S'(i) ← S(i)
        for k ← 0 to logN-1 do
            foreach i, 0 ≤ i < N do
                if BIT_k(i) = 0 then
                    if S'(i) > S'(i+2^k) then S'(i) ← S'(i+2^k)
        return (S'(0))
    end
```

Similarly, we can find the maximum of N numbers on a SMM with N processors.

Theorem 2.2. The minimum (maximum) of N numbers can be determined in time O(logN) on a SMM with N processors.

2.2 On the CCC with N Processors

We shall discuss some basic tools like data extraction, selected broadcasting, parallel searching, and finding the minimum (maximum) of N numbers. We shall develop efficient algorithms for these problems on a CCC with a number of processors linear in the problem size.

2.2.1 Data Extraction

Procedure EXTRACT1 described in Section 2.1.1 is not suitable for implementation on the CCC. The step which is causing difficulties is the routing of data to appropriate processors as determined by the data rank. The routing will be referred to as concentration. During concentration, selected data are moved to consecutive processors. Nassimi [26] solved this problem on a CM as follows: Let t(i), when it is equal to 1, be the indicator that data item A(i) is to be moved to the $R(i)^{th}$ processor. First, data A(i), with t(i) = 1, are moved to processors such that the processor index and R(i) agree in bit position 0. The next routing assures that processor indices and R(i) agree in bit positions 0 and 1; and so on until data are routed to the correct processors. Figure 3 is an example of concentration with t(i) = 1 for i = 1,2,4,7. Figure 3(a) shows the initial values of R(i) in binary. The first, second, and third iterations of the above procedure yield the configurations of Figures 3(b), 3(c) and 3(d) respectively. The third iteration completes the concentration.

The formal description of the concentration algorithm is as follows:

```
procedure CONCENTRATE(A,R,t):
    /* route A(i) with t(i) = 1 to processor R(i). This procedure will be
       used to move data A(i) with t(i) = 1 to consecutive processors */
    begin for k ← 0 to logN-1 do
          foreach i, 0 ≤ i < N do
              if t(i) = 1 and BIT_k(i) ≠ BIT_k(R(i))
                  then begin A(i+(1-BIT_k(i))2^k) ← A(i)
                             R(i+(1-BIT_k(i))2^k) ← R(i)
                             t(i+(1-BIT_k(i))2^k) ← t(i)
                       end
    end
```

i	000	001	010	011	100	101	110	111
R(i)		000	001		010			011

(a) initial configuration

i	000	001	010	011	100	101	110	111
R(i)	000			001	010			011

(b) after one iteration

i	000	001	010	011	100	101	110	111
R(i)	000	001					010	011

(c) after two iterations

i	000	001	010	011	100	101	110	111
R(i)	000	001	010	011				

(d) after three iterations

Figure 3. Data extraction with t(i) = 1 for i = 1,2,4,7.

It is straightforward to see that procedure CONCENTRATE can be implemented on a CCC with N processors in O(logN) steps; and procedure RANK, which is introduced in Section 2.1.1 to determine the number of elements with $t(i) = 1$ to the left of each data, can also be carried out on a CCC with N processors in O(logN) steps. We now describe an O(logN) time data extraction algorithm on a CCC with N processors:

```
procedure EXTRACT2 (A,t):
    /* extract A(i) with t(i) = 1 and move them to consecutive processors
       beginning at processor 0. */
    begin call RANK(A,t,R)
        /* determine |A| = number of A(i) with t(i) = 1 */
        if t(N-1) = 0 then |A| ← R(N-1) else |A| ← R(N-1)+1
        call CONCENTRATE (A,R,t)
        /* fill the right end of array A with null */
        foreach i, |A| ≤ i < N do A(i) ← null
    end
```

Theorem 2.3. A selected subset of an ordered array A(0:N-1) of elements can be moved to consecutive memory units in a stable fashion on a CCC with N processors in O(logN) steps.

2.2.2 Selected Broadcasting

Being able to transmit data efficiently is essential for a fast algorithm. We now consider a special case of selected broadcasting. Let P(0: N-1) be a storage array and let $\{a_1,\ldots,a_n\}$ be a selected subset of $\{0,\ldots,N-1\}$, where $a_i < a_{i+1}$. We denote the expression $a_{i+1} - a_i - 1$ by $L(a_i)$ for $i = 1,\ldots,n-1$, and $N - a_{n-1}$ by $L(a_n)$. Our objective is to copy data $D(a_i)$ into $P(a_i), P(a_i+1), \ldots, P(a_i + L(a_i))$ for $i = 1,\ldots,n$. For example, letting $N = 9$, $n = 2$, $a_1 = 2$ and $a_2 = 5$, we would copy D(2) into P(2), P(3), P(4) and D(5) into P(5), P(6), P(7) and P(8).

We shall describe the selected broadcasting procedure along with an example. Let $n = 1$, $N = 16$, $a_1 = 5$, and $L(a_1) = 5$, that is we want to move D(5) to P(5),P(6),...,P(10). In Figure 4 the shaded locations show the data movement in selected broadcasting. Selected broadcasting is carried out by the same routing as in concentration: during the k^{th} iteration, data D(i) is to be copied into P(i+h),P(i+h+1),...,P(i+L(i)), where $h = \min(2^k, L(i))$. Referring to the example, during the 0^{th} iteration, L(5) = 5 indicates that D(5) is to be copied into P(6),P(7),...,P(10); and during the 3^{rd} iteration, L(0) = 10 indicates that D(0) is to be copied into P(8),P(9),P(10). If $L(i) \geq 2^k$, we move data D(i) to the processor such that the processor index and $i + 2^k$ agree in bits 0,1,...,k. Referring to the 1^{st} iteration of the example, D(4) is moved to processor 6; and referring to the 2^{nd} iteration, D(4) is moved to processor 0, such that 0 and $8 = 4+2^2$ agree in bits 0,1,2. During this routing, data may be moving backward (i.e., moving to a processor with lower index) which is contrary to our objective of forward broadcasting. We indicate this transitional state by setting the flag BACKWARD(i) to 1. We have to adjust L(i) by $\pm 2^k$ depending on whether data is moved backward or forward. In the example, D(4) is moved to processor 0 during the 2^{nd} iteration, so the flag BACKWARD(0) is set to 1 and L(0) is assigned to be $L(4) + 2^2 = 10$. When $L(i) < 2^{k+1}$, we know that D(i) will not be moved in later iteration. Moreover, when $0 \leq L(i) < 2^{k+1}$ and D(i) is not in the backward transitional state, we can copy D(i) into P(i) and set L(i) to -1. Referring to the 1^{st} iteration of the example,

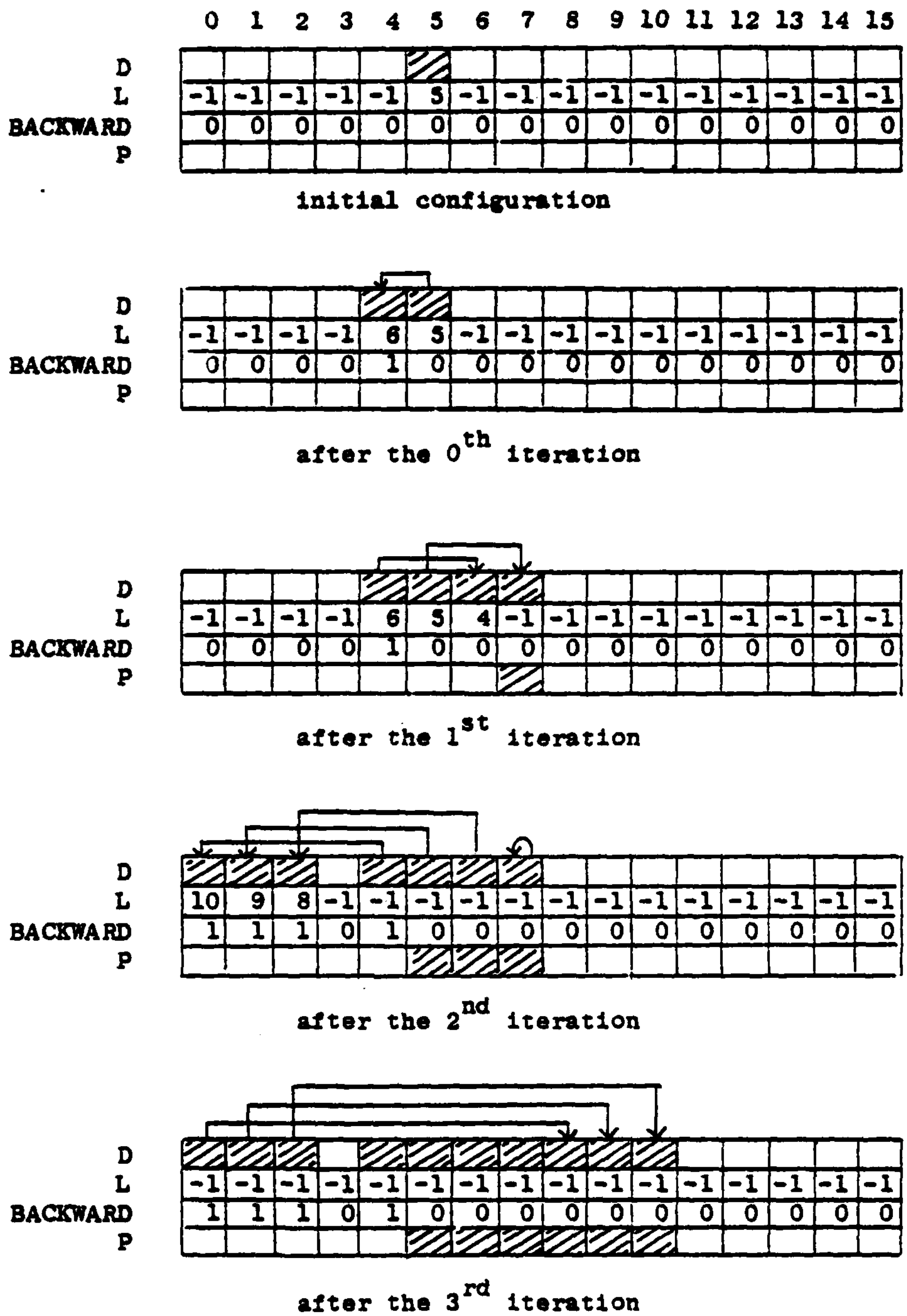


Figure 4. Broadcasting D(5) to P(5), P(6), ..., P(10).

L(7) is first set to 3, so $0 \le L(7) < 2^2$ and BACKWARD(7) is 0, then we can copy D(7) into P(7) and set L(7) to -1. We claim that at the end of (logN+1) iterations, the broadcasting is complete. The program for the selected broadcasting is as follows:

```
procedure SELECTED_BROADCASTING(D,L,P)
     /* when L(i) > 0, copy D(i) into P(i),P(i+1),...,P(i+L(i)).
        BACKWARD will be a flag for backward transitional stage.
        T,TL,BACKWARD will be used as temporary storage for D,L,
        BACKWARD respectively */
     begin
          foreach i, 0 ≤ i < N do
               begin TL(i) ← -1, BACKWARD(i) ← 0 end
          for k ← 0 to log N-1 do
               foreach i, 0 ≤ i < N do
                    begin

                         /* move D(i) to the processor such that the
                            processor index and the destination agree in
                            bits 0,1,...,k */
1.                       if L(i) ≥ 2^k then
                            begin T(i+(1-2BIT_k(i))2^k) ← D(i)
                                  TL(i+(1-2BIT_k(i))2^k) ← L(i)+(2BIT_k(i)-1)2^k
                                  TBACKWARD(i+(1-2BIT_k(i))2^k) ← BIT_k(i)
                            end

                         /* determine if data in D(i) is permanent,
                            discarded or have to be saved */
2.                       if 0 ≤ L(i) < 2^(k+1) then
                            begin if BACKWARD(i) = 0 then P(i) ← D(i)
                                  L(i) ← -1
                            end

                         /* determine if data in temporary location T(i) is
                            permanent, can be discarded or have to be saved */
3.                       if 0 ≤ TL(i) < 2^(k+1) then
                            begin if TBACKWARD(i) = 0 then P(i) ← T(i)
                                  TL(i) ← -1
                            end
```

```
4.                      if TL(i) ≥ 2^(k+1) then
                            begin D(i) ← T(i)
                                  L(i) ← TL(i)
                                  BACKWARD(i) ← TBACKWARD(i)
                                  TL(i) ← -1
                            end
                    end
      end
```

The correctness of SELECTED_BROADCAST is not immediate. We must show that (1) whenever data is to be stored at some location, the previous imformation at that location can be discarded; (2) $D(a_i)$ is moved to $P(a_i),\dots,P(a_i+L(a_i))$ for $i = 1,\dots,n$ at the termination of the procedure.

Theorem 2.4. Procedure SELECTED_BROADCAST is correct.

Proof. It is observed that at the beginning of each iteration $TL(i) = -1$, $\forall i$; so prior to step 1, information at T(i), TL(i) and TBACKWARD(i) can be discarded for $\forall i$.

Suppose $BIT_k(i) = 0$ and $L(2^k+i) \geq 2^k$ at step 1. Then TL(i) is assigned the value $L(2^k+i) + 2^k \geq 2^{k+1}$ and by the specification of the problem, $L(i) < 2^{k+1}$. At step 2, L(i) is then set to -1 which implies that prior to step 4, information at D(i), L(i), BACKWARD(i) can be discarded. Suppose $BIT_k(i) = 1$ and $L(i) \geq 2^k$ at step 1. By the specification of the problem, $L(i-2^k) < 2^{k+1}$ at step 1. TL(i) may be set to $L(i-2^k) - 2^k < 2^k$ or remains -1 depending on the value of $L(i-2^k)$; in either case TL(i) is -1 at the completion of step 3. Therefore, step 4 has no storage conflicts.

To complete the proof, it is now sufficient to show for $n = 1$, $D(a_1)$ is correctly moved to $P(a_1),\ldots,P(a_1+L(a_1))$ and data $D(a_1)$ is never moved to $P(i)$, for $i \notin \{a_1,\ldots,a_1+L(a_1)\}$ during the process. It is simple to see the routing in the algorithm guarantees $D(a_1)$ reaches processors $a_1, a_1+1,\ldots,a_1+L(a_1)$. Indicators BACKWARD(i) and TBACKWARD(i) determine whether a piece of data arrives at processor i should be written into P(i). If the data is arriving from a processor with higher index then this data is in a transitional stage, otherwise this data is in its destination. □

Procedure SELECTED_BROADCAST runs in time O(logN) on a CCC with N processors.

Theorem 2.4. Given a subset $\{a_1,\ldots,a_n\}$ of $\{0,\ldots,N-1\}$ and $a_i < a_{i+1}$, data items $D(a_i)$ can be copied into $P(a_i),P(a_i+1),\ldots,P(a_i+L(a_i))$, where $L(a_i) = a_{i+1}-a_i-1$, for $i = 1,\ldots,n$, in time O(logN) on a CCC with N processors.

2.2.3 Parallel Searching

Given an array A(0:N-1) of N elements in ascending order and a set Q(0:M-1) of test elements, we want to find for each i, $0 \le i < M$, $A(j_i)$ such that $A(j_i) \le Q(i) < A(j_i+1)$. We present the set of test elements in descending order. Then A and Q are merged using Batcher's bitonic merge. Then A(j) is broadcast to all the test elements between A(j) and A(j+1) in the resulting merged sequence of A and Q. For example, N = 4, M = 5, A(0),...,A(3) are 1, 3, 4, 8 respectively, and Q(0),...,Q(4) are 1, 2, 4, 5, 6 respectively. Figure 5(a) shows the sequences. Figure 5(b) shows the merged sequence. Then A(0) is broadcast to Q(0), Q(1), and A(2) is broadcast to Q(2), Q(3), Q(4).

A(0)	A(1)	A(2)	A(3)
1	3	4	8

Q(0)	Q(1)	Q(2)	Q(3)	Q(4)
1	2	4	5	6

(a) sequences A and Q

A(0)	Q(0)	Q(1)	A(1)	A(2)	Q(2)	Q(3)	Q(4)	A(3)
1	1	2	3	4	4	5	6	8

(b) merged sequence

Figure 5. Parallel searching.

The following program performs the parallel searching.

```
procedure SEARCH (A,Q,P):

    /* determine P(i) = A(j_i) such that A(j_i) ≤ A(j_i+1) */
    begin

        /* merge sequences A and Q */
        foreach i, 0 ≤ i < N do  D(i) ← A(i)
        foreach i, 0 ≤ i < M do  D(N+i) ← Q(i)
        apply bitonic merge to D;

        /* determine the distance L(i) such that D(i) has to be broadcast */
        foreach i, 0 ≤ i < N+M do
            begin t(i) ← 0; L(i) ← -1
                if D(i) ∈ A and D(i+1) ∈ Q
                    then begin t_1(i) ← 1; FIRST(i) ← i end
            end
        call EXTRACT2 (FIRST,t)
        foreach i, 0 ≤ i < N+M do
            if FIRST(i) ≠ null then L(i) ← FIRST(i+1)-FIRST(i)-1
        move L(i) to processor FIRST(i) by a procedure similar to CONCENTRATE

        /* broadcast D(i) to P(i),...,P(i+L(i)) */
        call SELECTED_BROADCAST (D,L,P).

        /* move P to origi- position */
        foreach i, 0 ≤ i < N+M do
            if D(i) ∈ A then t(i) ← 1 else t(i) ← 0
        call EXTRACT2(P,t)
    end
```

This procedure runs in time $O(\log(N+M))$ with N+M processors. Therefore, parallel searching runs in time $O((\log M)^2 + \log(N+M))$ on a CCC with N+M processors.

__Theorem 2.5__. Given an ordered array $A(0:N-1)$ of N elements and a set $Q(0:M-1)$ of test elements, for each i, $0 \le i < M$, the element $A(j_i)$, such that $A(j_i) \le Q(i) < A(j_i+1)$, can be determined in time $O((\log M)^2 + \log(M+N))$ on a CCC with $N+M$ processors.

2.2.4 Finding the Minimum (Maximum) of N Numbers

The algorithm presented in Section 2.1.2 for finding the minimum (maximum) of N numbers is directly within the ASCEND class. Therefore, we have the following result.

Theorem 2.6. The minimum (maximum) of N numbers can be determined in time O(logN) on a CCC with N processors.

CHAPTER 3

INTERSECTION OF RECTANGLES

Given a set of N rectangles (with sides parallel to the coordinate axes) in the plane, we are asked to report all pairs of rectangles which intersect. An important application of the problem is in VLSI design rule checking [4,19]. Bentley and Wood [7] presented an O(NlogN+k) (optimal) time algorithm for reporting intersections of rectangles on a uniprocessor machine, where k is the number of intersecting pairs found. In this chapter we investigate this problem on parallel computing machines.

Our approach to a parallel solution of the problem follows the general approach of Bentley and Wood and requires two intermediate steps: reporting intersections of horizontal and vertical line segments, and two-dimensional range searching. Two rectangles intersect if their edges intersect or one rectangle entirely encloses the other. The problem of finding rectangle enclosure can be reduced to that of two-dimensional range searching as follows. We associate with each rectangle A a representative point a in its interior, for example, its leftmost bottom vertex. If point a lies within rectangle B, then either B entirely encloses A or A and B have an edge intersection.

The rectangles in the given set are indexed 0 to N-1. Each rectangle r is defined by four reals giving its bottom B(r), top T(r), left L(r) and right R(r) extreme points.

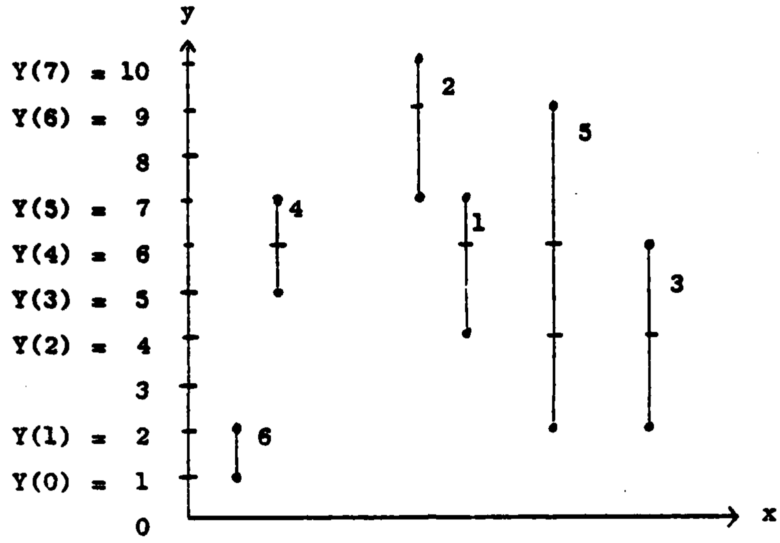
3.1 On the SMM with N Processors

In this section we shall present an algorithm which solves the rectangle intersection problem in time $O((\log N)^2 + k)$ on a SMM with N processors, where k is the maximum number of intersections per rectangle. We shall discuss two intermediate problems: intersection of horizontal and vertical line segments, and two-dimensional range searching.

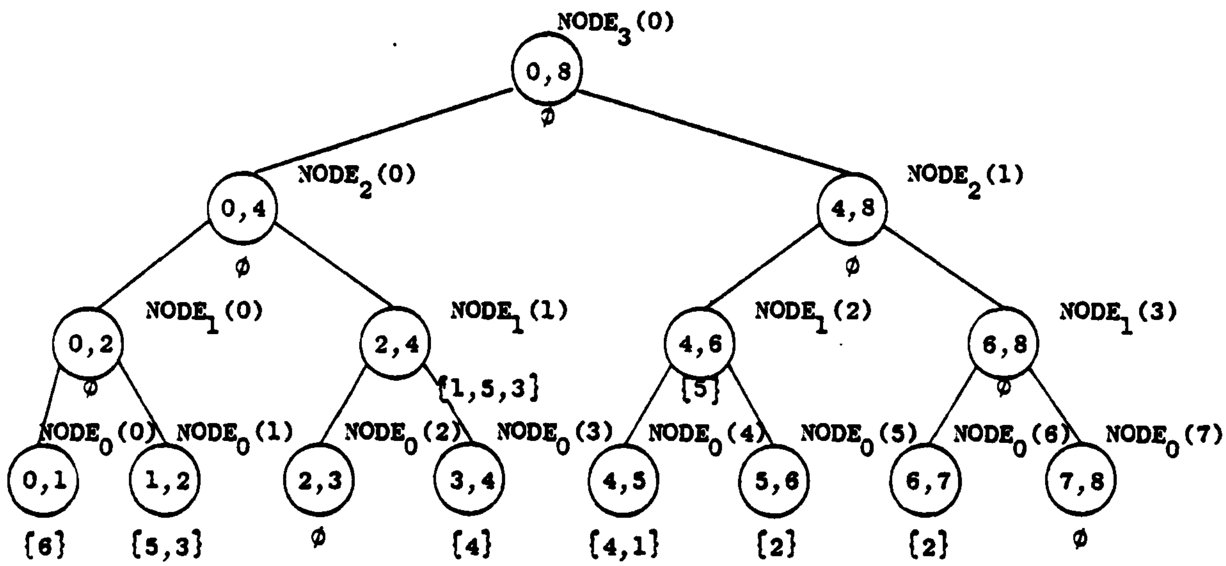
3.1.1 Intersection of Horizontal and Vertical Line Segments

Given a set V(0:n-1) of n vertical line segments and a set H(0:m-1) of m horizontal line segments, we want to report all pairs of vertical and horizontal line segments which intersect. V(i) and H(i) are records. In addition to the endpoint information, each V(i) contains two redundant fields B and T: V(i)[B] and V(i)[T] are the y-values of the bottom and top endpoints of V(i), respectively. H(i) also contains two fields L and R: H(i)[L] and H(i)[R] are the x-values of the left and right endpoints of H(i), respectively. Let Y(0:N-1) be a sorted array of distinct y-values of the endpoints of the vertical line segments, where $N \le 2n$ (refer to Figure 6(a)). We assume, for simplicity, that $N+1$ is a power of 2 and $Y(N+1) = Y(N)+1$; the details of the general case are straightforward.

We now describe the search tree $\mathcal{T}$ which can be produced for the set of vertical line segments. $\mathcal{T}$ is a binary tree of height $\log(N+1)$. In $\mathcal{T}$ $NODE_i(j)$ denotes the j^{th} leftmost node at height i; it represents an interval $[B_i(j), T_i(j)]$ where $B_i(j) = Y(j \cdot 2^i)$ and $T_i(j) = Y((j+1)2^i)$. If $i > 0$, $NODE_i(j)$ has two sons: $NODE_{i-1}(2j)$ and $NODE_{i-1}(2j+1)$. Each $NODE_i(j)$ contains a list of edges V(k) sorted in the positive x-direction where $V(k)[B] \le B_i(j)$ and $T_i(j) \le V(k)[T]$. Moreover V(k) does not belong to any ancestor of $NODE_i(j)$. Figure 6(b) is the search tree $\mathcal{T}$ for the set of vertical lines in Figure 6(a); pairs of integers in the circles are values of $j \cdot 2^i$ and $(j+1)2^i$.



(a) A set of vertical line segments and the corresponding Y array. (the "cuts" on the edges show the logarithmic segmentation for $\mathcal{T}$ and $\mathcal{S}$)



(b) Search tree $\mathcal{T}$ for the vertical line segments in (a)

Figure 6. Search tree $\mathcal{T}$ for vertical line segments.

We define C_{ij} as a list of candidate segments for $NODE_i(j)$ sorted in the positive x-direction. We shall construct $\mathcal{T}$ level by level beginning from the root. From $C_{\log(N+1),0}$, which is a list of all the vertical line segments sorted in the positive x-direction, we extract segments which lie in the range $[Y(0),Y(N)]$. This list of extracted segments is associated with $NODE_{\log(N+1)}(0)$. From the remaining segments in $C_{\log(N+1),0}$, we determine $C_{\log(N+1)-1,0}$ and $C_{\log(N+1)-1,1}$ as follows. Edge $C_{\log(N+1),0}(k)$ belongs to $C_{\log(N+1)-1,0}$ if $C_{\log(N+1),0}(k)[B] < T_{\log(N+1)-1}(0)$ and to $C_{\log(N+1)-1,1}$ if $C_{\log(N+1),0}(k)[T] > B_{\log(N+1)-1}(1)$. We repeat this procedure for constructing the set of $NODE_i(j)$ for every j in each level i. Given $C_{i,j}$, all of the three lists $NODE_i(j)$, $C_{i-1,2j}$ and $C_{i-1,2j+1}$ can be determined in $O(\log|C_{i,j}|)$ steps with $|C_{i,j}|$ processors. At each level i, every line segment can belong to at most four $C_{i,j}$. Therefore $\sum_{j=0}^{2^i-1} |C_{i,j}| \le 4n$. Thus, 4n processors and O(logn) time are sufficient to construct one level of $\mathcal{T}$. $\mathcal{T}$ has log(N+1)+1 levels, so $\mathcal{T}$ can be constructed in $O((\log N)^2)$ time with 4n processors and 4n memories. The following program CONSTRUCT_$\mathcal{T}$1 constructs $\mathcal{T}$ for vertical line segments. (A different program CONSTRUCT_$\mathcal{T}$2 will be written to construct $\mathcal{T}$ for edges of a planar graph.)

<u>procedure</u> CONSTRUCT_$\mathcal{T}$1(V)

```
    /* construct the point location tree T for the vertical line segments V */
    being sort V(0:n-1) by x-values and y-values of bottom endpoints
        foreach k, 0 ≤ k < n, do C_{log(N+1),0}(k) ← V(k)

        /* construct T level by level */
        foreach j, 0 ≤ j < 2^i-1 do
            begin NODE_i(j) ← C_{i-1,2j} ← C_{i-1,2j+1} ← ∅
                if C_{i,j} ≠ ∅ then
                    begin
```

```
/* determine NODE_i(j) by extracting the
   appropriate edges from C_{i,j} */
foreach k, 0 ≤ k < |C_{i,j}| do
    begin C_{i-1,2j}(k) ← C_{i-1,2j+1}(k) ← C_{i,j}(k)

          t(k) ← 0
          if C_{i,j}(k)[B] ≤ B_i(j) and
             T_i(j) ≤ C_{i,j}(k)[T]
               then t (k) ← 1
    end
call EXTRACT1(C_{i,j},t)
NODE_i(j) ← C_{i,j}

/* determine C_{i-1,2j} and C_{i-1,2j+1} by extracting
   edges from the remaining of C_{i,j} */
foreach k, 0 ≤ k < |C_{i-1,2j}| do
    begin
          if t = 0 and C_{i-1,2j}(k)[B] < T_{i-1}(2j)
               then t_1 ← 1 else t_1 ← 0
          if t = 0 and C_{i-1,2j}(k)[T] > B_{i-1}(2j+1)
               then t_2 ← 1 else t_2 ← 0
          end
    end
call EXTRACT1(C_{i-1,2j},t_1)
call EXTRACT1(C_{i-1,2j+1},t_2)
end
end
```

To find all the intersections of a horizontal line segment $H(k)$ with the set V of vertical line segments, we use $\mathcal{T}$ as a two-dimensional binary search tree: At a selected node $NODE_i(j)$ of $\mathcal{T}$, we report all the vertical segments in the list of $NODE_i(j)$ which are in the interval $[H(k)[L],H(k)[R]]$. Since the vertical segments at $NODE_i(j)$ are sorted by their x-values, the search can be done in $O(\log n+k')$, where k' is the number of intersections per segment reported in one level. In the next step, we proceed to one son or both of $NODE_i(j)$ by comparing the y-value of H(k) with $T_{i-1}(2j)$: if y-value of H(k) is less than, greater than or equal to $T_{i-1}(2j)$ then we proceed respectively to the left son, the right son or both sons. At the selected son, we again report all the vertical line segments in the list of this node which intersect with the horizontal line segment H(k). We continue this process until we reach the bottom of $\mathcal{T}$. Note that the y-value of H(k) may be equal to only one $T_{i-1}(2j)$. Thus, we trace a unique path, possibly two, from the root to the bottom level; at that stage all intersections k" of segment H(k) are reported. Since $\mathcal{T}$ is of height O(logN), this process runs in time $O((\log n)^2+k'')$. We can find intersections of all m horizontal lines with V simultaneously, provided we search in one level of $\mathcal{T}$ for all horizontal lines before going to the next level. The number of processors required is m for parallel searching. Thus, we have the following result.

Theorem 3.1. All intersecting pairs of n vertical line segments and m horizontal line segments can be reported in time $O((\log n)^2+k)$ on a SMM with max(4n,m) processors and max(4n,m) memory units, where k is the maximum number of intersection of any horizontal line segment and the set of vertical line segments.

The formal description of the intersection algorithm is as follows.

```
procedure INTERSECT1(V,H):

    /* find all intersecting pairs of horizontal line segments in H and
       vertical line segments in V */
    begin

        /* construct the point location tree 𝒯 for V */
        call CONSTRUCT_𝒯1(V)
        foreach k, 0 ≤ k < m do begin J_0(k) ← 0 ; J_1(k) ← -1 end

        /* search in 𝒯 level by level */
        for i ← log(N+1) downto 0 do
            for p ← 0 to 1 do
                foreach k, 0 ≤ k < m do
                    if J_p(k) ≥ 0 then
                        begin search in NODE_i(j) all vertical lines
                                   in the range [H(k)[L],H(k)[R]]
                              if y-values of H(k) = T_{i-1}(2J_p(k))
                                  then begin J_p(k) ← 2J_p(k)

                                             J_{p⊕1}(k) (1) ← 2J_p(k) + 1
                                       end
                                  else if y-value of H(k) < T_{i-1}(2J_p(k))
                                       then J_p(k) ← 2J_p(k)
                                       else J_p(k) ← 2J_p(k) + 1
                        end
    end
```

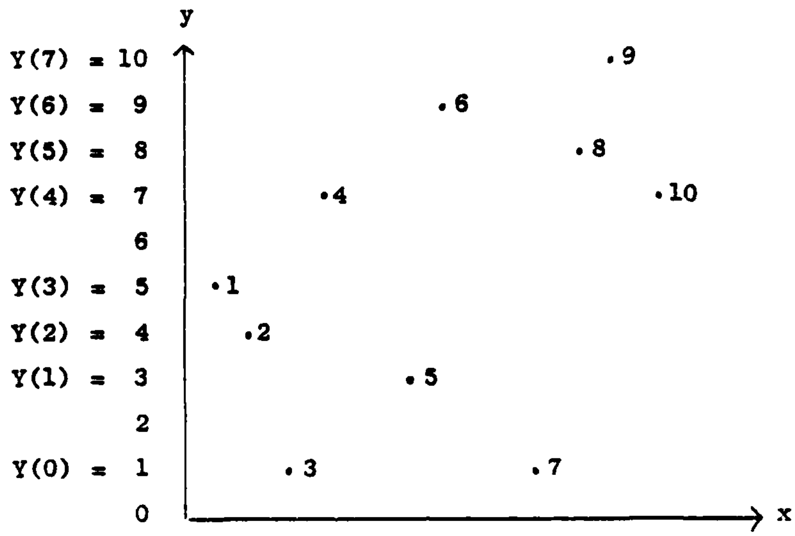
3.1.2 Range Searching

We are given a set S of n points in the plane and a set Q of queries: report all points of S in the range $Q(i)[L] \le x \le Q(i)[R]$ and $Q(i)[B] \le y \le Q(i)[T]$. We first organize the points in S so that we can answer the queries efficiently.

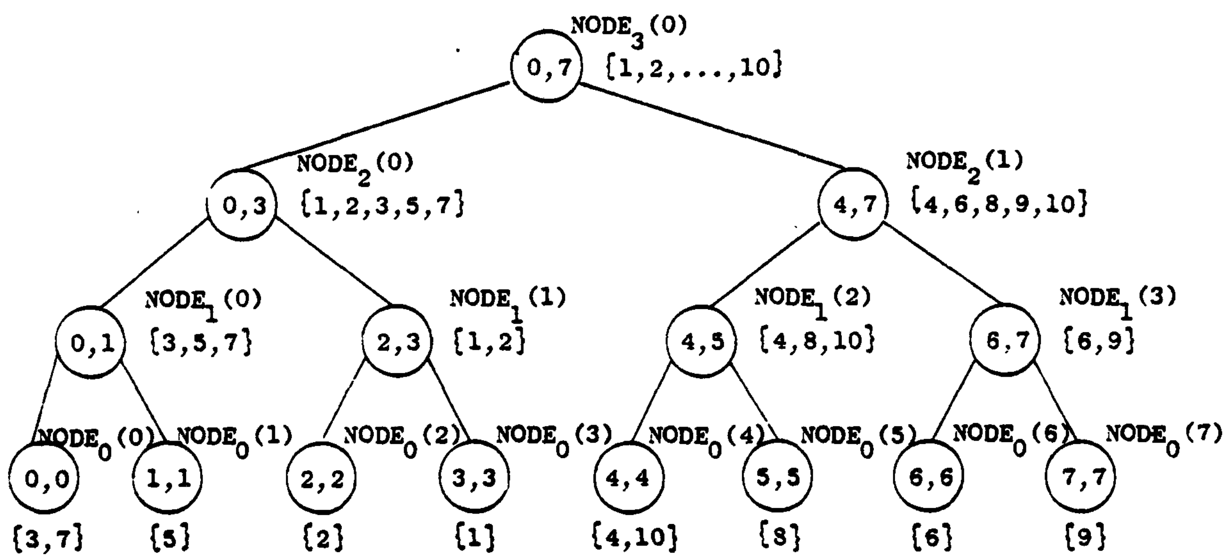
(1) $\oplus$ is the exclusive-or operator.

We assume that Y(0:N-1) is a sorted array of the distinct y-values of points in S, where $N \leq n$. We also assume that N is a power of 2. We construct a search tree $\mathcal{H}$ for the set of points. $\mathcal{H}$ is similar to $\mathcal{T}$, but with the following differences. Associated with $NODE_i(j)$ is a subset of points with their y-values in the interval $[B_i(j), T_i(j)]$, sorted by their x-values, where $B_i(j) = Y(j \cdot 2^i)$ and $T_i(j) = Y((j+1)2^i - 1)$. Figure 7 is an example of search tree $\mathcal{H}$. $NODE_{\log N}(0)$, the root, is the entire set S sorted by x-values. We use procedure EXTRACT1 to partition $NODE_{\log N}(0)$ into $NODE_{i-1}(2j)$ and $NODE_{i-1}(2j+1)$ such that all points in $NODE_{i-1}(2j)$ have y-values $\leq T_{i-1}(2j)$ and those in $NODE_{i-1}(2j+1)$ have y-values $\geq B_{i-1}(2j+1)$. Again, like in the construction of $\mathcal{T}$, $\mathcal{H}$ is constructed level by level. Since $\sum_{j=0}^{2^i-1} |NODE_i(j)| = n$ for all i, $\mathcal{H}$ can be constructed in time $O((\log n)^2)$ with n processors.

```
procedure CONSTRUCT_H(S):

     /* determine, from S, NODE_i(j) of H */
     begin
          sort S(0:n-1) by their x-values
          NODE_logN(0) ← S

          /* determine nodes of H level by level */
          for i ← logN downto 1 do
               foreach j, 0 ≤ j < 2^i do
                    begin
                         /* partition points of NODE_i(j) into NODE_{i-1}(2j)
                            and NODE_{i-1}(2j+1) according to their y-values */
                              NODE_{i-1}(2j) ← NODE_{i-1}(2j+1) ← NODE_i(j)
                              foreach a ∈ NODE_i(j) do
                              if y-values of a ≤ B_{i-1}(2j)
                                   then t_1(a) ← 1 ; t_1(a) ← 0
                                   else t_2(a) ← 1 ; t_1(a) ← 0
                              call EXTRACT1(NODE_{i-1}(2j), t_1)
                              call EXTRACT1(NODE_{i-1}(2j+1), t_2)
                    end
     end
```



(a) A set of points and the corresponding Y array



(b) Search tree $\mathcal{H}$ for points in (a)

Figure 7. Search tree $\mathcal{H}$ for points in the plane.

Given a query $Q(k)$, we search in $\mathcal{H}$ starting with the root until we reach a $NODE_i(j)$ such that $Q(k)[B] \le B_i(j) \le T_i(j) \le Q(k)[T]$. Then we report all points in $NODE_i(j)$ with x-values in the interval $[Q(k)[L],Q(k)[R]]$. Since points in $NODE_i(j)$ are ordered by their x-values, the query is answered in $O((\log n)^2 + k')$ time with 1 processor where k' is the number of inclusions. All m queries can be treated in parallel if we search in one level of $\mathcal{H}$ for all queries at a time. Therefore we have the following result for range searching:

Theorem 3.2. The two-dimensional range searching problem for n data and m queries can be solved in time $O((\log n)^2 + k)$ on a SMM with max(n,m) processors and memory units, where k is the maximum number of inclusions per query.

```
procedure RANGE_SEARCH1(S,Q)
    /* report all points a ∈ S such that Q(i)[L] ≤ x(a) ≤ Q(i)[R] and
       Q(i)[B] ≤ y(a) ≤ Q(i)[T], for every Q(i) ∈ Q */
    begin
        /* construct the search tree ℋ for the set S of points */
        call CONSTRUCT_ℋ(S)
        foreach k, k ≤ 0 < m do J_logN(k) ← {0}
        /* search in ℋ, beginning at the root */
        for i ← logN downto 0 do
            foreach k, k ≤ 0 < m do
                begin J_i-1(k) ← ∅
                    for each j ∈ J_i(k) do
                        begin if Q(k)[B] ≤ B_i(j) and T_i(j) ≤ Q(k)[T]
                            then search in NODE_i(j) and report any
                                pair (Q(k),a) where
                                Q(k)[L] ≤ x(a) ≤ Q(k)[R], a ∈ NODE_i(j)
                            else begin if Q(k)[B] ≤ T_i-1(2j)
```

```
                                        then J_{i-1}(k) ← J_{i-1}(k) ∪ {2j}
                                    if Q(k)[T] ≥ B_{i-1}(2j+1)
                                        then J_{i-1}(k) ← J_{i-1}(k) ∪ {2j+1}
                              end
                        end
                  end
      end
```

3.1.3 The Rectangle Intersection Algorithm

In previous subsections of this section we have investigated the rectangle intersection problem in a top-down fashion. Procedure RECTINT1(REC) is the complete description of the entire algorithm for reporting all pairs of intersections of rectangles REC. Another two programs (RECTINT2 and RECTINT3) will be written for the CCC.

```
procedure RECTINT1(REC):
     begin
          V ← all vertical edges of rectangles in REC
          H ← all horizontal edges of rectangles in REC
          call INTERSECT1(V,H)
          S ← all left bottom points of rectangles in REC
          Q ← REC
          call RANGE_SEARCH1(S,Q)
     end
```

Combining the results in previous subsections, we can show that RECTINT1 runs in time $O((\log N)^2+k)$ on a SMM with 8N processors and memories, where N is the number of rectangles and k is the maximum number of intersections per rectangle. However, a simple-minded processor-time tradeoff can reduce the number of processors to N by increasing the time by a factor of 8 as follows. We can position the set of vertical edges into eight subsets, each of which has N/8 edges. We then find the intersections of the set of horizontal edges with each of these eight subsets of vertical edges sequentially. We conclude this section by the following theorem.

<u>Theorem 3.3</u>. Given N rectangles with edges parallel to the coordinate axes, all intersecting pairs of these rectangles can be reported in time $O((\log N)^2 + k)$ on a SMM with N processors and N memories, where k is the maximum number of intersections per rectangle.

3.2 On the CCC with N Processors

In this section we shall present an algorithm which solves the rectangle intersection problem in time $O((\log N)^2 + k)$ on a CCC with N processors, where k is the maximum number of intersections per rectangle. We shall first discuss three intermediate problems: one-dimensional range searching, intersection of horizontal and vertical line segments, and two-dimensional range searching.

3.2.1 One-Dimensional Range Searching

Given a set A(0:N-1) sorted in ascending order and a set Q(0:M-1) of queries specified by two bounds [L] and [R] (left and right respectively), we want to report all elements of A which lie in the range [Q(i)[L], Q(i)[R]] $0 \le i < M$. We approach this problem by first finding $A(j_i)$ such that $A(j_i-1) < Q(i)[L] \le A(j_i)$, for each i, and then reporting sequentially the pairs $(Q(i),A(j_i)),(Q(i),A(j_i+1)),\ldots,(Q(i),A(\bar{j}_i))$ where $A(\bar{j}_i) \le Q(i)[R] < A(\bar{j}_i+1)$: we assume that Q is sorted by the values of the left bounds in ascending order. We then merge A and Q. We perform a parallel search, similar to the one introduced in Section 2.2.3, for determining $A(j_i)$ for all Q(i). Before reporting any inclusions, we eliminate those queries which do not have any inclusion (i.e., if $Q(i)[R] < A(j_i)$) from further consideration. We report sequentially all the inclusions for every query.

For example, consider the case where N = 7, M = 4 and the sequences of A and Q are as shown in Figure 8(a). Figure 8(c) is the merged sequence with Q(1) eliminated as Q(1)[R] = 4 < A(3) = 5, i.e., none of the A's lies in the range [Q(1)[L],Q(1)[R]]. We then start to report all inclusions by looking to the right simultaneously for every query: (Q(0),A(2)), (Q(2),A(3)) and (Q(3),A(5)) are reported first; next (Q(0),A(3)), and (Q(2),A(4)) are reported at the same time; then (Q(0),A(4)), (Q(0),A(5)) are reported one at a time.

```
procedure RANGE_SEARCH_1D(A,Q)

     /* A(0:N-1) is a sorted array, Q(0:M-1) is a set of queries sorted
        by values of Q(i)[L]. Report all elements of A which lie in
        [Q(i)[L],Q(i)[R]] for i = 0,...,M-1 */
     begin

          /* copy information of A and Q into D */
          foreach i, 0 ≤ i < M do begin D(i)[type] ← query
                                        D(i)[key] ← Q(i)[L]
                                        D(i)[k] ← Q(i)[R]
                                   end  D(i)[value] ← Q(i)

          foreach i, 0 ≤ i < M do begin D(M+1)[type] ← data
                                        D(M+1)[key] ← A(i)
                                        D(M+i)[value] ← A(i)
                                   end
          apply bitonic merge to D
          determine P such that P(i) = A(j_i) and A(j_i-1) < Q(i) ≥ A(j_i)

          /* eliminate those queries which do not have inclusions */
          foreach i, 0 ≤ i < N+M do
               if D(i)[type] = query and D(i)[R] < P(i)
                    then t(i) ← 0
                    else t(i) ← 1
          call EXTRACT2 (D,t)

          /* report inclusions */
          foreach i, 0 ≤ i < N+M do
               begin T(i) ← null
                    if D(i)[type] = query then T(i) ← D(i)[value]
               end
          while ∃ T(i) ≠ null do
```

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)
0	1	2	5	6	7	8

Q(0)	Q(1)	Q(2)	Q(3)	
2	3	3	7	[L]
7	4	6	7	[R]

(a) initial sequences A and Q

A(0)	A(1)	Q(0)	A(2)	Q(1)	Q(2)	A(3)	A(4)	Q(3)	A(5)	A(6)
0	1	2	2	3	3	5	6	7	7	8
		7		4	6			7		

(b) the merged sequence

A(0)	A(1)	Q(0)	A(2)	Q(2)	A(3)	A(4)	Q(3)	A(5)	A(6)
0	1	2	2	3	5	6	7	7	8
		7		6			7		

(c) Q(1) being eliminated

(Q(0),A(2)),(Q(2),A(3)),(Q(3),A(5))

(Q(0),A(3)),(Q(2),A(4))

(Q(0),A(4))

(Q(0),A(5))

time

(d) the pairs in each row are reported simultaneously

Figure 8. One-dimensional range searching.

```
begin for j ← 1 to ℓ do /* ℓ = loop length of CCC */
    foreach i, 0 ≤ i < N+M do
        begin
            if i mod ℓ ≥ j-1 & D(i)[type] = data
                then if T(i)[R] ≥ D(i)[value] then
                        report (T(i),d(i)[value])
                     else T(i) ← null
            T(i+1 mod ℓ) ← T(i)
        end
end
```

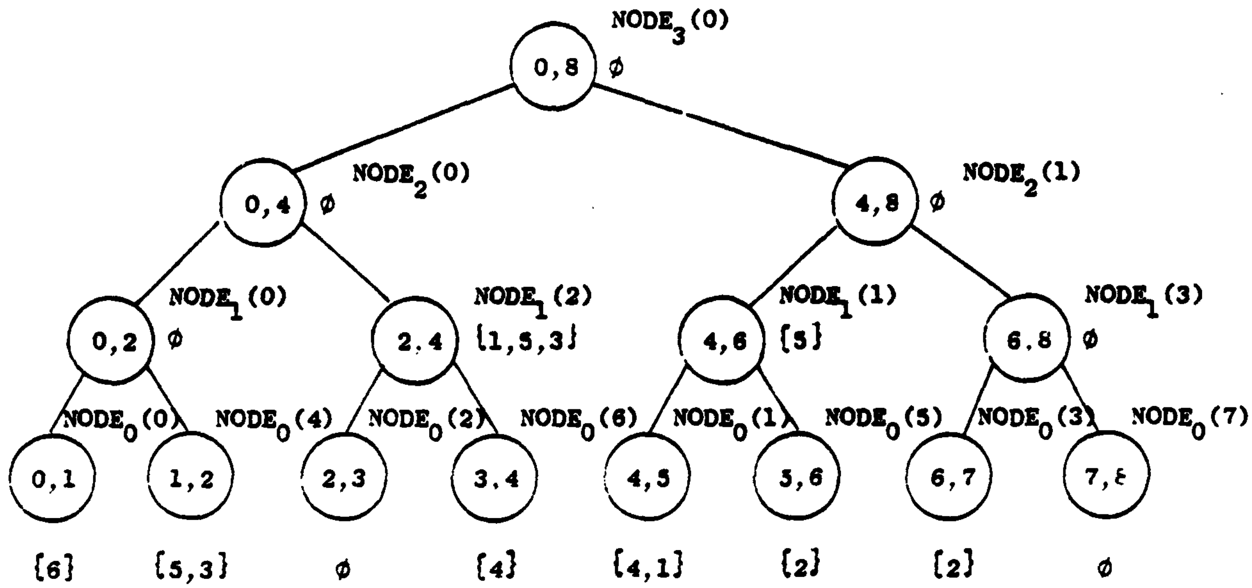
All steps except the last <u>while</u> loop clearly require at most O(log(N+M)) steps. The evaluation of the condition of the <u>while</u> loop and step in this loop require O(log(N+M)) time. But these are only performed at most k/ℓ times, where k is the maximum number of inclusions per query and ℓ is the loop length of the CCC, which is of order log(N+M). Therefore, the time complexity of the while loop is k. Hence, procedure RANGE_SEARCH-1D runs in time O(log(N+M) + k) on a CCC with N+M processors.

<u>Theorem 3.4</u>. Given a sorted array A(0:N-1) and a set Q(0:M-1) of queries sorted by values of the left bounds, all elements of A which lie in the range [Q(i)[L],Q(i)[R]], for i = 0,...,M-1, can be found in time O(log(N+M) + k) on a CCC with N+M processors, where k is the maximum number of inclusions per query.

3.2.2 Intersection of Horizontal and Vertical Line Segments

We revisit the problem of reporting intersecting pairs of horizontal and vertical line segments as introduced in Section 3.1.1. We shall revise procedure INTERSECT1 so that it will be suitable for implementation on a CCC with linear number of processors. Most of the variables used here will have the same meanings as those in Section 3.1.1.

For the set V(0:n-1) of vertical line segments, we construct a search structure $\mathcal{S}$ which consists of $\log N + 1$ arrays $E_0, E_1, \ldots, E_{\log N}$, where N is the number of distinct y-values of the endpoints of the segments in V. Each E_i is a selected subset of vertical line segments in V. The underlying structure of $\mathcal{S}$ is a binary tree similar to $\mathcal{T}$ except for the indexing of the nodes. Instead of indexing the nodes, in some level i, from left to right, a node will be indexed as j if it is the right son of $NODE_{i+1}(j)$ in level i+1 for some j and it will be indexed as $2^{\log N-i-1}+j$ if it is the left son of $NODE_{i+1}(j)$. Therefore, the left and the right sons of $NODE_{i+1}(j)$ are $NODE_i(j)$ and $NODE_i(2^{\log N-i-1}+j)$ respectively. Suppose $NODE_{i+1}(j)$ is the k^{th} leftmost node in level i+1, then $NODE_i(j)$ represents the interval $[B_i(j),T_i(j)] = [Y(2k\cdot 2^i),Y((2k+1)2^i)]$, and $NODE_i(2^{\log N-i-1}+j)$ represents the interval $[B_i(2^{\log N-i-1}+j),T_i(2^{\log N-i-1}+j)] = [Y((2k+1)2^i),Y((2k+2)2^i)]$. The left-to-right sequence of the node indices at any level of $\mathcal{S}$ is the bit-reversal permutation of the node indices at the corresponding level of $\mathcal{T}$, where the bit-reversal permutation maps a binary number $a_{n-1}a_{n-2}\ldots a_0$ into the binary number $a_0a_1\ldots a_{n-1}$. Figure 9(a) is the underlying binary tree of $\mathcal{S}$ for the vertical line segments in Figure 6(a). Note that Figure 9(a) is the same as $\mathcal{T}$ in Figure 6(b) except for the node indices. The array E_i of $\mathcal{S}$ is the concatenation of the lists of vertical line segments associated with the nodes in level i in the order of increasing node indices. We also associate with each element $E_i(j)$ the node number $N\#_i(j)$ such that $E_i(j)[B] \le B_i(N\#_i(j))$ and $E_i(j)[T] \ge T_i(N\#_i(j))$ and $E_i(j)$ does belong to any ancestor of $NODE_i(N\#_i(j))$. Therefore, E_i is a selected list of vertical line segments sorted lexicographically by values of $N\#_i$ and their position in the positive x direction. Figure 9(b) shows the arrays E_3,E_2,E_1,E_0 for the vertical line segments in Figure 6(a) (null elements are denoted by λ).



(a) the underlying binary tree.

E_3	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
$N\#_3$																								
E_2	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
$N\#_2$																								
E_1	5	1	5	3	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
$N\#_1$	1	2	2	2																				
E_0	6	4	1	2	5	3	2	4	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
$N\#_0$	0	1	1	3	4	4	5	6																

(b) the collection of arrays $E_3,\ldots,E_0$

Figure 9. Search structure $\mathcal{S}$ for vertical line segments in Figure 6(a).

Construction of $\mathcal{S}$ is similar to that of $\mathcal{T}$; the arrays E_i are constructed one at a time:

```
procedure CONSTRUCT_𝒮1(V)
    /* construct the search structure 𝒮 i.e. E_logN,...,E_0 for the set
       V(0:N-1) of vertical line segments */
    begin sort V by x-values and then y-values of the bottom endpoints.
        foreach j, 0 ≤ j < n do begin S (j) ← V(j)
                                      π (j) ← 0 end
        foreach j, n ≤ j < 4n do S (j) ← null

        /* determine E_logN,...,E_0 one at a time */
        for i ← logN downto 0 do
            begin

                /* determine E_i by extracting edges from S */
                foreach j, 0 ≤ j < 4n do
                    begin t_1(j) ← t_2(j) ← 0
                        E_i(j) ← S(j); N#_i(j) ← π(j)
                        if S(j) ≠ null
                            then if S(j)[B] ≤ B_i(π(j)) and
                                    T_i(π(j)) ≤ S(j)[T]
                                    then t_1(j) ← 1
                                    else t_2(j) ← 1
                    end
                call EXTRACT2 (E_i t_1); call EXTRACT2(N#_i,t_1)
                call EXTRACT2 (S,t_2); call EXTRACT2(π,t_2)

                /* rearrange the order of elements in S according to
                   their node numbers in the next level */
               foreach j, 0 ≤ j < 4n do
                   begin TEMP(j) ← S(j)
                       t_1(j) ← t_2(j) ← 0
                       if S(j)[B] < T_i-1(π(j)) then t_1(j) ← 1
                       if S(j)[T] > T_i-1(π(j)) then begin
                                                    t_2(j) ←1
                                                    TEMPπ(j) ← 2^(logN-i)+π(j)
                                                   end
```

```
        call EXTRACT2(S,t1); call EXTRACT2(π,y1)
        call EXTRACT2(TEMP,t2); call EXTRACT2(TEMPπ,t2)
        foreach j, 0 ≤ j < |TEMP| do begin S(j+|S|) ← TEMP(j)
                                           π(j+|S|) ← TEMPπ(j)
                                     end
    end
end
```

Analysis of procedure CONSTRUCT_$\mathcal{S}$ is similar to that of CONSTRUCT $\mathcal{T}$. It is easy to show that CONSTRUCT_$\mathcal{S}$ can be implemented on a CCC with 4n processors in $O((\log n)^2)$ steps.

To find intersecting pairs, we use $\mathcal{S}$ as a binary tree. We associate with each horizontal line segment H(i) a node number NN(i) indicating that H(i) may intersect some vertical line in node NN(i). We start at $E_{\log N}$ (the root). It is obvious that NN(i) = 0 for all i (there is only node 0 at this level). The set of horizontal lines is maintained sorted lexicographically by their node numbers and x-values of their left endpoints. Since E_i is sorted in the same manner, we can use the one-dimensional range searching algorithm in Section 3.2.1 to report all intersecting pairs at level i. We then determine which node in the next level should be associated with each horizontal line segment. We continue this process which geometrically traces a unique path, possibly two, from the root to a leaf. Since the depth of $\mathcal{S}$ is logN+1, this process requires $O(\log n.\log(n+m)+k)$ time on a CCC with 4n + 2m processors. We now present formally the intersection algorithm.

```
procedure INTERSECT2(V,H):
    /* search all intersecting pairs of horizontal line segments in H
       and vertical line segments in V */
    begin
        /* construct the search structures E_logN,...,E_0 for V */
        call CONSTRUCT_S1(V)
```

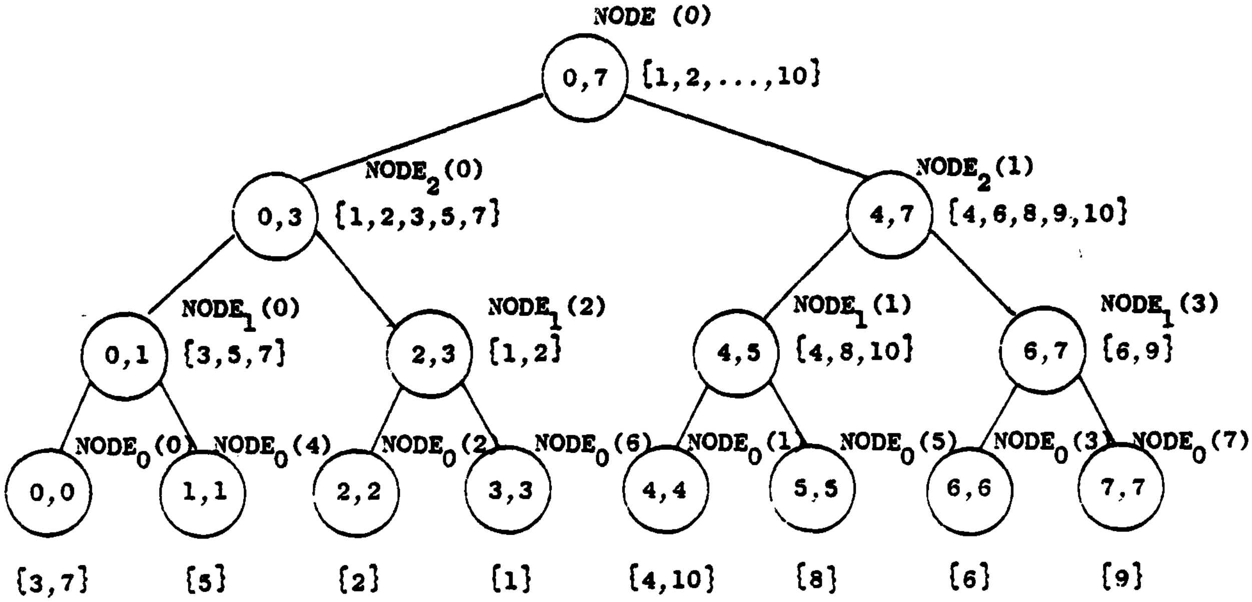
```
/* H', the set of horizontal line segments, is maintained sorted
   lexicographically by their node number and x-values of their
   left endpoints */
sort H by x-values of left endpoints
foreach j, 0 ≤ j < m do begin H'(j) ← H(j)
                              NN(j) ← 0; end
foreach j, m ≤ j < 2m do H'(j) ← null

/* search in 𝒮 beginning at E_logN */
for i ← logN downto 0 do
    begin call RANGE_SEARCH_1D(E_i,H')

        /* determine node numbers for horizontal line segments
           to be used in the next level; then H' is reordered
           according to their node numbers */
        foreach j, 0 ≤ j < 2m do
            begin t_1(j) ← t_2(j) ← 0
                TEMP(j) ← H'(j)
                if H'(j) ≠ null then
                    begin if y-value of H(j) ≤ T_{i-1}(NN(j))
                            then t_1(j) ← 1
                          if y-value of H(j) ≥ T_{i-1}(NN(j))
                            then begin
                                    t_2(j) ← 1
                                    TEMPNN(j)←2^{logN-i}+NN(j)
                                 end
                    end
            end
        call EXTRACT2(H',t_1)
        call EXTRACT2(NN,t_1)
        call EXTRACT2(TEMP,t_2)
        call EXTRACT2(TEMPNN,t_2)
        foreach j, 0 ≤ j < |TEMP| do
            begin H'(|H'|+j) ← TEMP(j)
                  NN(|H'|+j) ← TEMPNN(j)
            end
    end
end
```

Procedure INTERSECT2 gives the following theorem.

<u>Theorem 3.5</u>. All intersecting pairs of n vertical line segments and m horizontal line segments can be reported in time $O((\log(n+m))^2+k)$ on a CCC with $4n+2m$ processors, where k is the maximum number of intersections per vertical line segment.

3.2.3 Two-Dimensional Range Searching

We now investigate the two-dimensional range searching problem stated in Section 3.1.2 on a CCC with linear number of processors. Again, we assume that Y(0:N-1) is a sorted array of distinct y-values of points of S, where $N \leq n$ and N is a power of 2. We construct a search structure $\mathcal{F}$ which consists of logN+1 arrays $F_{logN}, F_{logN-1}, \ldots, F_0$. The underlying structure of $\mathcal{F}$ is a binary tree similar to $\mathcal{K}$ (Section 3.1.2) except for the indexing of the nodes. The nodes in the underlying binary tree of $\mathcal{F}$ are indexed in the same manner as that of $\mathcal{S}$ (Section 3.2.2). Figure 10(a) shows the underlying binary tree of $\mathcal{F}$ for the set of points in Figure 7(a). Note that Figure 10(a) is the same as $\mathcal{K}$ in Figure 7(b) except for the node indices. Suppose that $NODE_{i+1}(j)$ is the k^{th} leftmost node in level i+1, then its right son $NODE_i(j)$ represents the interval $[B_i(j), T_i(j)] = [Y(2k \cdot 2^i), Y(2k+1)2^i)]$ and its left son $NODE_i(2^{logN-i-1}+j)$ represents the interval $[B_i(2^{logN-i-1}+j), T_i(2^{logN-i-1}+j)] = [Y((2k+1)2^i), Y((2k+2)2^i-1)]$. Therefore, F_i is the set S of points sorted lexicographically by their node numbers and x-values. At level i, the node number of $F_i(k)$ is $NN_i(k)$, where the y-value of $F_i(k)$ is in the range $[B_i(NN_i(k)), T_i(NN_i(k))]$. Figure 10(b) shows the contents of F_i and NN_i for the example in Figure 7(a). The construction of $\mathcal{F}$ is similar to $\mathcal{S}$: The set S of points is first sorted by their x-values. The resulting array is F_{logN}. We then determine the node numbers NN_{logN-1} for each point and rearrange the order of points in the array according to their node numbers. Since the cardinality of F_i is n for all i, $\mathcal{F}$ can be constructed in time $O((\log n)^2)$ on a CCC with n processors. The program CONSTRUCT_$\mathcal{F}$ for constructing $\mathcal{F}$ is presented in the Appendix.



(a) the underlying binary tree.

F_3	1	2	3	4	5	6	7	8	9	10
NN_3	0	0	0	0	0	0	0	0	0	0
F_2	1	2	3	5	7	4	6	8	9	10
NN_2	0	0	0	0	0	1	1	1	1	1
F_1	3	5	7	4	8	10	1	2	6	9
NN_1	0	0	0	1	1	1	2	2	3	3
F_0	3	7	4	10	2	6	5	8	1	9
NN_0	0	0	1	1	2	3	4	5	6	7

(b) the collection of arrays $F_3,\ldots,F_0$.

Figure 10. Search structure $\mathcal{F}$ for the set of points in Figure 7(a).

To answer the set Q of queries, we search in $\mathcal{F}$ for each k until we reach level i such that $Q(k)[B] \le B_i(j)$ and $T_i(j) \le Q(k)[T]$ for some j. Then we perform a one-dimensional range search to report all the inclusions. Since we may visit at most four nodes on one level for a particular query, 4m+n processors are sufficient. We use the result in Section 3.2.1 for one-dimensional range searching, so we have the following result.

Theorem 3.6. The two-dimensional range searching problem for n data and m queries can be solved in time $O((\log(n+m))^2+k)$ on a CCC with n+4m processors, where k is the maximum number of inclusions per query.

```
procedure RANGE_SEARCH2(S,Q)
    /* report all points a ∈ S such that Q(i)[L] ≤ x(a) ≤ Q(i)[R]
       and Q(i)[B] ≤ y(a) ≤ Q(i)[T] for every Q(i) */
    begin
        /* construct the search arrays 𝓕:F_logN,...,F_0 for the set S */
        call CONSTRUCT_𝓕(S)

        /* Q' is the set Q sorted by the values of left bounds */
        Q' ← Q
        sort Q' by Q'(i)[L]
        foreach j, 0 ≤ j < m do NN(j) ← 0
        foreach j, m ≤ j < 4m do Q'(j) ← null

        /* search in F_logN,...,F_0 one at a time */
        for i ← logN downto 0 do
        begin
            /* determine Q" which is a subset of queries that can be
               answered at this level. For the remaining queries,
               determine their node numbers in the next level */
            foreach j, 0 ≤ j < 4m do
                begin t_1(j) ← t_2(j) ← t_3(j) ← 0
                    Q"(j) ← TEMP(j) ← Q'(j)
                    NN"(j) ← NN(j);TEMPNN(j) ← NN(j) + 2^(logN-i)
                    if Q'(j)[B] ≤ B_i(NN(j)) and T_i(NN(j) ≤ Q'(j)[T])
```

```
                    then t_1(j) ← 1
                    else begin
                              if Q'(j)[B] ≤ T_{i-1}(NN(j)) then t_2(j) ← 1
                              if Q'(j)[T] ≤ B_{i-1}(NN(j) + 2^{logN-i})
                                   then t_3(j) ← 1
                         end
          end
     call EXTRACT2(Q",t_1); call EXTRACT2(NN",t_1)

     /* answer queries in Q" by performing a one-dimensional
        range searching */
     call RANGE_SEARCH_1D(F_i,Q")

     /* extract Q'-Q" from Q' and rearrange the order according
        to their node numbers */
     call EXTRACT2(Q',t_2); call EXTRACT2(NN,t_2)
     call EXTRACT2(TEMP,t_3); call EXTRACT2(TEMPNN,t_3)
     foreach j, 0 ≤ j < |TEMP| do
          begin Q'(j + |Q'|) ← TEMP(j)
               NN(j + |Q'| ← TEMPNN(j)
          end
   end
end
```

3.2.4 The Rectangle Intersection Algorithm

The rectangle intersection algorithm for a CCC is the same as that for a SMM but uses different algorithms for finding the intersections of horizontal and vertical line segments and for two-dimensional range searching.

```
procedure RECTINT2 (REC):
     begin
          V ← all vertical edges of rectangles in REC
          H ← all horizontal edges of rectangles in REC
          call INTERSECT2(V,H)
          S ← all left bottom endpoints of rectangles in REC
          Q ← REC
          call RANGE_SEARCH2(S,Q)
     end
```

Theorem 3.7. Given N rectangles with edges parallel to the coordinate axes, all intersecting pairs of these rectangles can be reported in time $O(\log N^2 + k)$ on a CCC with N processors, where k is the maximum number of intersections per rectangle.

Proof: Combining results in Sections 3.2.2 and 3.2.3, we use some simple processor-time tradeoffs similar to the one used in the previous section to achieve the time complexity of $O((\log N)^2 + k)$ and processor complexity of N. □

3.3 On the CCC with $N^{1+\alpha}$ Processors

In this section we shall develop an algorithm for reporting intersecting pairs of N rectangles for a CCC with superlinear number of processors. This algorithm can be implemented in $O(\frac{1}{\alpha} \log N + k)$ time requiring $N^{1+\alpha}$ processors, where $0 < \alpha \leq 1$ and k is the maximum number of intersections per rectangle.

3.3.1 Intersection of Horizontal and Vertical Line Segments

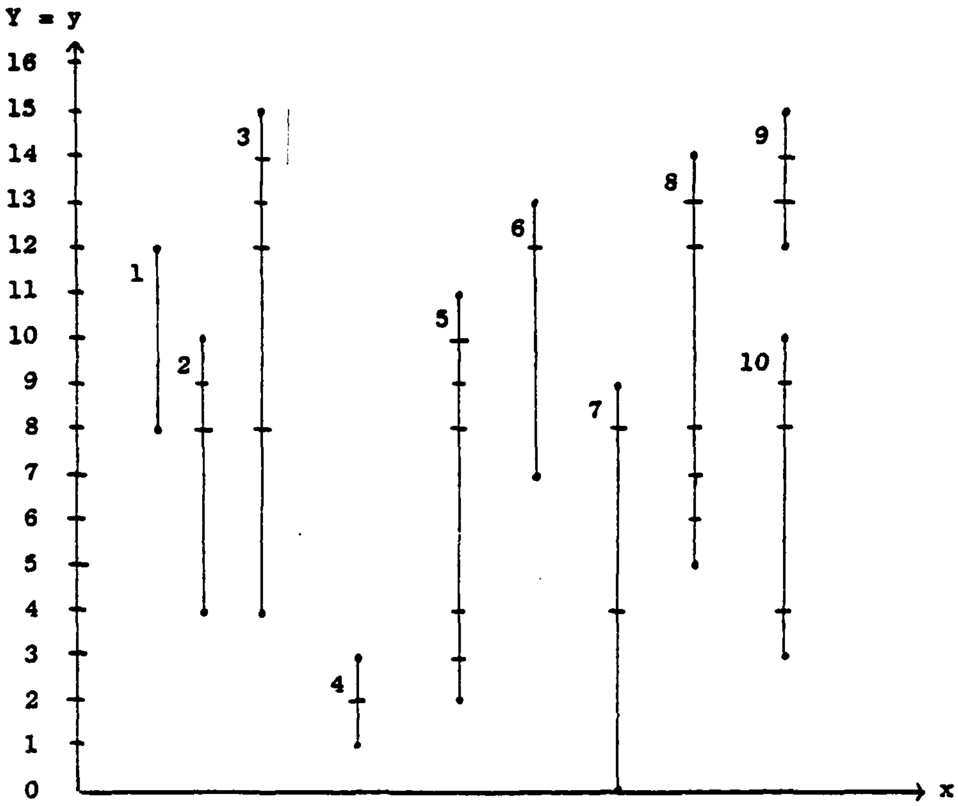
As in the algorithms developed for a CCC with N processors, we construct a search structure $\mathcal{B}$ for the set V(0: n-1) of vertical line segments so that the intersections of horizontal line segments in H(0: m-1) and V(0: n-1) can be found efficiently. Let N be the number of distinct y-values of the endpoints of V. $\mathcal{B}$ consists of $\frac{1}{\alpha}+1$ arrays $D_{1/\alpha}, D_{1/\alpha-1}, \ldots, D_0$. Each D_i is a selected subset of V sorted lexicographically by their node number (as defined in Section 3.2.2) and their positions in the positive x direction. The underlying geometric structure of $\mathcal{B}$ is a N^{α}-ary tree of height $\frac{1}{\alpha}$: there are $N^{1-i\alpha}$ nodes at height i, indexed as follows. At level $\frac{1}{\alpha}$, the root is indexed 0. Node j which is the k^{th} leftmost node at level i has N^{α} sons at level i-1; they are nodes j, $N^{1-i\alpha}+j$, $2N^{1-i\alpha}+j, \ldots,$ $(N^{\alpha}-1)N^{1-i\alpha}+j$ representing respectively the intervals

$[Y(kN^{\alpha}N^{i\alpha-\alpha}),Y((kN^{\alpha}+1)N^{i\alpha-\alpha})],[Y(kN^{\alpha}+1)N^{i\alpha-\alpha}),Y((kN^{\alpha}+2)N^{i\alpha-\alpha})],$

$[Y((kN^{\alpha}+2)N^{i\alpha-\alpha}),Y(kN^{\alpha}+3)N^{i\alpha-\alpha})],\ldots,[Y((kN^{\alpha}+N^{\alpha}-1)N^{i\alpha-\alpha}),Y(kN^{\alpha}+N^{\alpha})N^{i\alpha-\alpha})].$

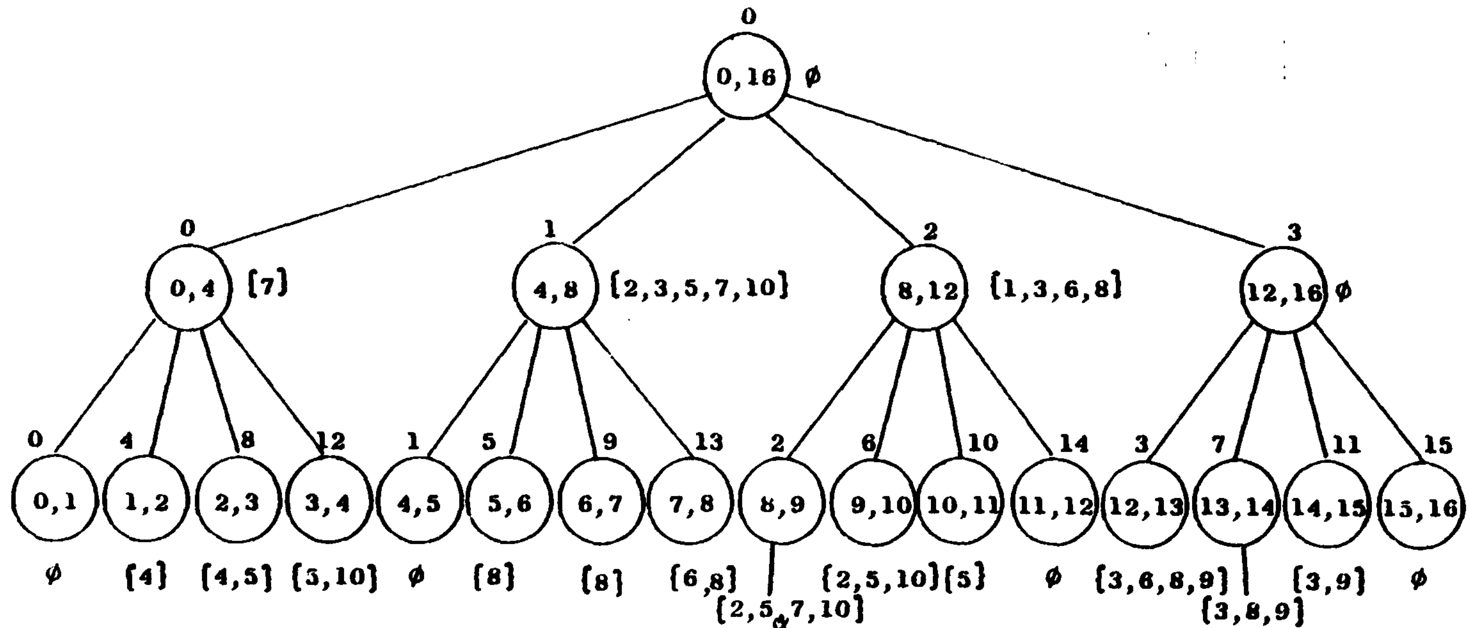
Figure 11 shows an example with $N = 16$, $\alpha = \frac{1}{2}$. Figure 11(b) is the underlying N^{α}-ary tree; pairs of integers in the circles are values of $B_i(j)$ and $T_i(j)$, and the integers above the circles are node numbers.

The construction of arrays $D_{1/\alpha},\ldots,D_0$ runs as follows. Initially, the node number of each vertical line segment is 0. Let S be the set V of vertical line segments sorted lexicographically by their node numbers, x-values, and y-values of bottom endpoints. We extract from S all the segments which cover the range $[Y(0),Y(N)]$ and form the set $D_{1/\alpha}$. After the extraction, the remaining elements of S are duplicated $N^{\alpha}-1$ times. Then we determine to which of the N^{α} subtrees we should branch for each vertical line segment, that is, we determine the node numbers for the remaining elements of S in the next level as follows. We branch to the leftmost subtree if the y-value of one or both endpoints of the vertical line segment is in the range $[B_{1/\alpha-1}(0),T_{1/\alpha-1}(0)]$; branch to the second leftmost subtree if it is in range $[B_{1/\alpha-1}(1),T_{1/\alpha-1}(1)]$; and so on. We then repeat the process until all arrays of $\mathcal{D}$ are determined. Let us analyze the time and number of processors required. At each iteration i, a vertical line segment may appear at most $2N^{\alpha}$ times in S. After the extraction of D_i, S contains at most 2n elements. Then the elements of S are replicated into N^{α} copies. Therefore, at any time, the maximum number of elements in S is $2nN^{\alpha} < 4n^{1+\alpha}$. Since data extraction and replication can be done in time



(a) a set of vertical line segments (the "cuts" on the edges show the segmentation for $\mathcal{B}$).

Figure 11. Search structure $\mathcal{B}$ for a set of vertical line segments.



(b) the underlying N-ary tree, α =1/2.

D_2	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	...
node#																												

D_1	7	2	3	5	7	10	1	3	6	8	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	...
node#	0	1	1	1	1	1	2	2	2	2																		

D_0	2	5	7	10	3	6	8	9	4	8	2	5	10	3	8	9	4	5	8	5	3	9	5	10	6	8	λ	...
node#	2	2	2	2	3	3	3	3	4	5	6	6	6	7	7	7	8	8	9	10	11	11	12	12	13	13		

(c) search arrays D_2, D_1, D_0.

Figure 11.

O(logn) on a CCC with a number of processors linear in the problem size and $\mathcal{D}$ contains $\frac{1}{\alpha}+1$ arrays, $\mathcal{D}$ can be determined in time $O(\frac{1}{\alpha}\log n)$ with $4n^{1+\alpha}$ processors. We now present formally the construction algorithm which we just described.

```
procedure CONSTRUCT_𝒟1(V)

     /* construct the arrays D_{1/α},D_{1/α-1},...,D_0 for the set V of vertical

        line segments */
     begin

          /* maintain S as an array of vertical line segments sorted
             lexicographically by their node numbers and their
             x-coordinates */
          sort V by x-values and then y-values of bottom endpoints
          foreach j, 0 ≤ j < n do begin S(j) ← V(j); π(j) ← 0 end
          foreach j, n ≤ j < 2nN^α do S(j) ← null

          /* D_{1/α},...,D_0 are constructed one by one in descending order */
          for i ← 1/α downto 0 do
               begin

                    /* for each vertical line segment of S, determine if
                       it belongs to some node at this level; extract
                       those which do and assign them to D_i */

                    foreach j, 0 ≤ j < 2nN^α do
                         begin t_1(j) ← t_2(j) ← 0; D_i(j) ← S(j);N#_i(j) ← π(j)
                              if S(j) ≠ null
                                   then if S(j)[B] ≤ B_i(π(j)) and
                                                            T_i(π(j)) ≤ S(j)[T]
                                        then t_1(j) ← 1
                                        else t_2(j) ← 1
                         end
                    call EXTRACT2(D_i,t_1); call EXTRACT2(N#_i,t_1)

                    /* for the remaining of S, determine their node numbers
                       for the next level; and reorder them according to
                       their node numbers */
                    call EXTRACT2(S,t_2); call EXTRACT2(π,t_2)
                    for k ← log2n to log2N^α-1 do /* duplicate N^α times */
                         for j, 0 ≤ j < 2nN^α do
```

```
                    if BIT_k(j) = 0 then begin S(j + 2^k) ← S(j)
                                                π(j + 2^k) ← π(j)
                                         end
            foreach j, 0 ≤ j < 2nN^α do /* determine node numbers */
                begin π(j) ← π(j) + ⌊j/2n⌋ N^(1-iα)
                      t(j) ← 0
                      if S(j) ≠ null and (S(j)[B] < T_{i-1}(π(j)) or
                                         S(j)[T] > B_{i-1}(π(j)))
                          then t(j) ← 1
                end
            call EXTRACT2(S,t); call EXTRACT2(π,t) /* reordering */
        end
    end
```

Searching in $\mathcal{B}$ for all intersecting pairs of horizontal and vertical line segments is the same as searching in $\mathcal{S}$ except we have to choose one, possibly two, out of N^{α} branches at one level of $\mathcal{B}$ for each horizontal line. The procedure INTERSECT3 to be presented in the Appendix can be implemented on a CCC with $4n^{1+\alpha} + 2mN^{\alpha}$ processors in $(\frac{1}{\alpha} \log(n+m) + k)$ parallel steps, where k is the maximum number of intersections per vertical line segment. We state this result in the following theorem.

Theorem 3.8. All intersecting pairs of n vertical line segments and m horizontal line segments can be reported in time $O(\frac{1}{\alpha} \log(n+m) + k)$ on a CCC with $4(n+m)n^{\alpha}$ processors, $0 < \alpha \le 1$, where k is the maximum number of intersections per vertical line segment.

3.3.2 Two-Dimensional Range Searching

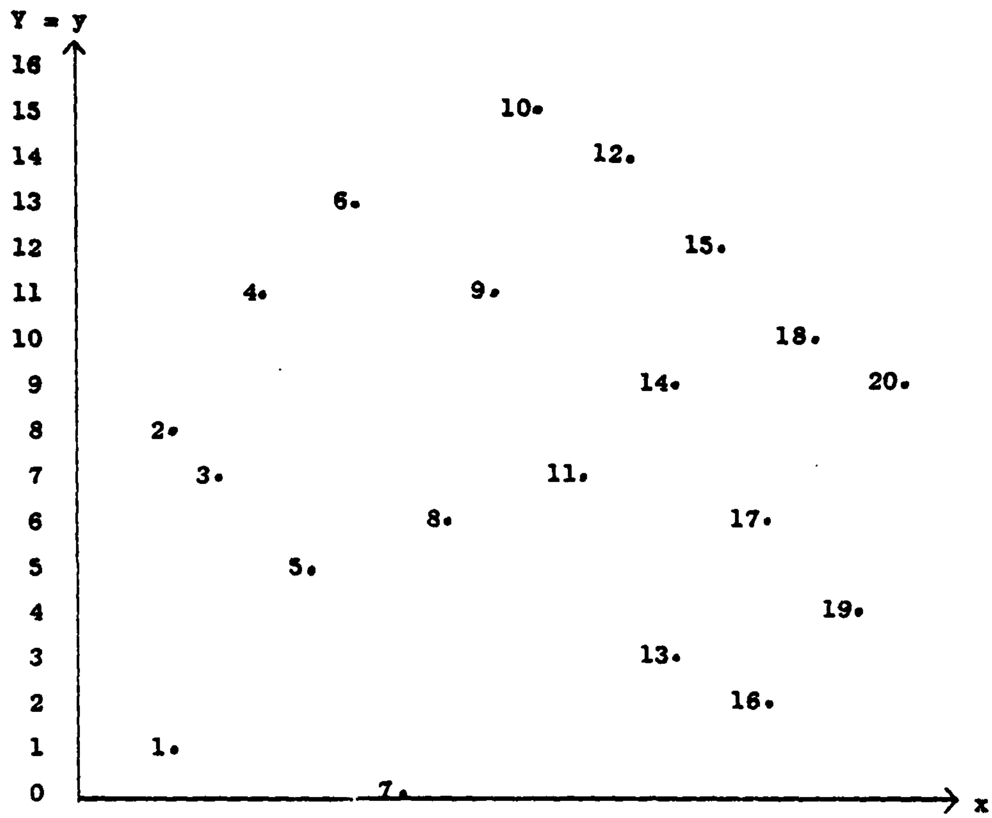
For the two-dimensional range searching problem, we arrange the set S of points into the data structure $\mathcal{J}$ (similar to $\mathcal{B}$), so that the set Q of queries can be answered efficiently. In $\mathcal{J}$, $G_{1/\alpha}, \ldots, G_0$ are arrays of points in S. The points in array G_i are ordered by their node numbers at level i and x-values. The node number, at level i, of a point is j if its y-value is in the range $[B_i(j), T_i(j)]$. Node j which is the k^{th} (for some k) leftmost node in level i has N^α sons at level i-1; they are nodes j, $N^{1-i\alpha}+j, \ldots, (N^\alpha-1)N^{1-i\alpha}+j$ representing respectively the intervals $[B_{i-1}(j), T_{i-1}(j)] = [Y(kN^\alpha N^{i\alpha-\alpha}), Y((kN^\alpha+1)N^{i\alpha-\alpha}-1)]$, $[B_{i-1}(N^{1-i\alpha}+j), T_{i-1}(N^{1-i\alpha}+j)] = [Y((kN^\alpha+1)N^{i\alpha-\alpha}), Y((kN^\alpha+2)N^{i\alpha-\alpha}-1)], \ldots,$ $[B_{i-1}((N^\alpha-1)N^{1-i\alpha}+j), T_{i-1}((N^\alpha-1)N^{1-i\alpha}+j)] = [Y((kN^\alpha+N^\alpha-1)N^{i\alpha-\alpha}),$ $Y((kN^\alpha+N^\alpha)N^{i\alpha-\alpha}-1)]$.

Figure 12 is an example of a set of 20 points and the corresponding data structure $\mathcal{J}$, with N = 16 and $\alpha = \frac{1}{2}$. Figure 12(b) is the underlying N^α-ary tree; the pairs of integers in the circles are values of $B_i(j)$ and $T_i(j)$, and the integer above the circles are node numbers.

The construction of $\mathcal{J}$ is similar to that of $\mathcal{B}$. Since the cardinality of G_i is n for all i, $\mathcal{J}$ can be constructed in time $O(\frac{1}{\alpha}\log n)$ on a CCC with nN^α processors. The program CONSTRUCT_$\mathcal{J}$ for constructing $\mathcal{J}$ will be presented in the Appendix.

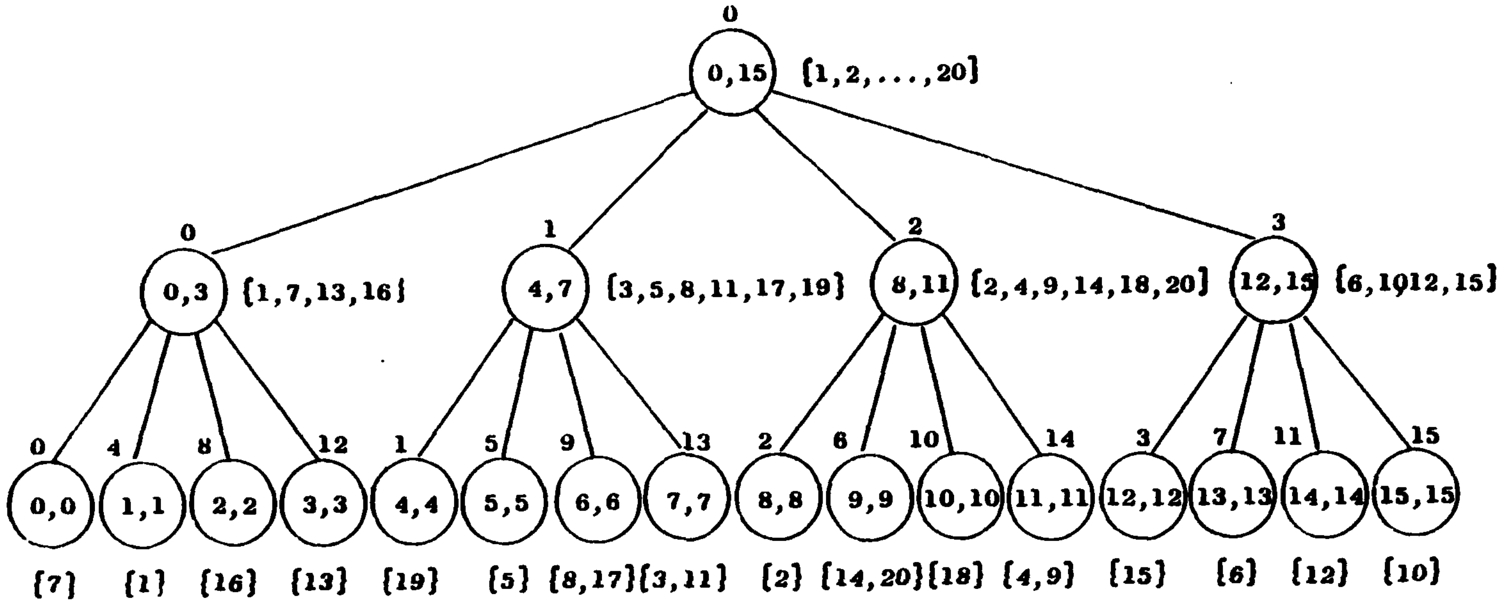
Given a set Q of m queries, we search in $\mathcal{J}$ until we reach a node j such that $Q(k)[B] \leq B_i(j)$ and $T_i(j) \leq Q(k)[T]$. Then we perform a one-dimensional range searching on G_i. We may have to search at most $2N^\alpha$ nodes at one particular level for a particular query. Therefore, we may need at most $2N^\alpha m + nN^\alpha$ processors. The analysis of time complexity of this range searching is straightforward.

Theorem 3.9. The two-dimensional range searching problem for n data and m queries can be solved in time $O(\frac{1}{\alpha}\log(n+m)+k)$ on a CCC with $2(n+4m)n^\alpha$ processors where $0 < \alpha \leq 1$ and k is the maximum number of inclusions per query.



(a) A set of 20 points with 16 distinct y-values.

Figure 12. Search structure $\mathcal{J}$ for a set of points.



(b) the underlying N^{α}-ary tree, $\alpha=1/2$.

G_2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
node no.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

G_1	1	7	13	16	3	5	8	11	17	19	2	4	9	14	18	20	6	10	12	15
node no.	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3

G_0	7	19	2	15	1	5	14	20	6	16	8	17	18	12	13	3	11	4	9	10
node no.	0	1	2	3	4	5	6	6	7	8	9	9	10	11	12	13	13	14	14	15

(c) search structure $\mathcal{S}$

Figure 12.

The program RANGE_SEARCH3 will be presented in the Appendix.

3.3.3 <u>The Rectangle Intersection Algorithm</u>

The rectangle intersection algorithm for a CCC with superlinear number of processors uses results in Sections 3.3.1 and 3.3.2. The running time is $O(\frac{1}{\alpha}\log N+k)$ and the number of processors is $10N^{1+\alpha}$.

```
procedure RECTINT3(REC):
    begin
        V ← all vertical edges of rectangles in REC
        H ← all horizontal edges of rectangles in REC
        call INTERSECT3(V,H)
        S ← all leftmost bottom points of REC
        Q ← REC
        call RANGE_SEARCH3(S,Q)
    end
```

We can use some processor-time tradeoffs similar to the one used in Section 3.1.3 to obtain the following results.

<u>Theorem 3.10</u>. Given N rectangles with edges parallel to the coordinate axes, all intersecting pairs of these rectangles can be reported in time $O(\frac{1}{\alpha}\log N+k)$ on a CCC with $N^{1+\alpha}$ processors, $0 < \alpha \le 1$, where k is the maximum number of intersections per rectangle.

CHAPTER 4

PLANAR POINT LOCATION

The problem of planar point location is stated as follows: given a planar graph embedded in the plane as a straight line graph [21] G with N vertices and a point P, find the region of the planar subdivision induced by G which contains P. This problem is quite important in computational geometry. We shall show in later sections how it can be applied to solve other problems. A recent and practical result for serial computation on this problem is due to Preparata [28]. His algorithm runs in O(logN) time on a data structure which can be constructed in O(NlogN) time.

Many times, point locations are performed repeatedly on the same graph; therefore, it is beneficial to arrange the given graph into an organized structure to facilitate searching. Furthermore, very often, these searches are independent and can be performed simultaneously. In this chapter we preprocess the given graph G = (V,E) so that we can locate M points simultaneously on the SMM and on the CCC. V(0: N-1) is the set of vertices and E(0:|E|-1) is an array of records containing information about each edge: its two endpoints and the regions lying on either side of it (left and right). We shall assume that Y(0: N-1) is the sorted array of distinct y-values of V and N is a power of 2. Figure 13 shows a planar straight-line graph with 20 vertices and 16 distinct y-values, i.e., N = 16.

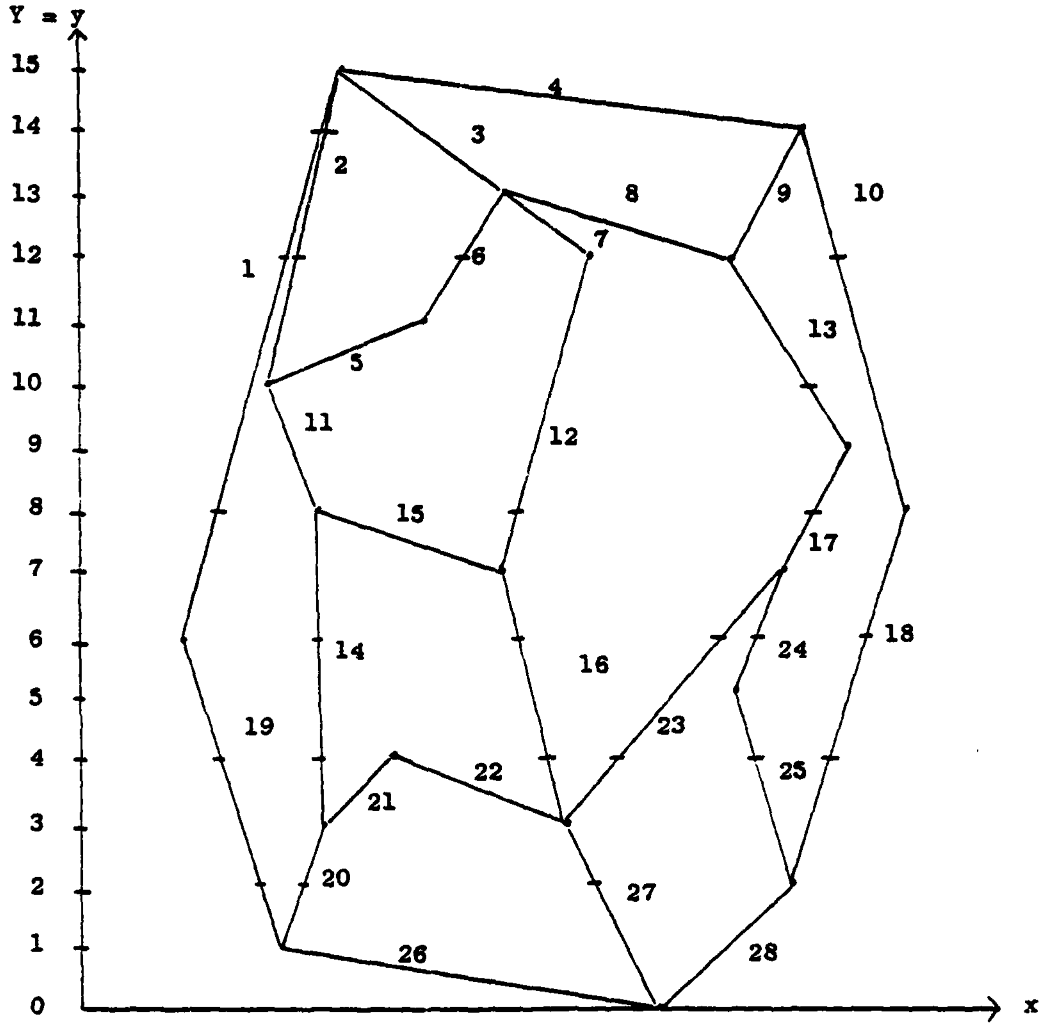


Figure 13. A planar straight line graph. (the "cuts" on the edges show the segmentation for $\mathcal{T}$ and $\mathcal{S}$)

4.1 On the SMM with max(N,M) Processors

In this section we describe two algorithms: (i) the construction of a search structure for the set of edges on the SMM with N processors and (ii) the concurrent location of M points with M processors. The construction and the location run in time $O((\log N)^2 \log\log N)$ and $O((\log N)^2)$ respectively.

4.1.1 Definition and Construction of the Point Location Tree

Recall the search tree $\mathcal{T}$ introduced in Section 3.1.1. We can produce $\mathcal{T}$, for the set of edges of the given graph, G, which will be referred to as the point location tree for G. Figure 14 gives the point location tree for the graph in Figure 13. Recall that the initial step of the procedure CONSTRUCT_$\mathcal{T}$1 developed in Section 3.1.1 is to obtain an ordering of the set $E(0{:}|E|-1)$ of edges such that if $E(i)$ is the left of $E(j)$ then $E(i)$ procedes $E(j)$ in the ordering. Unfortunately, there is no known efficient parallel algorithm for topological sorting. Therefore, we cannot use the same procedure CONSTRUCT_$\mathcal{T}$1 to produce the point location tree $\mathcal{T}$ for the edges. Since the list associated with node $NODE_i(j)$ consists of edges which span the same y-interval $[B_i(j),T_i(j)]$, these edges are comparable, that is, every edge is either to the left or to the right of another edge in the same list. We can sort the edges in the lists associated with each node after the members of the lists have been determined. Since each node contains at most $|E|$ edges ($|E| < 3N$) and each edge is contained in at most two nodes at any one level, we can sort the edges in every node at

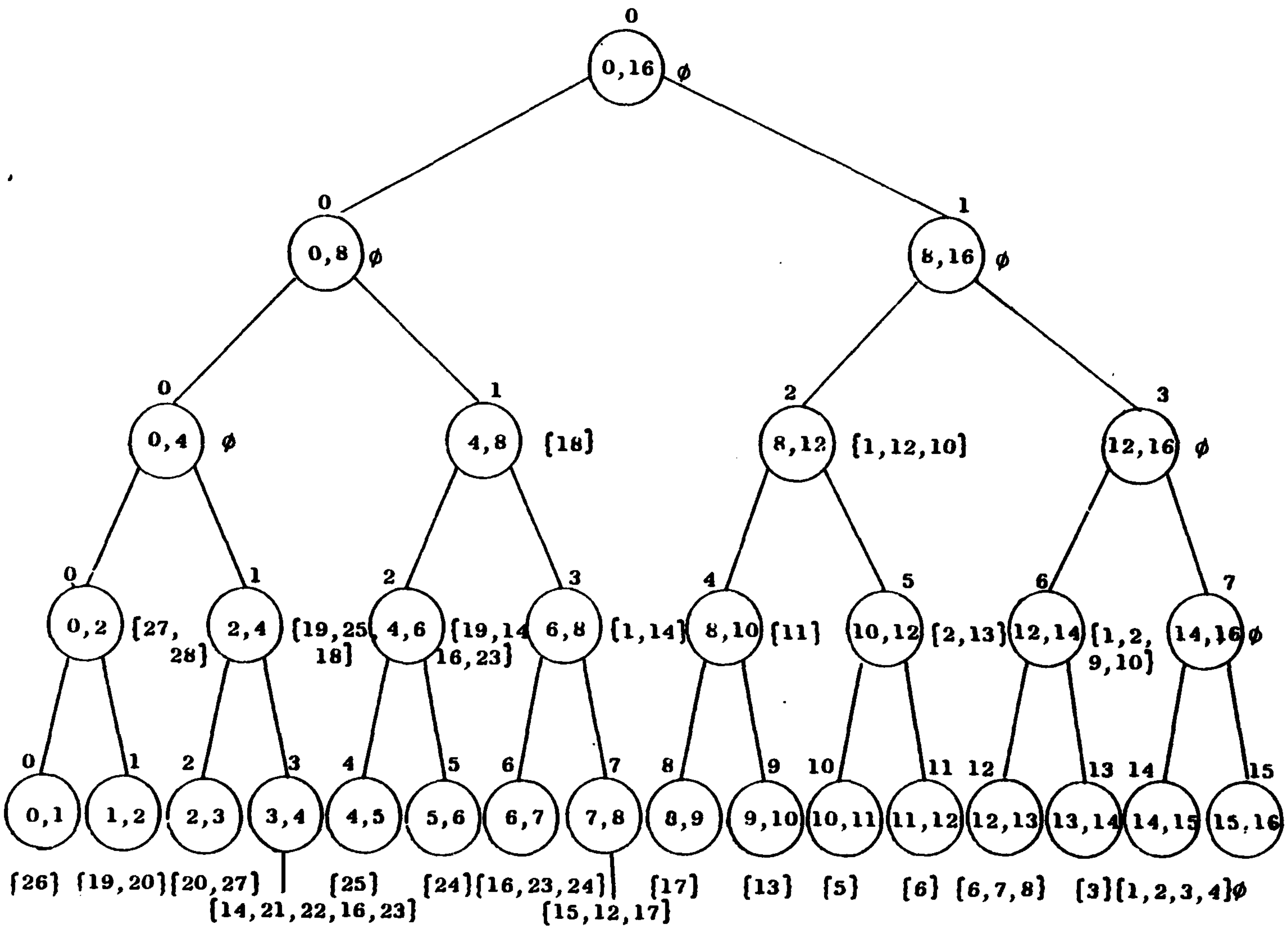


Figure 14. Point location tree for the graph in Figure 13.

level in time O(logNloglogN) using N processors. Again we construct $\mathcal{T}$ level by level beginning from the root. The procedure CONSTRUCT_$\mathcal{T}$2, which will be presented in the appendix, for the set of edges is the same as CONSTRUCT_$\mathcal{T}$1 for the set of vertical line segments except we do not initially order the edges in the entire set.

4.1.2 Point Location

To locate a point P(k) in the planar subdivision induced by G, we use $\mathcal{T}$ as a binary search tree. We define two "dummy" vertical edges $\bar{E}_{-\infty}$ and $\bar{E}_{\infty}$ of infinite length which are at negative and positive infinity respectively. Associated with P(k), we determine a pair of edges L(k) and R(k) of E which bound P(k) on the left and on the right respectively. Initially, we set L(k) and R(k) to $\bar{E}_{-\infty}$ and $\bar{E}_{\infty}$, respectively. We search $\mathcal{T}$ until L(k) and R(k) bound the same region: at a selected node $NODE_i(j)$ of $\mathcal{T}$ where the edges form an ordered set we perform a binary search, for an edge immediately to the left (right) of P(k), compare this edge with L(k) (R(k)); the one closer to P(k) is the new value of L(k) (R(k)). If L(k) and R(k) bound the same region, P(k) is in this region: otherwise, we have to choose a branch or both by comparing the y-value of P(k) with $T_{i-1}(j)$: if it is less than, greater than or equal to $T_{i-1}(2j)$ then we branch respectively to the left, the right or both branches (refer to Figure 15). Note that the y-value of P(k) may be equal to only one $T_{i-1}(2j)$. Thus, we trace a unique path, possibly two (when the y-value of P(k) is equal to some $T_{i-1}(2j)$), from the root to (at most) the bottom level of $\mathcal{T}$. Since $\mathcal{T}$ is of height logN + 1 and the edges in each node are sorted, this

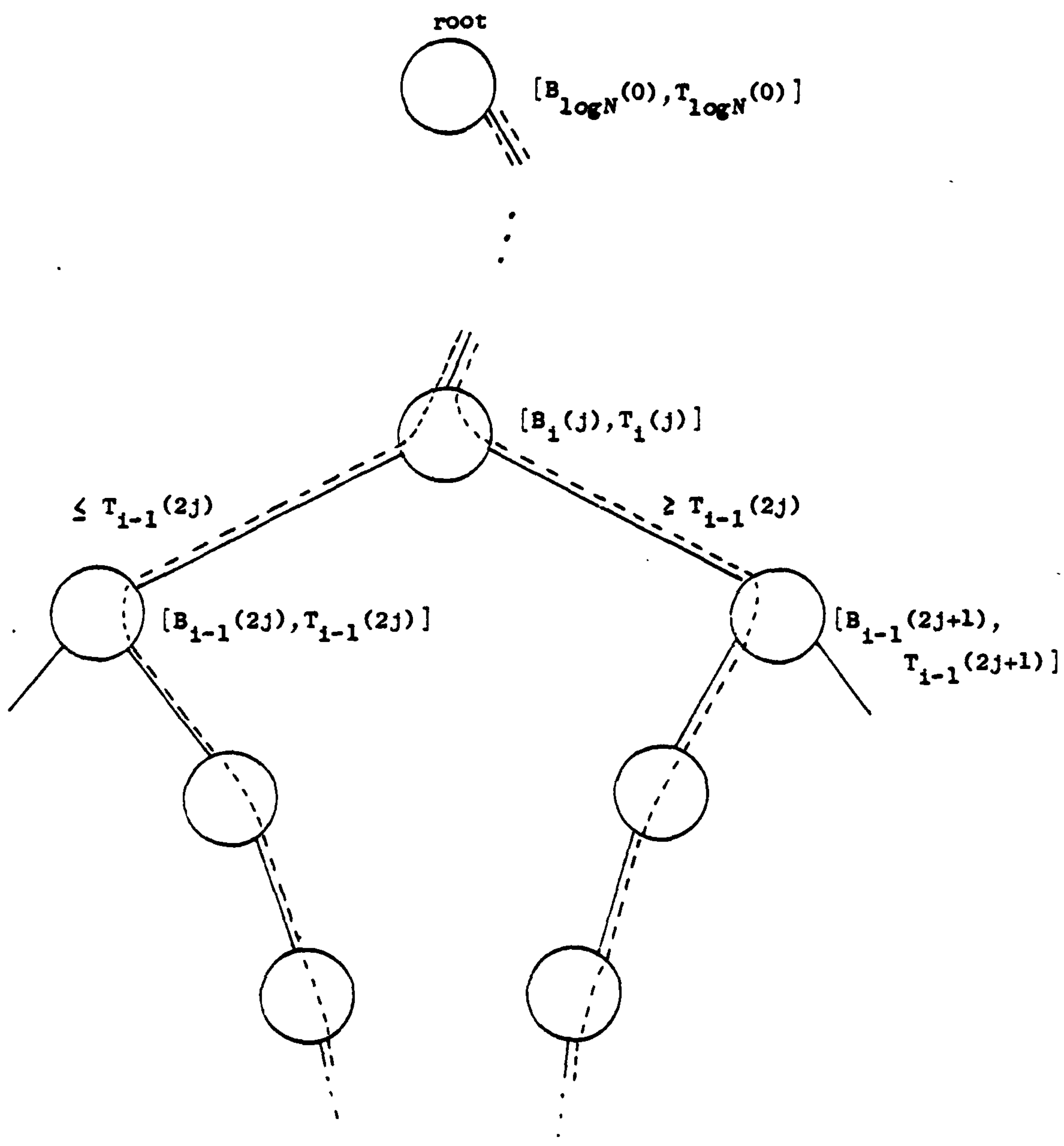


Figure 15. Searching in the point location tree.

process runs in time $O((\log N)^2)$. We can locate all M points simultaneously, provided we search in one level of $\mathcal{T}$ for all points before going to the next level. The number of processors required is M for parallel searching. We shall present the formal description LOCATE1 in the appendix.

We conclude this section by the following theorem.

Theorem 4.1. Given a planar straight line graph with N vertices, we can locate M points in the planar subdivision induced by the graph in time $O((\log N)^2)$ with $O((\log N)^2 \log\log N)$ preprocessing time on a SMM with max(N,M) processors and memory units.

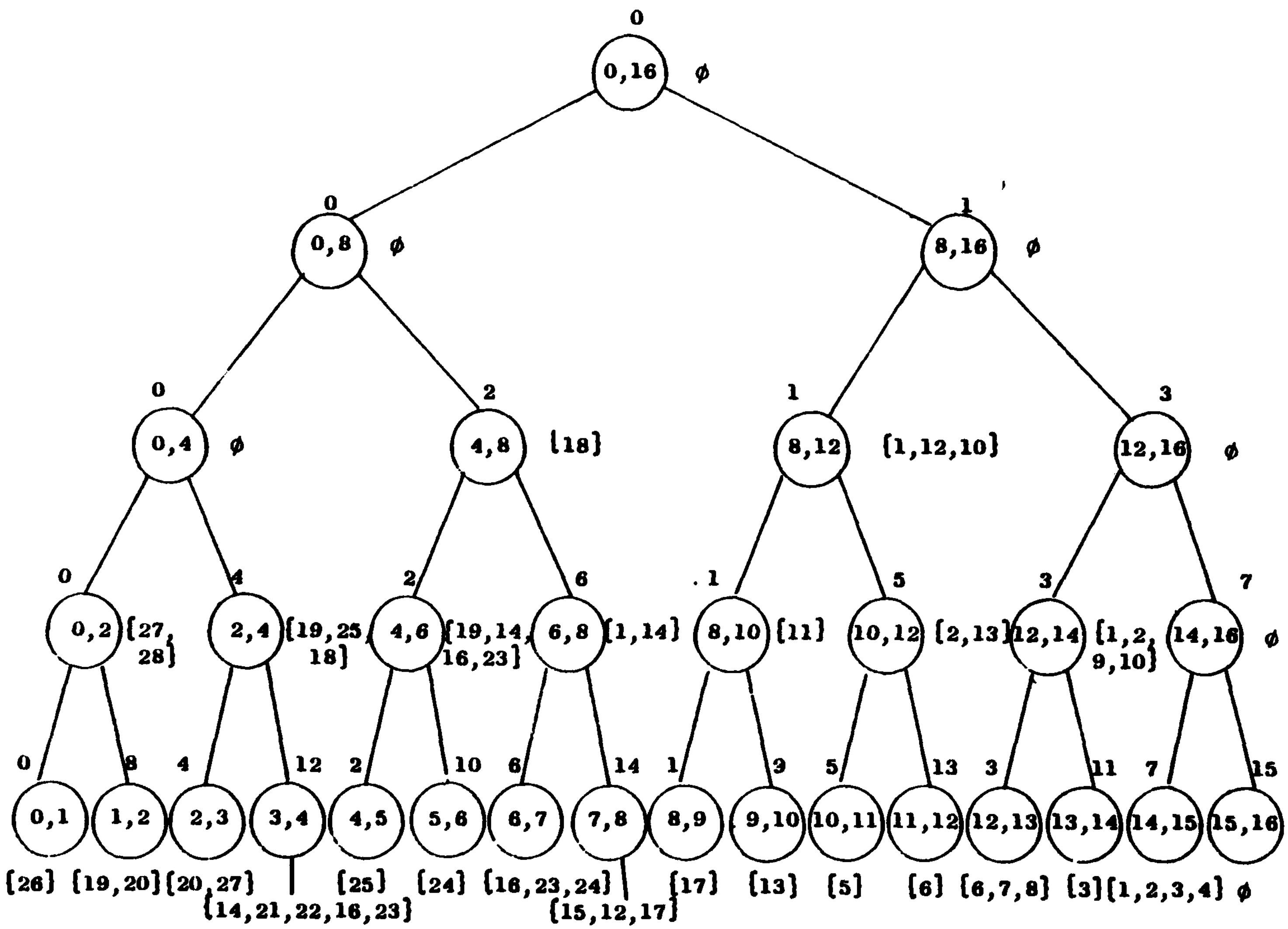
4.2 On the CCC with N+M Processors

In this section we revisit the problem of planar point location as discussed in Section 4.1. We shall revise procedure LOCATE1 so that it will be suitable for implementation on a CCC with linear number of processors.

4.2.1 Construction of the Search Structure

In Section 3.2.2, we construct a search structure $\mathcal{S}$ (a set of arrays $E_0, E_1, \ldots, E_{\log N}$) for a set of vertical line segments. We can produce the same structure $\mathcal{S}$ for the set of edges. Figure 16(a) is the underlying binary tree of $\mathcal{S}$ for the graph in Figure 13. Note that this tree is the same as the point location tree in Figure 14 except for the node indices. Figure 16(b) shows the collection of arrays $E_4, \ldots, E_0$ and the corresponding node number of edges.

As discussed in Section 4.1.1, it is relatively time-consuming to obtain initially a total ordering of the edges. Thus, we first determine the edges in E_i then sort them lexicographically by their node numbers and their positions in the positive x direction. We can develop a



(a) the underlying binary tree of $\mathcal{S}$

Figure 16. Search Structure $\mathcal{S}$ for the graph in Figure 13.

E_4	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
node #																														
E_3	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
node #																														
E_2	1	12	10	18	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
node #	1	1	1	2																										
E_1	27	28	11	19	14	16	23	1	2	9	10	19	25	18	2	13	1	14	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
node #	0	0	1	2	2	2	2	3	3	3	3	4	4	4	5	5	6	6												
E_0	26	17	25	6	7	8	20	27	5	16	23	24	1	2	3	4	19	20	13	24	3	14	21	22	16	23	6	15	12	17
node #	0	1	2	3	3	3	4	4	5	6	6	6	7	7	7	7	8	8	9	10	11	12	12	12	12	12	13	14	14	14

(b) the set of arrays $E_4,\ldots,E_0$

Figure 16.

procedure CONSTRUCT_δ2 for producing δ for the set of edges which will be the same as procedure CONSTRUCT_δ1 in Section 3.2.2 for a set of vertical line segments except in CONSTRUCT_δ2 we do not initially order the entire set of edges, but order the edges in each E_i separately. Since the cardinality of each E_i is at most $2|E|$ ($|E| < 3N$), we can easily verify that the procedure CONSTRUCT_δ2 in the appendix runs in time $O((\log N)^3)$ on a CCC with N processors.

4.2.2 Point Location

As a preliminary step, we sort the set P(0: M-1) points to be located by their x-coordinates. Like point location on a SMM in Section 4.1.2, for each point P(k), we search in δ until the two edges L(k) and R(k) bound the same region. We associate with each point P(k) a node number NN(k) indicating that the y-coordinate of P(k) is in the range $[B_i(NN(k)), T_i(NN(k))]$ at some level i. We start at $E_{\log N}$ (the root). It is obvious that NN(k) is equal to 0 for all k at the root. The set of points is maintained sorted lexicographically by their node numbers NN(k) and their x-coordinates. Since E_i is sorted in the same manner, we can use the parallel searching algorithm in Section 2.2.3 to determine the pairs of edges L(k) and R(k). If L(k) and R(k) do not bound the same region, we have to determine which node in the next level of δ we should continue to search. This process pictorially traces, in the underlying binary search tree of δ, a unique path, possibly two, for each point, from the root to the bottom level. Since the parallel searching at each level requires $O(\log(N+M))$ time and $\mathcal{S}$ has $\log N+1$ levels, the point location described above runs in time $O(\log(N+M)\log N)$ on a CCC with $N+M$ processors. We present the formal point ·ation procedure LOCATE2 in the appendix.

Procedure LOCATE2 gives us the following theorem.

Theorem 4.2. Given a planar straight-line graph with N vertices, we can locate M points in the planar subdivision induced by the graph in time $O((\log(N+M))^2)$ with $O((\log N)^3)$ preprocessing time on a CCC with $N+M$ processors.

4.3 On the CCC with $(N+M)^{1+\alpha}$ Processors

In this section we investigate the problem of point location on a CCC with $(N+M)^{1+\alpha}$ processors, where N is the number of vertices of a given graph, M is the number of points to be located, and $0 < \alpha \leq 1$.

4.3.1 Definition and Construction of the Search Structure

Recall the search structure $\mathcal{D}$ we constructed for a set of vertical line segments in the algorithm for reporting intersection of vertical and horizontal line segments (Section 3.3.1). The underlying geometric structure of $\mathcal{D}$ is a N^{α}-ary tree of height $\frac{1}{\alpha}$ (refer to Figure 18). Figure 17 shows the same planar straight line graph as in Figure 13 but with different edge segmentation. We can produce the same structure $\mathcal{D}$ for the set of edges. $\mathcal{D}$ will consist of $\frac{1}{\alpha}+1$ arrays $D_{1/\alpha},\ldots,D_0$, each of which is a selected subset of edges sorted lexicographically by their node numbers and their positions in the positive x direction. Here again, for well known reasons, we first determine the edges in D_i and then

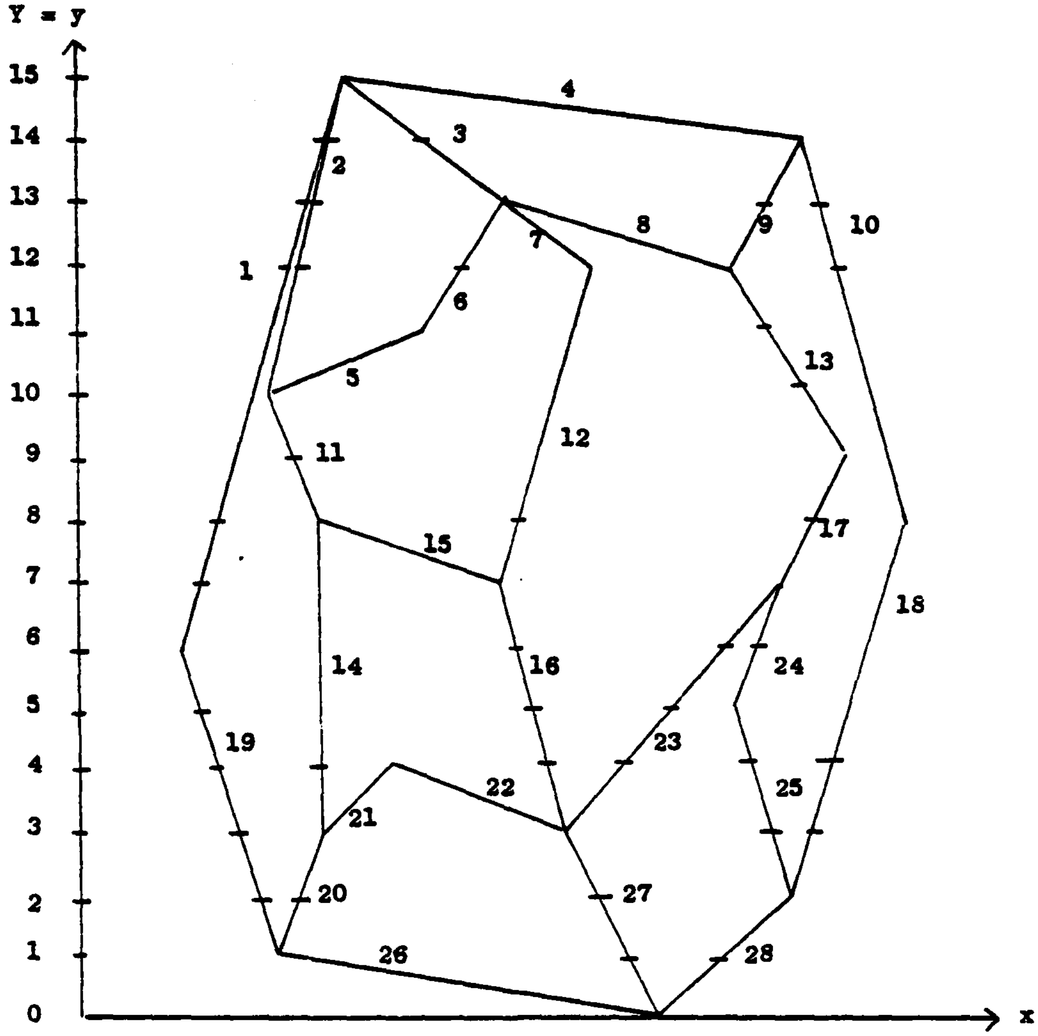


Figure 17. A planar straight line graph. (the "cuts" on the edges show the segmentation for $\mathcal{P}$)

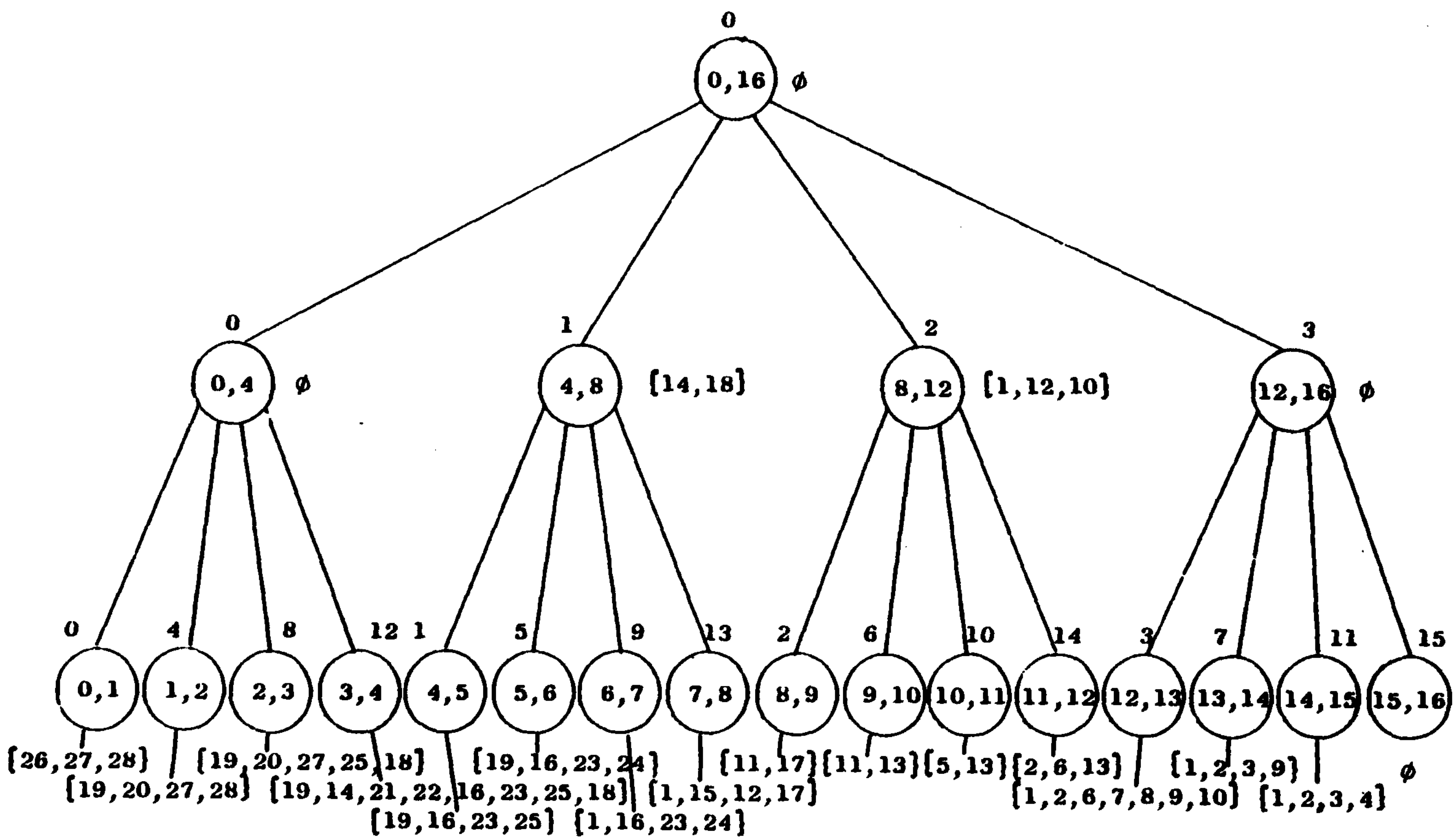


Figure 18. The underlying N^{α}-ary tree, with $\alpha=1/2$, for the search structure $\mathcal{A}$ for the graph in Figure 17.

sort them. By the same argument as in Section 3.3.1, each D_i contains at most $2N^{1+\alpha}$ edges. Therefore $\mathcal{B}$ can be constructed in time $O(\frac{1}{\alpha}(\log N)^2)$ on a CCC with $N^{1+\alpha}$ processors. The procedure CONSTRUCT_$\mathcal{B}$2 which will be presented in the appendix for the set of edges is similar to the procedure CONSTRUCT_$\mathcal{B}$1 for a set of vertical line segments with the following difference. In procedure CONSTRUCT_$\mathcal{B}$2, we do not initially order the entire set of edges but we determine the members of each D_i before we order them.

4.3.2 Point Location

Point location $\mathcal{B}$ is the same as point location in $\mathcal{B}$ except we have to choose one, possibly two, out of N^{α} branches at any level of $\mathcal{B}$ for each point. The procedure LOCATE3 to be presented in the appendix, can be implemented on a CCC with $(N+M)^{1+\alpha}$ processors in $O(\frac{1}{\alpha}(\log(N+M))^2)$ parallel steps. We state this in the following theorem.

Theorem 4.3. Given a planar straight line graph G with N vertices, we can locate M points in the planar subdivision induced by G in time $O(\frac{1}{\alpha}\log(N+M))$ with $O(\frac{1}{\alpha}(\log(N+M))^2)$ processing time on a CCC with $(N+M)^{1+\alpha}$ processors.

CHAPTER 5

CONVEX HULLS OF SETS OF POINTS IN TWO DIMENSIONS

Formally, the convex hull of a finite set S of points is the intersection of all convex sets containing S. In the plane, the convex hull of S, CH(S), is a convex polygon. Specifying a polygon unambiguously requires giving its vertices in the order that they occur on the boundary.

A simple polygon is in <u>standard form</u> if its vertices occur in clockwise order with all vertices distinct and no three consecutive vertices collinear, beginning with the vertex that has largest y-coordinate.

The problem of convex hulls arises in many applications: finding diameter of a set, determining the existence of a linear classifier of a set, etc. Several optimal algorithms for determining sequentially the convex hull of a set of N points in two dimensions have been developed [2,9,30,35]. These algorithms use the well-known technique called "divide and conquer" [1] and achieve the running time of O(NlogN). In a parallel machine, the subproblems generated by the "divide and conquer" method can be solved simultaneously, so an efficient algorithm for combining the results of these subproblems is essential for an overall fast parallel algorithm. We shall develop some preliminaries before designing convex hulls algorithms on the SMM and on the CCC.

5.1 Preliminaries

Given a convex polygon A(0: n-1) in standard form, let ℓ_A, s_A and r_A be the indices[(1)] of the vertices with least x coordinate, least y coordinates and largest x coordinate respectively. Given two points p

[(1)]Indices of polygon A(0:n-1) are modulo n.

and q in the plane, $\theta(p,q)$ denotes the polar angle of q with p as the origin. We define $\alpha_{i,j} = \theta(A(i),A(j))^{(1)}$. Due to convexity, in the range $0 \le i < n-1$, the sequence $(\alpha_{01},\ldots,\alpha_{i,i+1},\ldots)$ is decreasing.

Let A(0: n-1) and B(0: m-1) be two convex polygons where the y-coordinate of A(i) is less than that of B(j), for $0 \le i < n$ and $0 \le j < m$, so A and B are non-intersecting. We define $\gamma_{i,j} = \theta(A(i),B(j))^{(2)}$. A sequence is <u>$\vee$-bitonic</u> if it consists of a decreasing sequence, which may be empty, followed by an increasing sequence. A sequence is <u>$\wedge$-bitonic</u> if it consists of an increasing sequence, which may be empty, followed by a decreasing sequence. Due to convexity, in the range $0 \le i \le s_A$ the sequence $(\gamma_{i,0},\gamma_{i,1},\ldots,\gamma_{i,s_B})$ is $\vee$-bitonic and in the range $s_A \le i < n$ the sequence $(\gamma_{i,s_B},\gamma_{i,s_B+1},\ldots,\gamma_{i,m})$ is $\wedge$-bitonic (refer to Figure 19). We define $j^{(i)}$ as $\min\{j \mid \gamma_{ij} \le \gamma_{ik},\ 0 \le k \le r_B\}$ for i, $0 \le i \le r_A$ and as $\min\{j \mid \gamma_{ij} \le \gamma_{ik}, r_B \le k \le s_B\}$ for i, $r_A \le i \le s_A$. We also define $\bar{j}^{(i)}$ as $\max\{j \mid \gamma_{ij} \ge \gamma_{ik}, s_B \le k \le \ell_B\}$ for i, $s_A \le i \le \ell_A$ and as $\max\{j \mid \gamma_{ij} \ge \gamma_{ik}, \ell_B \le k \le m\}$ for i, $\ell_A \le i \le n$. We shall explore some characteristics of $j^{(i)}$ and $\bar{j}^{(i)}$.

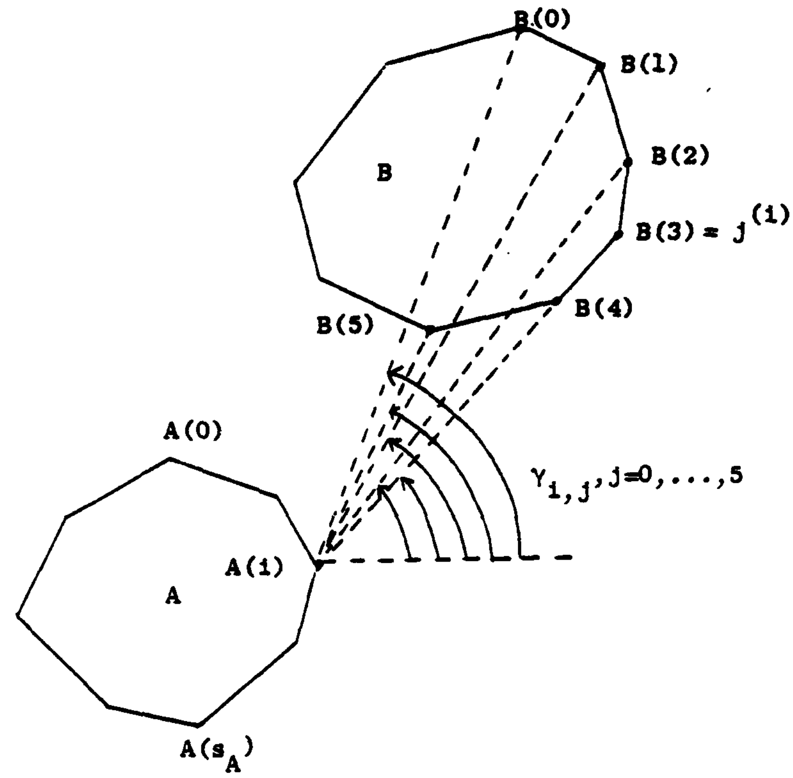
<u>Lemma 5.1</u>. $\alpha_{i+1,i} < \gamma_{i,j}(i) \Rightarrow j^{(i)} \le j^{(i+1)}$, $0 \le i \le s_A$

<u>Proof</u>: The condition $\alpha_{i+1,i} < \gamma_{i,j}(i)$ implies A(i+1) is in the hatched region (refer to Figure 20). Suppose $j^{(i+1)} < j^{(i)}$; this implies that $B(j^{(i+1)})$ is in the crosshatched region. Then it yields the contradiction $\gamma_{i+1,j}(i+1) > \gamma_{i+1,j}(i)$ on the definition of $j^{(i+1)}$. □

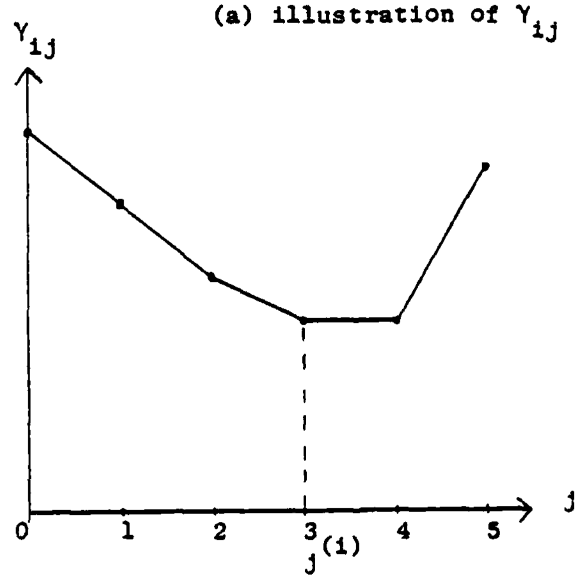
---

(1) $\alpha_{i,j}$ is defined as polar angle for explanatory purpose only; in the implementation of the operation of comparing two angles, we shall avoid computation of angles by replacing it with the operation of comparing the negative values of their cotangents, where the function contangent: $[0,\pi] \to [-\infty,\infty]$ is an order-reversing mapping.

(2) Same as (1).

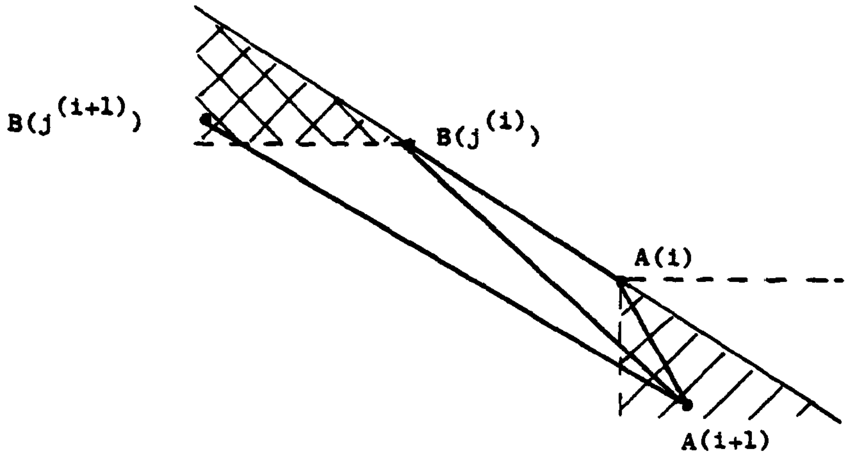


(a) illustration of γ_{ij}

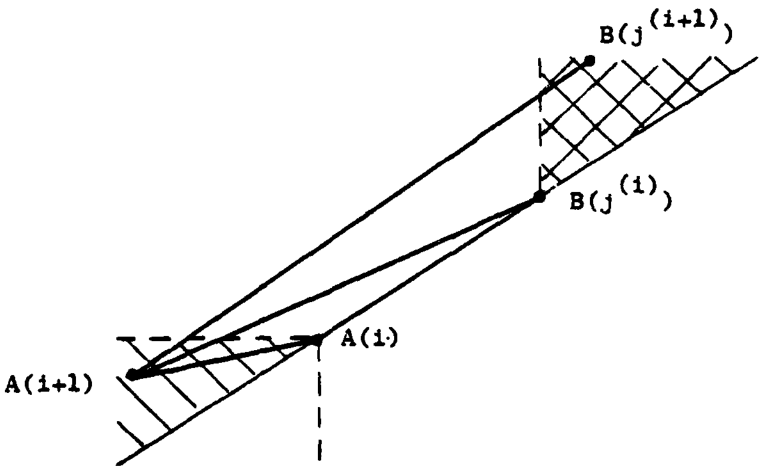


(b) γ_{ij} vs. j

Figure 19. Illustration of the V-bitonic sequence $(\gamma_{i0},\ldots,\gamma_{i,s_B})$ and $j^{(i)}$.



(a) $0 \leq i \leq r_A$



(b) $r_A \leq i \leq s_A$

Figure 20. Illustration of the proof of Lemma 5.1.

Lemma 5.2. $j^{(i)} < j^{(i+1)} \Rightarrow \alpha_{i+1,i} < \gamma_{i,j}(i),\ 0 \le i \le s_A$

Proof: $j^{(i)} < j^{(i+1)}$ means $B(j^{(i+1)})$ is in the hatched region in Figure 21. Suppose $\alpha_{i+1,i} \ge \gamma_{i,j}(i)$ which implies $A(i+1)$ is in the crosshatched region. We then have $\gamma_{i+1,j}(i+1) > \gamma_{i+1,j}(i)$ which contradicts the definition of $j^{(i+1)}$. □

By similar arguments we have the following lemmas on $\bar{j}(i)$.

Lemma 5.3. $\alpha_{i,i+1} < \gamma_i, \bar{j}^{(i)} \Rightarrow \bar{j}^{(i)} \ge \bar{j}^{(i+1)},\ s_A \le i \le n,$

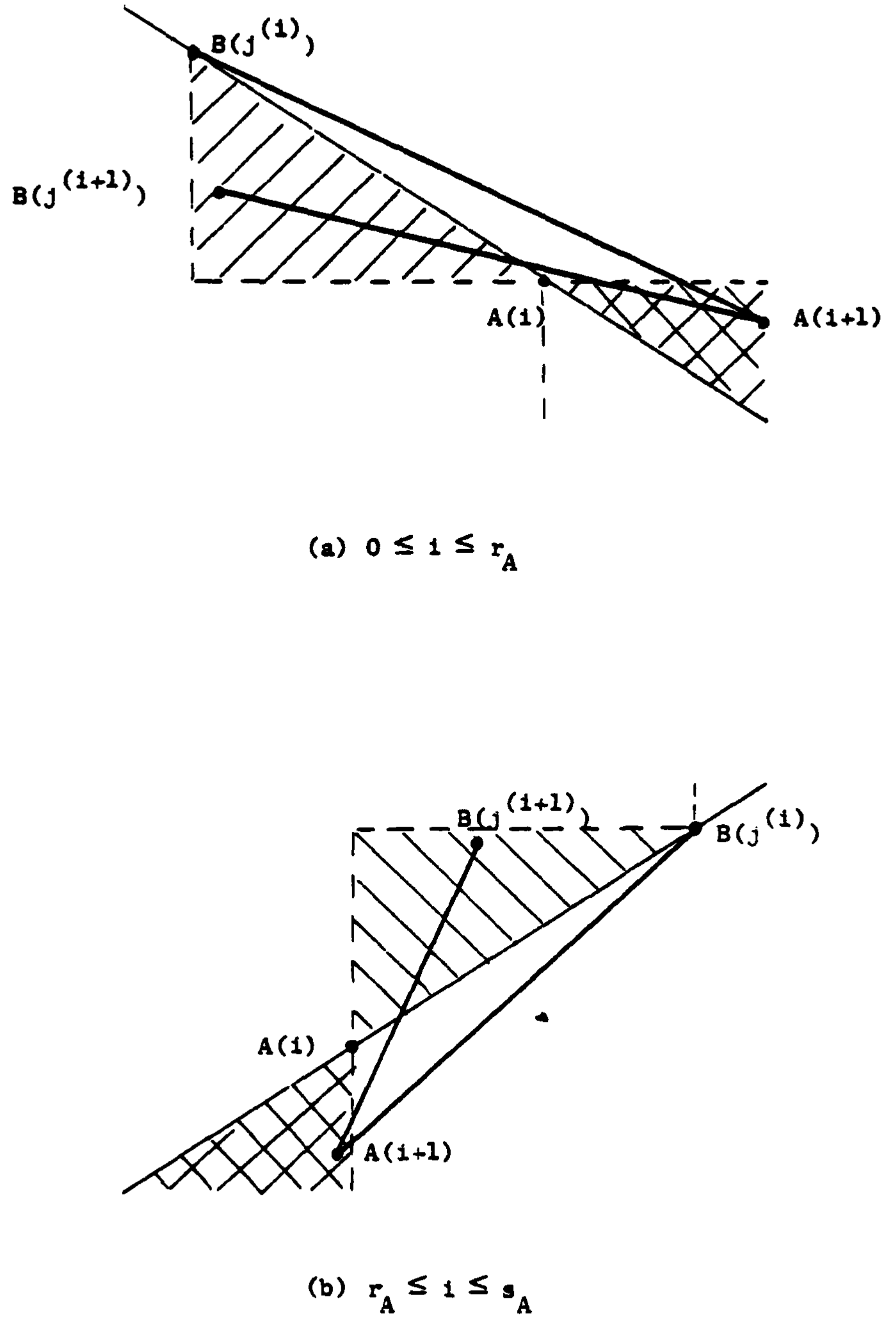
Lemma 5.4. $\bar{j}^{(i)} > \bar{j}^{(i+1)} \Rightarrow \alpha_{i,i+1} < \gamma_i \bar{j}^{(i)},\ \ s_A \le i \le n$.

We are going to use these lemmas to show an important property of the sequence of $j^{(i)}$ $(\bar{j}^{(i)})$.

Theorem 5.1. In the range $0 \le i \le r_A$, if $j^{(i-1)} < j^{(i)}$ for some i then $j^{(i)} \le j^{(i+1)} \le \ldots \le j^{(r_A)}$. And in the range $r_A \le i \le s_A$, if $j^{(i-1)} < j^{(i)}$ then $j^{(i)} \le j^{(i+1)} \le \ldots \le j^{(s_A)}$.

Proof: We shall show that if $j^{(i-1)} < j^{(i)}$ then $\alpha_{k+1,k} < \gamma_k, j^{(k)}$ for

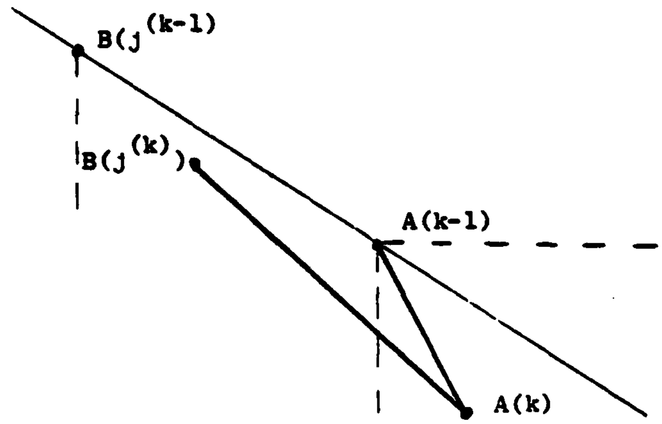
$k = i-1, i, \ldots, h$, where h is r_A for $0 \le i \le r_A$ and s_A for $r_A \le i \le s_A$. We prove by induction on k. The basis $\alpha_{i,i-1} < \gamma_{i-1}, j^{(i-1)}$ is true by Lemma 5.2. In the inductive step, we assume that $\alpha_{k,k-1} < \gamma_{k-1}, j^{(k-1)}$. Then by Lemma 5.1, $j^{(k-1)} \le j^{(k)}$. Referring to Figure 22, we have $\alpha_{k,k-1} < \gamma_k, j^{(k)}$. Due to convexity, $\alpha_{k+1,k} < \alpha_{k,k-1}$. Therefore, we have $\alpha_{k,k+1} < \gamma_k, j^{(k)}$. Hence, the statement in Lemma 5.1 completes the proof. □



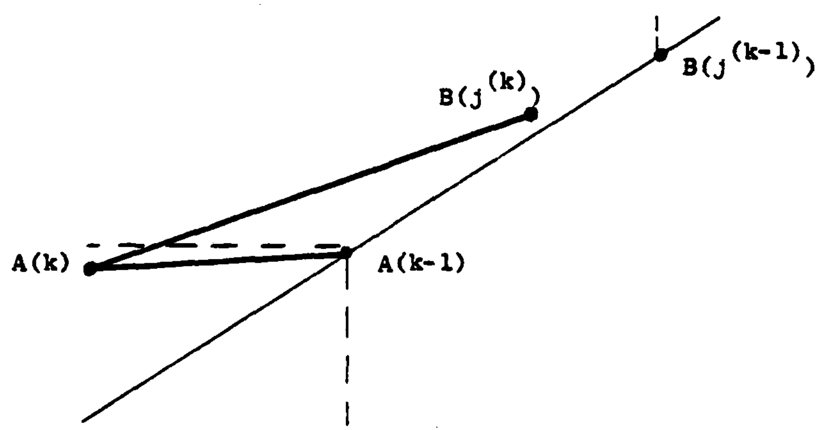
(a) $0 \leq i \leq r_A$

(b) $r_A \leq i \leq s_A$

Figure 21. Illustration of the proof of Lemma 5.2.



(a) $0 \leq k \leq r_A$



(b) $r_A \leq k \leq s_A$

Figure 22. Illustration of portion of the proof of Theorem 5.1.

Using an argument similar to the one above, we can establish the following theorem.

Theorem 5.2. In the range $s_A \le i \le \ell_A$, if $\bar{j}^{(i-1)} > \bar{j}^{(i)}$ for some i then $\bar{j}^{(i)} \ge \bar{j}^{(i+1)} \ge \bar{j}^{(\ell_A)}$. And in the range $\ell_A \le i \le n$, if $\bar{j}^{(i-1)} > \bar{j}^{(i)}$ then $\bar{j}^{(i)} \ge \bar{j}^{(i+1)} \ge \ldots \ge j^{(n)}$.

These two theorems can be interpreted as follows:

Corollary 5.1. $(j^{(0)},\ldots,j^{(r_A)})$ is a nonincreasing sequence followed by a nondecreasing sequence: so is $(j^{(r_A)},\ldots,j^{(s_A)})$. $(\bar{j}^{(s_A)},\ldots,\bar{j}^{(\ell_A)})$ is a nondecreasing sequence followed by a nonincreasing sequence; so is $(\bar{j}^{(\ell_A)},\ldots,\bar{j}^{(n)})$.

5.2 Merging Two Convex Hulls

Given two convex polygons A(0: n-1) and B(0: m-1), where the y-value of A(i) is smaller than that of B(j) for $0 \le i < n$ and $0 \le j < m$, by merging of A and B we mean the determination of the convex polygon $C(0{:}\ j^*-i^*+\bar{i}^*-\bar{j}^*+m-1)$ which is obtained by tracing the two lines of support $(A(\bar{i}^*),B(\bar{j}^*))$ and $(A(i^*),B(j^*))$ common to A and B, to be referred to as left and right tangents respectively, and by eliminating the vertices of A and B which becomes internal to the resulting polygon (refer to Figure 23).

It is observed that if $B(r_B)$ is to the left of $A(r_A)$, then i^* and j^* are in the ranges $[0,r_A]$ and $[0,r_B]$, respectively; otherwise, i^* and j^* are in the ranges $[r_A,s_A]$ and $[r_B,s_B]$ respectively. It is also observed that if $B(\ell_B)$ is to the left of $A(\ell_B)$, then $\bar{i}^*$ and $\bar{j}^*$ are in the intervals $[s_A,\ell_A]$ and $[s_B,\ell_B]$ respectively, otherwise, $\bar{i}^*$ and $\bar{j}^*$ are in the intervals $[\ell_A,n]$ and $[\ell_B,m]$, respectively. Furthermore, the tangents $(A(i^*),B(j^*))$ and

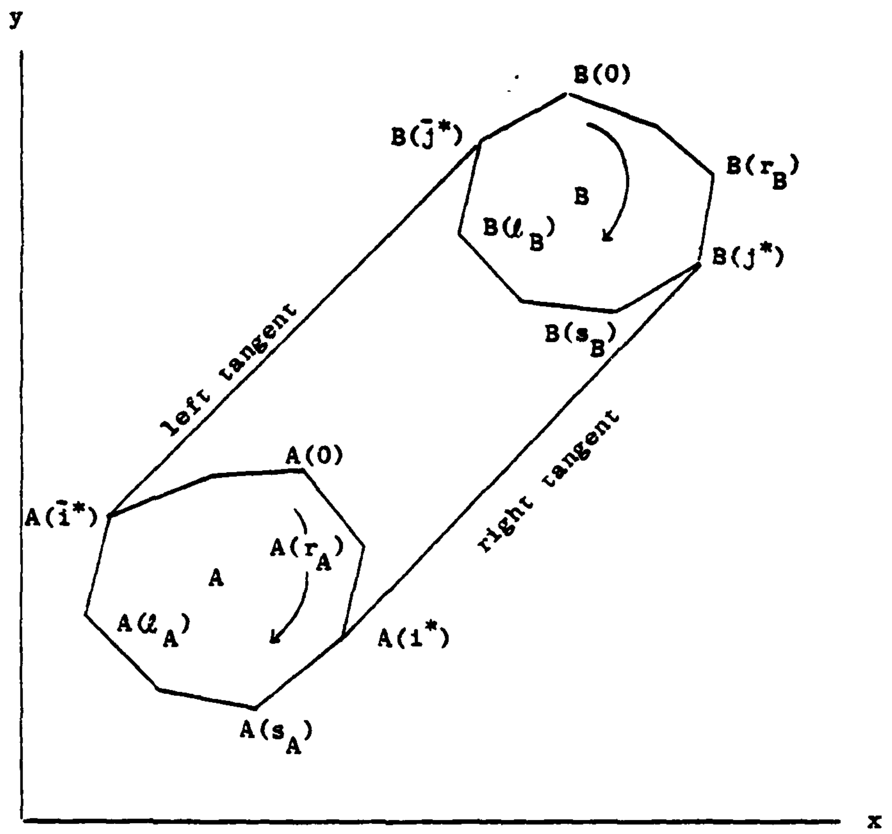


Figure 23. Illustration of the merging of two planar convex hulls.

$(A(\bar{i}^*), B(\bar{j}^*))$ are characterized by the following properties:

(1) $j^* = j^{(i^*)}$ and $\bar{j}^* = j^{(\bar{i}^*)}$

(2) $\alpha_{i^*,i^*-1} > \gamma_{i^*j^*}$ and $\alpha_{i^*,i^*+1} - \gamma_{i^*j^*} < \pi$;

$\alpha_{\bar{i}^*,\bar{i}^*+1} < \gamma_{\bar{i}^*\bar{j}^*}$ and $\alpha_{\bar{i}^*,\bar{i}^*-1} - \gamma_{\bar{i}^*\bar{j}^*} > \pi$.

Figure 24 clarifies these properties.

The index j^* has another property which is not so obvious as those above, as expressed by the following lemma:

Lemma 5.5. $j^* \le j^{(i)}$ for $0 \le i \le r_A$ when $0 \le j^* \le r_B$ and for $r_A \le i \le s_A$ when $r_B \le j^* \le s_B$.

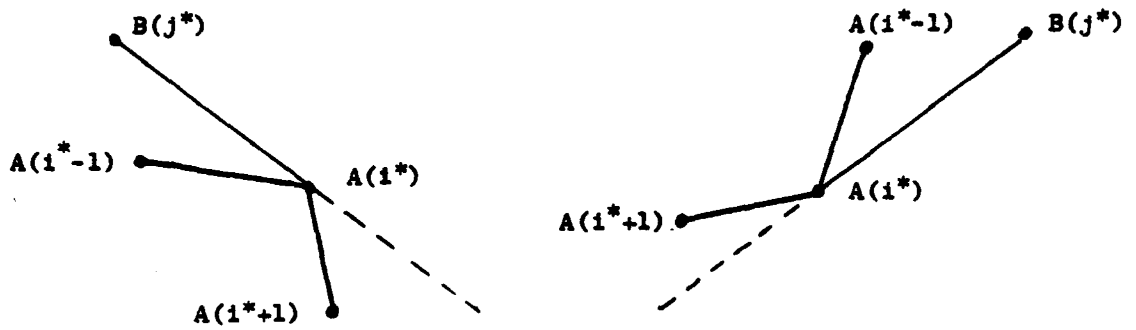
Proof: Suppose $j^* > j^{(k)}$ for some k in the appropriate range. Due to property (2) of i^* and j^*, A(k) must be in the hatched region, and due to property (1) and the assumption $j^* > j^{(k)}$, $B(j^{(k)})$ must be in the crosshatched region. We observe from Figure 25 that $\gamma_{k,j^{(k)}} > \gamma_{k,j^*}$ which contradicts the definition of $j^{(k)}$. Therefore, $j^* \le j^{(i)}$ for all i in the specified range. □

By a similar proof, we can show that the index $\bar{j}^*$ is largest among $\bar{j}^{(i)}$'s.

Lemma 5.6. $\bar{j}^* \ge \bar{j}^{(i)}$ for $s_A \le i \le \ell_A$ when $s_B \le \bar{j}^* \le \ell_B$ and for $\ell_A \le i \le n$ when $\ell_B \le \bar{j}^* \le m$.

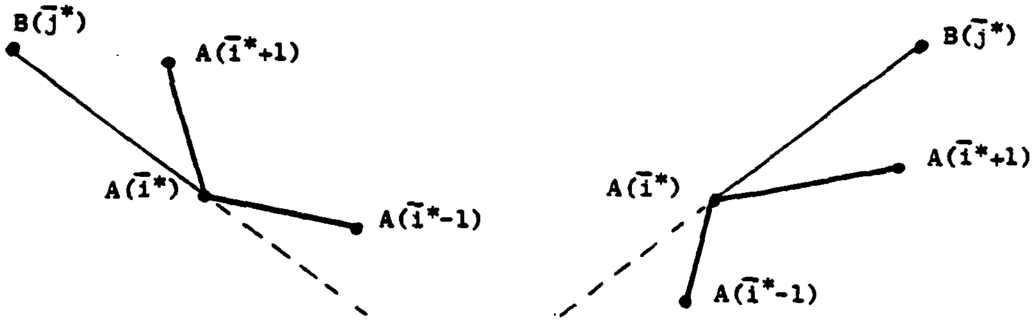
A merging algorithm for two convex polygons may consist of the following three major steps:

1. find j^* and $\bar{j}^*$;
2. determine i^* and $\bar{i}^*$ which, with j^* and $\bar{j}^*$, satisfy properties (1) and (2);
3. rearrange the vertices of the resulting polygon.



(i) $B(r_B)$ is to the left of $A(r_A)$ (ii) $B(r_B)$ is to the right of $A(r_A)$

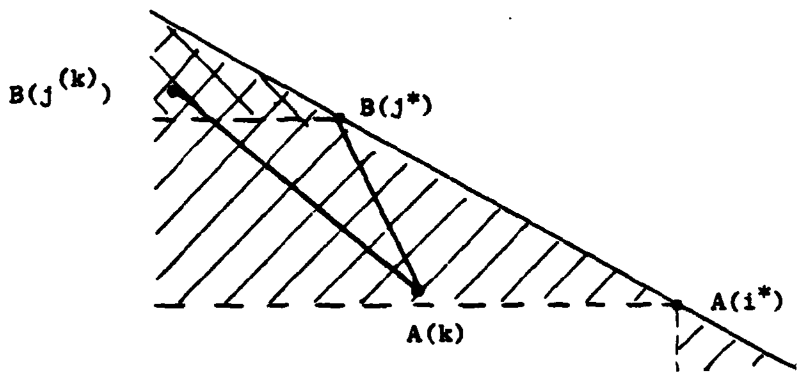
(a) $\alpha_{i^*,i^*-1} > \gamma_{i^*j^*}$ and $\alpha_{i^*,i^*+1} - \gamma_{i^*j^*} < \pi$



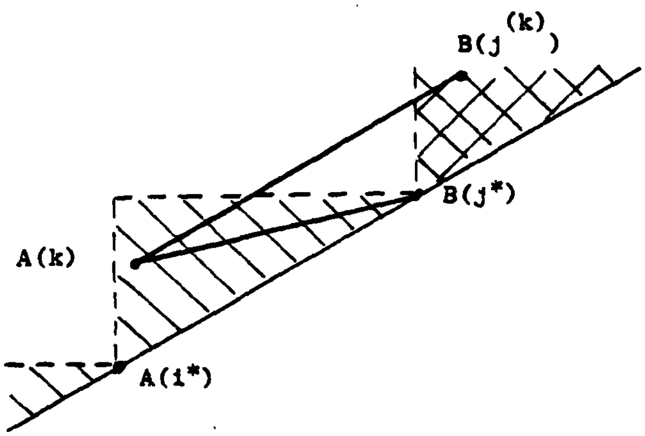
(i) $B(\ell_B)$ is to the left of $A(\ell_A)$ (ii) $B(\ell_B)$ is to the right of $A(\ell_A)$

(b) $\alpha_{\bar{i}^*,\bar{i}^*+1} < \gamma_{\bar{i}^*\bar{j}^*}$ and $\alpha_{\bar{i}^*,\bar{i}^*-1} - \gamma_{\bar{i}^*\bar{j}^*} > \pi$

Figure 24. Illustration of properties of tangents.



(a) $B(r_B)$ is left of $A(r_A)$



(b) $B(r_B)$ is right of $A(r_A)$

Figure 25. Proof of Lemma 5.5.

We shall describe the merging algorithm in more details in the following sections.

5.3 On the SMM with N Processors

In this section we shall present a "divide and conquer" algorithm for finding the convex hull of a set of N points in the plane on a SMM with N processors. We shall study methods for finding the minimum (maximum) of a $\vee$-bitonic ($\wedge$-bitonic) sequence and for merging two convex polygons on the SMM.

5.3.1 Finding the Minimum (Maximum) of a $\vee$-bitonic ($\wedge$-bitonic) Sequence

Given a $\vee$-bitonic ($\wedge$-bitonic) sequence $D(0: n-1)$, we want to find the smallest (largest) index k such that $D(k)$ is a minimum (maximum) of the sequence. The index k has the property that $D(k-1) > D(k) \leq D(k+1)$ ($D(k-1) \leq D(k) > D(k+1)$). Therefore, it is obvious that k can be found in constant time on a SMM with n processors and n memory units.

We are going to solve this problem on a SMM with $\sqrt{n}$ processors and n memory units. We first find the smallest (largest) index i such that $D(i\sqrt{n})$ is a minimum (maximum) of the sequence $(D(\sqrt{n}), D(2\sqrt{n}), \ldots, D((\sqrt{n}-1)\sqrt{n}))$. Note that this sequence is also $\vee$-bitonic ($\wedge$-bitonic). It is observed that k must be in the interval $[(i-1)\sqrt{n}+1, (i+1)\sqrt{n}-1]$ which is of length $2\sqrt{n}-1$; $(D(i-1)\sqrt{n}+1), \ldots, D(i\sqrt{n})$ and $D(i\sqrt{n}), \ldots, D(i+1)\sqrt{n}-1))$ are both $\vee$-bitonic sequences of length $\sqrt{n}$. Therefore, the index k can be determined in constant time with $\sqrt{n}$ processors. The function MIN_V_BITONIC is a formal description of the above method to determine the index k.

```
function MIN_V_BITONIC (D(0: n-1))

    /* this function returns the index k such that D(k-1) > D(k) ≤ D(k+1),
       when D is V-bitonic sequence */
    begin
        foreach j, j ∈ {1,2,...,√n-1)} do
            if D((j-1)√n) > D(j√n) and
               D(j√n) ≤ D((j+1)√n)
                  then i ← j
        foreach j, j ∈ {(i-1)√n+1,(i-1)√n+2,...,i√n} do
            if D(j-1) > D(j) and D(j) ≤ D(j+1) then k ← j

        foreach j, j ∈ {i√n,i√n+1,...,(i+1)√n-1} do
            if D(j-1) > D(j) and D(j) ≤ D(j+1) then k ← j

        return k
    end
```

We can obtain the function MAX_Λ_BITONIC for a Λ-bitonic sequence by interchanging > and ≤ in MIN_V_BITONIC.

5.3.2 Finding the Common Tangents of Two Convex Polygons

We now develop an algorithm for an SMM for finding the left tangent $(A(\bar{i}^*),B(\bar{j}^*))$ and the right tangent $(A(i^*),B(j^*))$, as defined in Section 5.2, for a SMM. Let us consider the determination of j^*. Assume that $B(r_B)$ is to the left of $A(r_A)$ (the other case can be treated in the same way). Since $j^{(i)}$, where $0 \le i \le r_A$, is the smallest index of the minimum of the $\vee$-bitonic sequence $(Y_{i,0},\ldots,Y_{i,r_B})$, $j^{(i)}$ can be found in constant time with $\sqrt{r_B+1}$ processors. We determine $j^{(i)}$ for $i = \sqrt{r_A+1}, 2\sqrt{r_A+1},\ldots,(\sqrt{r_A+1}-1)\sqrt{r_A+1}$ (refer to Figure 26). This can be achieved in constant time with $(\sqrt{r_A+1}-1)\sqrt{r_B+1}$ processors. Then we find the smallest index $\bar{i}$ such that $j^{(\bar{i})}$ is a minimum among $\{j^{(\sqrt{r_A+1})},\ldots,j^{((\sqrt{r_A+1}-1)\sqrt{r_A+1})}\}$. This can be done

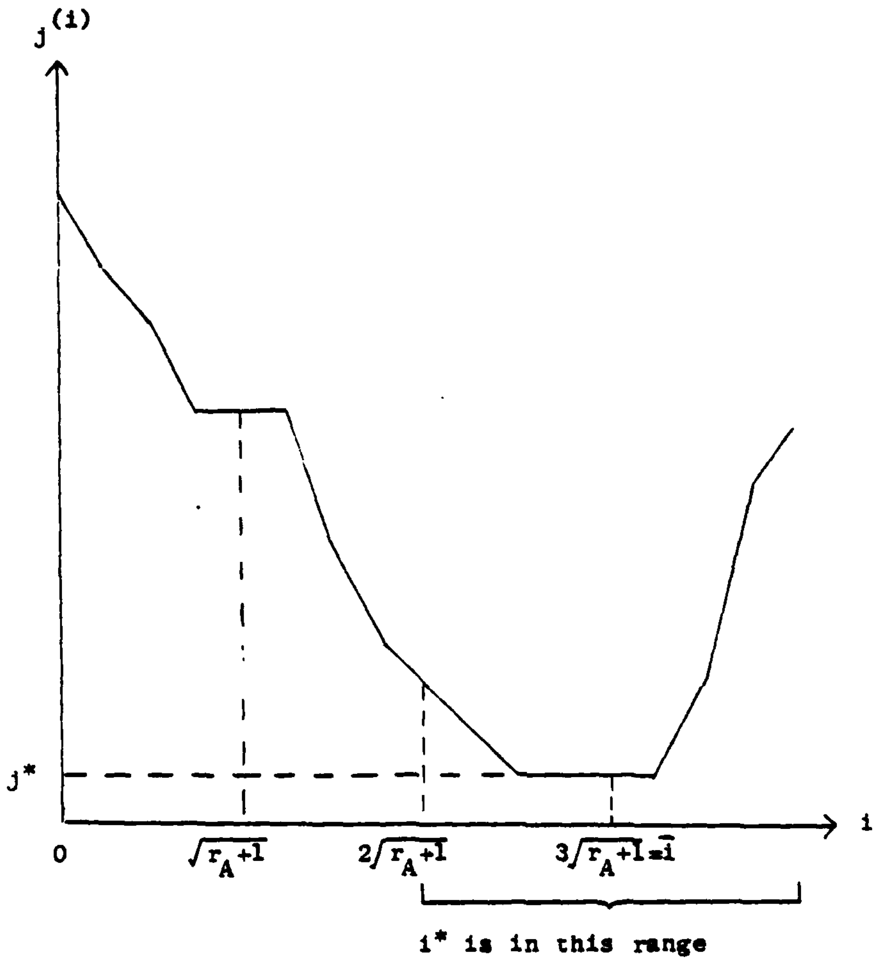


Figure 26. Determination of j^*.

in time $O((\log(\sqrt{r_A+1}-1))^2)$ with $\sqrt{r_A+1}-1$ processors (refer to Section 2.1.2). The index j* is the smallest in the set $\{j^{(\bar{i}-\sqrt{r_A+1}+1)}, j^{(\bar{i}-\sqrt{r_A+1}+2)}, \ldots, j^{(\bar{i}+\sqrt{r_A+1}-1)}\}$ of size $2\sqrt{r_A+1}-1$. Therefore j* can be found in $O((\log n)^2)$ time on an SMM with $\sqrt{nm}$ processors. The index $\bar{j}$* can be determined in a similar way. The indices i* and $\bar{i}$* are the two i's which satisfy properties (1) and (2) as described in Section 5.2. Knowing j* and $\bar{j}$*, the indices i* and $\bar{i}$* can be determined in constant time with n processors. We shall present formally, in the appendix, the procedure TANGENTS which determines and returns the indices j*, i*, $\bar{j}$* and $\bar{i}$*.

In conclusion, the left and right tangents can be determined in time $O((\log n)^2)$ with at most $m+n$ processors. Next, we shall consider the entire convex hulls algorithm.

5.3.3 Convex Hulls Algorithm

As a preliminary step, we sort the set S of points by their y coordinates in descending order. This can be done in $O((\log N)^2)$ time with N processors. The convex hulls algorithm to be presented is a recursive program. The major step is the merging procedure which determines the left and right tangents of two convex hulls and rearranges the vertices of the resulting hull.

```
function CH21 (S)

    /* returns CH(S); S is a set of N points in the plane */
    begin if N ≤ 2 then return (S)
        return (MERGE1(CH21(S(N/2: N-1)),CH21(S(0 : N/2-1))))
    end

function MERGE1(A,B):

    /* returns the convex hull of polygons A and B */
    begin
        (j*, i*, j̄*, ī*) ← TANGENTS1(A,B)
        foreach k, 0 ≤ k ≤ j* do C(k) ← B(k)
        foreach k, i* ≤ k ≤ ī* do C(j* - i* + 1 + k) ← A(k)
        foreach k, j̄* ≤ k < m do C(j* - i* + 2 + ī* - j̄* + k) ← B(k)
        return (C(0: j* - i* + ī* - j̄* + m - 1))
    end
```

The running time T(N) of function CH21 can be obtained by recurrence relation $T(N) \le T(N/2) + M(N)$, where M(N) is the running time of function MERGE1. We have shown that the tangents can be found in $O((\log N/2)^2)$ with N processors, and it obvious that the rearrangement can be done in constant time. Therefore, $M(N) = O((\log N)^2)$. Hence $T(N) = O((\log N)^2)$.

Theorem 5.3. The convex hull of a set of N points in the plane can be determined in time $O((\log N)^2)$ on a SMM with N processors and N memory units.

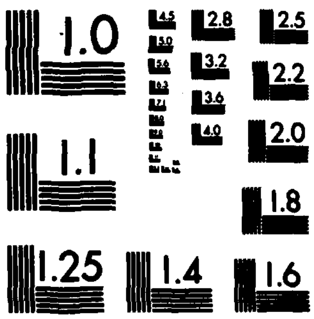
5.4 On the CCC with N Processors

In this section we discuss how the convex hulls algorithm developed in Section 5.3 can be implemented on a CCC with N processors in $O((\log N)^2)$ parallel steps. We shall discuss the data movement in detail.

AD A124 353 PARALLEL ALGORITHMS FOR GEOMETRIC PROBLEMS(U) ILLINOIS UNIV AT URBANA APPLIED COMPUTATION THEORY GROUP A L CHOW DEC 81 ACT-30 N00014-79-C-0424 2/2

UNCLASSIFIED F/G 12/1 NL

END
DATE
FILMED
83
DTIC



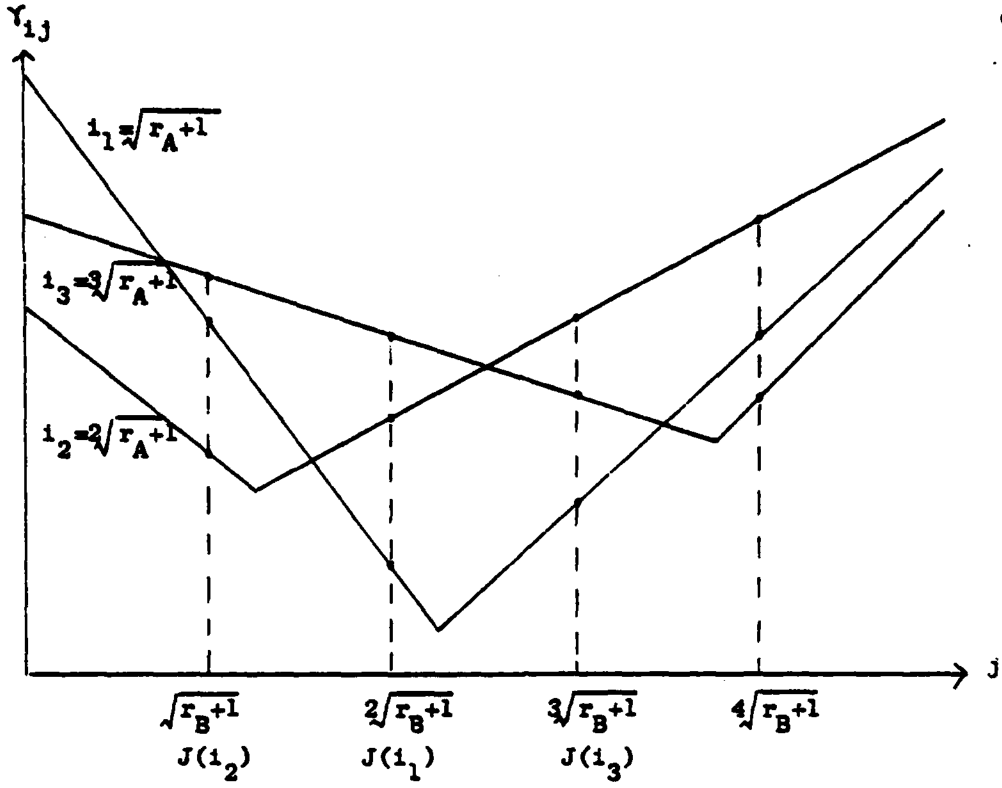
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

5.4.1 Finding the Left and Right Tangents of Two Convex Polygons

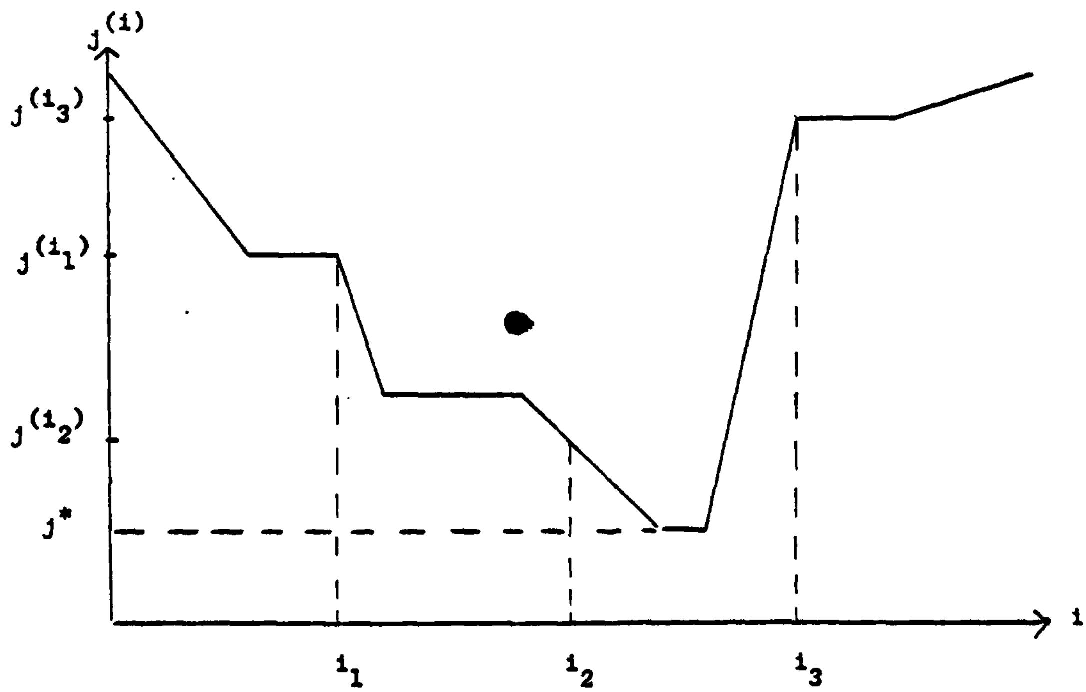
The function TANGENTS1 introduced in Section 5.3.2 for determining the indices of the extremes of the left and the right tangents of two convex polygons cannot be directly implemented on a CCC. We shall make some modifications to TANGENTS1 so that it will be suitable for implementation on a CCC.

Using the facts that j^* is the minimum among the $j^{(i)}$'s and that the sequences of $\gamma_{i,j}$'s are $\vee$-bitonic, we can determine j^* as follows.

First of all (refer to Figure 27 for the following discussion), we describe how to determine simultaneously a set of integers $\{J(i), i = \sqrt{r_A+1}, 2\sqrt{r_A+1}, \ldots, (\sqrt{r_A+1}-1)\sqrt{r_A+1}\}$, where $\gamma_{i,J(i)} = \min\{\gamma_{i,\sqrt{r_B+1}}, \gamma_{i,2\sqrt{r_B+1}}, \ldots, \gamma_{i,(\sqrt{r_B+1}-1)\sqrt{r_B+1}}\}$ if $B(r_B)$ is to the left of $A(r_A)$; and a set of integers $\{J(i), i = r_A + \sqrt{s_A - r_A + 1}, r_A + 2\sqrt{s_A - r_A + 1}, \ldots, r_A + (\sqrt{s_A - r_A + 1}-1)\sqrt{s_A - r_A + 1}\}$, where $\gamma_{i,J(i)} = \min\{\gamma_{i,r_B+\sqrt{s_B-r_B+1}}, \gamma_{i,r_B+2\sqrt{s_B-r_B+1}}, \ldots, \gamma_{i,r_B+(\sqrt{s_B-r_B+1}-1)\sqrt{s_B-r_B+1}}\}$ if $B(r_B)$ is not to the left of $A(r_A)$. We now consider two duplicating patterns of a data array $D(0: q-1)$; (i) the first pattern, to be referred to as $P1(\ell)$ consists in duplicating D ℓ times into $\{D(0), D(1), \ldots, D(q-1), D(0), \ldots, D(q-1), \ldots\}$ (ii) the second pattern, to be referred to as $P2(\ell)$, consists in duplicating each element of D ℓ times into $\{D(0), D(0), \ldots, D(0), D(1), \ldots, D(1), \ldots, D(q-1), \ldots, D(q-1)\}$. Both patterns have $q \cdot \ell$ elements. The first pattern $P1(\ell)$ can be achieved by copying each element of $\{D(0), \ldots, D(q-1)\}$



(a) γ_{ij} vs. j, for $i = \sqrt{r_A+1},\ 2\sqrt{r_A+1},\ 3\sqrt{r_A+1}$



(b) $j^{(i)}$ vs. i

Figure 27. Graphical illustration of the determination of j^* on the CCC.

into the module q positions away, then copying each element of $\{D(0),\ldots,D(q-1),D(0),\ldots,D(q-1)\}$ into the module 2q positions away, and so on. It will take logarithmic steps to achieve the pattern $P1(\ell)$. We achieve the second pattern $P2(\ell)$ as follows. We copy $D(0),D(1),\ldots,D(q-1)$ into modules $0,\ell,2\ell,\ldots,(q-1)\ell$ respectively by a reverse process of the concentration procedure described in Section 2.2.1. We then perform a selected broadcasting as described in Section 2.2.2 to achieve pattern $P2(\ell)$. Recall that both of these operations can be achieved in logarithmic time. Therefore, both patterns can be achieved on a CCC with $q\cdot\ell$ processors in $O(\log(q\cdot\ell))$ steps. We shall discuss only the case that $B(r_B)$ is to the left of $A(r_A)$; The other case can be treated in a similar manner. We duplicate $\{B(\sqrt{r_B+1}),B(2\sqrt{r_B+1}),\ldots,B((\sqrt{r_B+1}-1)\sqrt{r_B+1})\}$ into pattern $P1(\sqrt{r_A+1}-1)$ and $\{A(\sqrt{r_A+1}),A(2\sqrt{r_A+1}),\ldots,A((\sqrt{r_A+1}-1)\sqrt{r_A+1})\}$ into pattern $P2(\sqrt{r_B+1}-1)$. Now we can compute $\{\gamma_{\sqrt{r_A+1},\sqrt{r_B+1}},\gamma_{\sqrt{r_A+1},2\sqrt{r_B+1}},\ldots,$

$\gamma_{\sqrt{r_A+1},(\sqrt{r_B+1}-1)\sqrt{r_B+1}},\gamma_{2\sqrt{r_A+1},\sqrt{r_B+1}},\ldots,\gamma_{2\sqrt{r_A+1},(\sqrt{r_B+1}-1)\sqrt{r_A+1}},\ldots\}$

in constant time. Since sequences $(\gamma_{i,\sqrt{r_B+1}},\ldots,\gamma_{i,(\sqrt{r_B+1}-1)\sqrt{r_B+1}}\}$, for $i=\sqrt{r_A+1},2\sqrt{r_A+1},\ldots,(\sqrt{r_A+1}-1)\sqrt{r_A+1}$, are $\vee$-bitonic, the indices $J(i)$'s of the minima of the sequences can be determined in $O(\log\sqrt{nm})$ time. Figure 27(a) shows three $\vee$-bitonic sequences $(\gamma_{i,\sqrt{r_B+1}},\gamma_{i,2\sqrt{r_B+1}},\ldots)$, for $i=\sqrt{r_A+1},2\sqrt{r_A+1},3\sqrt{r_A+1}$, and the values of $J(i)$. The index j', the minimum of $J(i)$, can be determined in $O(\log\sqrt{nm})$ time on the CCC. We then determine $J'(i)$, where $\gamma_{i,J'(i)}=\min\{\gamma_{i,j'-\sqrt{r_B+1}+1},\gamma_{i,j'-\sqrt{r_B+1}+2},$

$\ldots,\gamma_{i,j'},\ldots,\gamma_{i,j'+\sqrt{r_B+1}-1}\}$ for $i=\sqrt{r_A+1},2\sqrt{r_A+1},\ldots,(\sqrt{r_A+1}-1)\sqrt{r_A+1}$

in the same way as we determine $J(i)$. We also find $\bar{i}$ which is the smallest index such that $J'(\bar{i})$ is a minimum among $\{J'(\sqrt{r_A+1}), J'(2\sqrt{r_A+1}), \ldots, J'((\sqrt{r_A+1}-1)\sqrt{r_A+1})\}$. It is easy to show that $\bar{i}$ be determined in $O(\log\sqrt{nm})$ on the CCC. Now j^* is the minimum of $\{j^{(\bar{i}-k+1)}, j^{(\bar{i}-k+2)}, \ldots, j^{(\bar{i}+k-1)}\}$ and can be found in a procedure similar to the one given above. The procedure R_TANGENT_INDEX, which is a formal description of what we discussed above, will be presented in the appendix.

In an analogous way, we can describe a procedure L_TANGENT_INDEX (A,B) which returns $\bar{j}^*$. Knowing j^* and $\bar{j}^*$, we can determine i^* and $\bar{i}^*$ by finding pairs of (i', j^*) and $(i'', \bar{j}^*)$ which satisfy properties (1) and (2) defined in Section 5.2.

```
function TANGENTS2(A,B)

    /* return the indices of the extremes of left and right tangent
       of A and B  */
    begin

        /* determine j* and j̄*  */
        j* ← R_TANGENT_INDEX(A,B)
        j̄* ← L_TANGENT_INDEX(A,B)

        /* determine i* and ī* with which j* and j̄* respectively
           satisfy property (1) and (2)  */
        if x-values of B(r_B) < x-values of A(r_A)
            then begin a ← 0; b ← r_A; end; else begin a ← r_A; b ← s_A; end
        foreach i, a ≤ i ≤ b do
            if γ_{i,j*-1} > γ_{i,j*} ≤ γ_{i,j*+1}    /* j* = j^(i)  */
                and α_{i,i-1} > γ_{i,j*} and α_{i,i+1} - γ_{i,j*} < π  /* property (2) */
                then i* ← i

        if x-values of B(ℓ_B) < x-values of A(ℓ_A)
            then begin a ← s_A; b ← ℓ_A; end; else begin a ← ℓ_A; b ← n; end
        foreach i, a ≤ i ≤ b do
```

if $\gamma_{i,j^*-1} > \gamma_{i,\bar{j}^*} \le \gamma_{i,\bar{j}^*+1}$ /* $\bar{j} = j^{(i)}$ */
  and $\alpha_{i,i+1} < \gamma_{i,\bar{j}^*}$ **and** $\alpha_{i,i-1} - \gamma_{i,\bar{j}^*} > \pi$
  then $\bar{i}^* \leftarrow i$
return $(j^*, i^*, \bar{j}^*, \bar{i}^*)$
end

Therefore, the left and right tangents can be determined in time $O(\log(n+m))$ on a CCC with $n+m$ processors. Next, we shall consider the entire convex hulls algorithm

5.4.2 Convex Hulls Algorithm

We presort the set S of points by their y coordinates in descending order. This can be done in time $O((\log N)^2)$ on a CCC with N processors [31]. The convex hulls algorithm has the same structure as the one described in Section 5.3.3. The main difference is in the merging step.

function MERGE2(A,B)
  begin /* determine the tangents */
    $(j^*, i^*, \bar{j}^*, \bar{i}^*) \leftarrow$ TANGENTS2(A,B)

    /* reorder the vertices */
    foreach $i,\ 0 \le i < n$ **do** $T2(i) \leftarrow A(i)$
    foreach $i,\ 0 \le i < m$ **do** $T1(i) \leftarrow T3(i) \leftarrow B(i)$
    if $j^*+1 > i^*$ **then** shift T2 forward by j^*+1-i^* positions
        else shift T2 backward by i^*-j^*-1 positions
    if $(j^*+1+\bar{i}^*-i^*) > \bar{j}^*$ **then** shift T3 forward by $j^*+\bar{i}^*-i^*+2-\bar{j}^*$ positions
        else shift T3 backward by $\bar{j}^*-(j^*+\bar{i}^*-i^*+2)$ positions
    foreach $i,\ 0 \le i \le j^*$ **do** $C(i) \leftarrow T1(i)$
    foreach $i,\ j^*+1 \le i \le j^*+\bar{i}^*-i^*+1$ **do** $C(i) \leftarrow T2(i)$
    foreach $i,\ j^*+\bar{i}^*-i^*+2 \le i \le j^*+\bar{i}^*-i^*+2+m-\bar{j}^*$
      do $C(i) \leftarrow T3(i)$
    return $(C(0{:}\ j^*-i^*+\bar{i}^*-\bar{j}^*+m+1))$
  end

Cyclic forward or backward shift of an array of data can be implemented on a CCC with $n+m$ processors in $O(\log(n+m))$ parallel steps. Therefore, MERGE2 runs in time $O(\log(n+m))$ on a CCC with $n+m$ processors. We immediately obtain an $O((\log N)^2)$ algorithm for finding the convex hull of N points in the plane.

```
function CH22 (S(0: N-1)):
    /* returns CH(S); S is presorted by y coordinates in descending order */
    begin if N ≤ 2 then return (S)
        else return (MERGE2(CH22(S(N/2:N-1)),CH22(S(0:N/2-1))));
    end .
```

Theorem 5.4. The convex hull of a set of N points in the plane can be determined in time $O((\log N)^2)$ on a CCC with N processors.

5.5 On the CCC with $2N^{1+\alpha}$ Processors

In this section we shall develop a "divide and conquer" algorithm for finding the convex hull of a set S of N points in the plane on a CCC with $2N^{1+\alpha}$ processors, $0 < \alpha \leq 1$. We partition S into N^{α} subsets $S_0, S_1, \ldots, S_{N^{\alpha}-1}$ of $N^{1-\alpha}$ elements each. We then determine convex hulls $CH(S_0), \ldots, CH(S_{N^{\alpha}-1})$ simultaneously. Finally $CH(S_0), \ldots, CH(S_{N^{\alpha}-1})$ are merged to give CH(S). Since the determinations of $CH(S_0), \ldots, CH(S_N$ recursive calls, we obtain for the running time T(N) of this algorithm the recurrence relation

$$T(N) = T(N^{1-\alpha}) + M(N),$$

where M(N) is the time to merge $CH(S_0), \ldots, CH(S_{N^{\alpha}-1})$. If we can show that N^{α} convex hulls can be merged in time $O(\log N)$ with $2N^{1+\alpha}$ processors, then we have $T(N) = O(\frac{1}{\alpha} \log N)$.

We shall define some terms and then describe the merger, which is a major part of our convex hulls algorithm.

5.5.1 Notations and Definitions

Consider a set of polygons $A_0, A_1, \ldots, A_{n^{\alpha}-1}$ $(0 < \alpha \leq 1)$, each having at most $n^{1-\alpha}$ vertices. Each A_i is in standard form, that is $A_i(0: n_i-1)$ is the clockwise sequence of its vertices starting with the one with largest y coordinate. Variables n_i, r_i, s_i, ℓ_i denote the indices of the topmost rightmost, bottommost and leftmost vertices of A_i. We assume that the y-coordinates of $A_k(0: n_k-1)$ less than those of $A_\ell(0: n_\ell-1)$ for $k > \ell$, that is in any horizontal slab there will be only one A_i. The indices of the extremes of the left and the right tangents of A_k and $A_\ell (k > \ell)$ are $\bar{j}^*_{k,\ell}, \bar{i}^*_{k,\ell}, j^*_{k,\ell}, i^*_{k,\ell}$ respectively (refer to Figure 28). We define the polar angles $\delta_{k,\ell} = \theta(A_k(\bar{i}^*_{k,\ell}), A_\ell(\bar{j}^*_{k,\ell}))$ and $\phi_{k,\ell} = \theta(A_k(i^*_{k,\ell}), A_\ell(j^*_{k,\ell}))$.(1)

5.5.2 Merging Multiple Convex Hulls

We shall discuss how to merge the set of N^α convex polygons, $A_0, \ldots, A_{N^\alpha-1}$, as introduced in Section 5.5.1. Like merging two convex polygons, we have to determine those vertices belonging to the resulting convex hull and those becoming internal to the resulting convex hull; then we have to rearrange the vertices. We shall develop some preliminary tools first.

Lemma 5.7. If $\delta_{i,k} < \delta_{i,\ell}$ or $\delta_{i,k} = \delta_{i,\ell}$ and $\ell < k$, for k and $\ell < i$, then $(A_i(\bar{i}^*_{i,k}), A_k(\bar{j}^*_{i,k}))$ is not an edge of the resulting convex hull of $A_0, \ldots, A_{N^\alpha-1}$.

Proof: We have to consider two cases (a) $\ell < k$ and (b) $\ell > k$. Referring to Figure 29, in both cases, the edge $(A_i(\bar{i}^*_{i,k}), A_k(\bar{j}^*_{i,k}))$ becomes internal to the edge $(A_i(\bar{i}^*_{i,\ell}), A_\ell(\bar{j}^*_{i,\ell}))$. □

(1) In the implementation, the operation of comparing two angles will be replaced by the operation of comparing the negative values of their cotangents as in the case of $\alpha_{i,j}$ and $\gamma_{i,j}$.

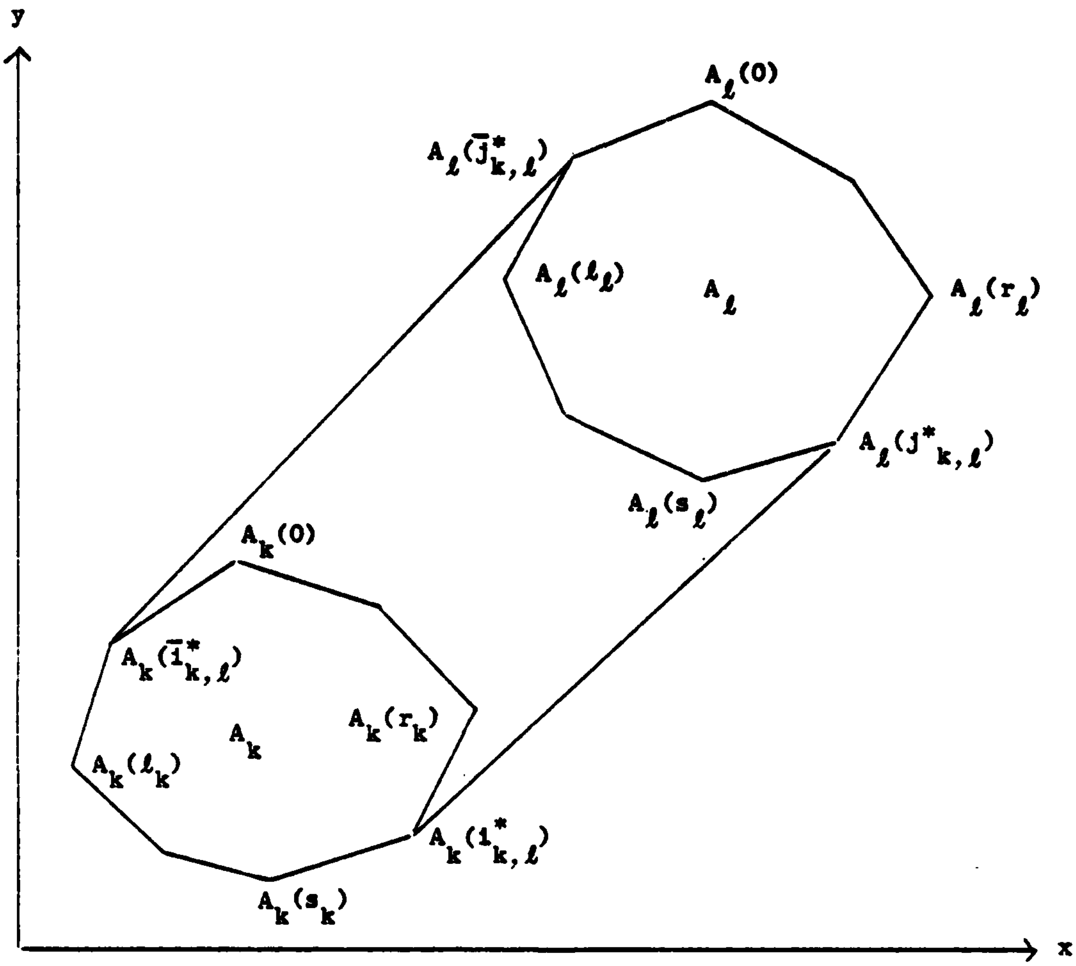
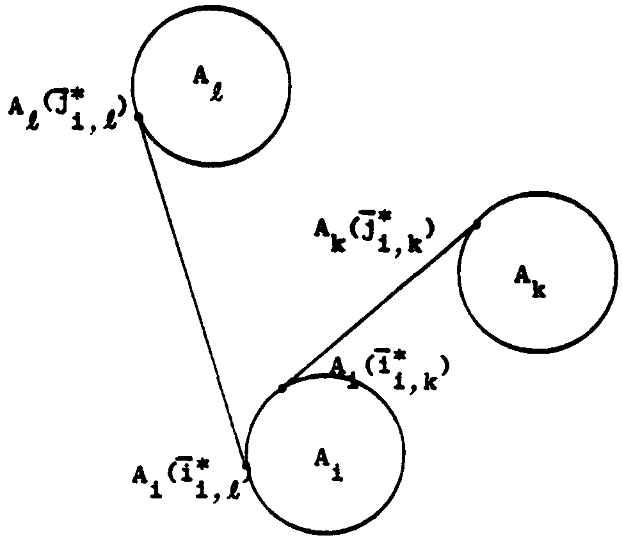
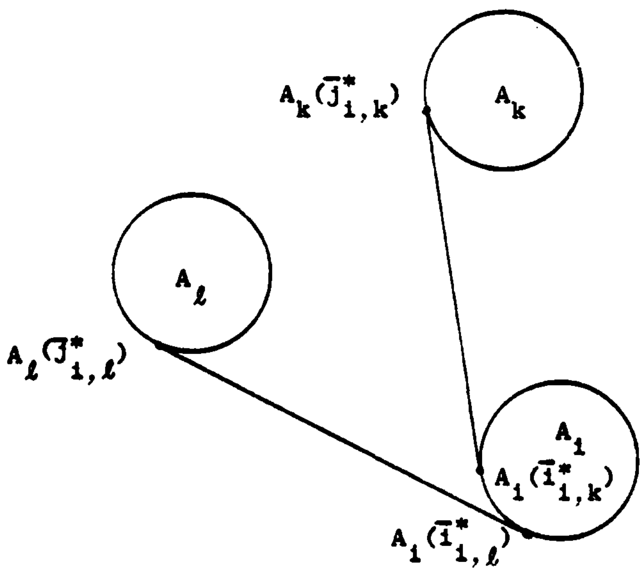


Figure 28. Notations for two convex polygons A_ℓ and A_k being merged.



(a) $\ell < k$



(b) $\ell > k$

Figure 29. Proof of Lemma 5.7.

We associate with each polygon A_i an index $\bar{t}(i)(< i)$ which is the smallest index such that $\delta_{i,\bar{t}(i)} \geq \delta_{i,k}$, $0 \leq k < i$. Using Lemma 5.7, we have the following result.

Corollary 5.2. Among all edges $(A_i(\bar{i}^*_{i,k}), A_k(\bar{j}^*_{i,k}))$ $(0 \leq k < i)$, $(A_i(\bar{i}^*_{i,\bar{t}(i)}), A_{\bar{t}(i)}(\bar{j}^*_{i,\bar{t}(i)}))$ (to be referred to as edge candidate) is the only candidate for being an edge of the resulting convex hull of $A_0,\ldots,A_{N^\alpha-1}$.

We now consider polygons below A_i.

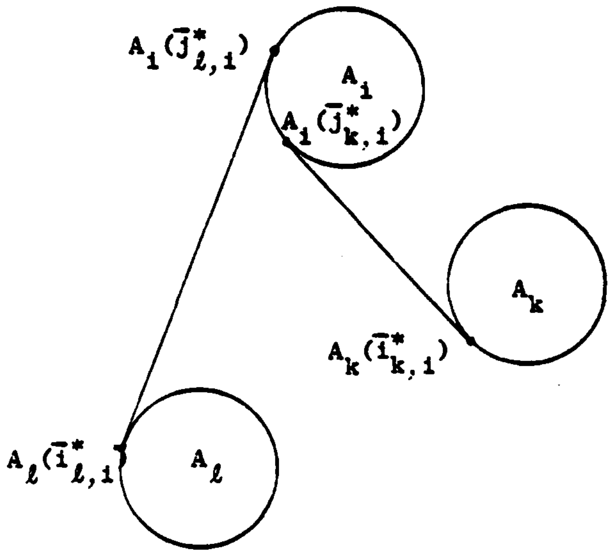
Lemma 5.8. If $\delta_{k,i} > \delta_{\ell,i}$ or $\delta_{k,i} = \delta_{\ell,i}$ and $k < \ell$ for $k,\ell > i$ then $(A_i(\bar{j}^*_{k,i}), A_k(\bar{i}^*_{k,i}))$ is not an edge of the resulting convex hull of $A_0,\ldots,A_{N^\alpha-1}$.

Proof: We have considered two cases (a) $k < \ell$ and (b) $k > \ell$. Referring to Figure 30, in both cases, the edge $(A_i(\bar{j}^*_{k,i}), A_k(\bar{i}^*_{k,i}))$ become internal to edge $(A_i(\bar{j}^*_{\ell,i}), A_\ell(\bar{i}^*_{\ell,i}))$. □

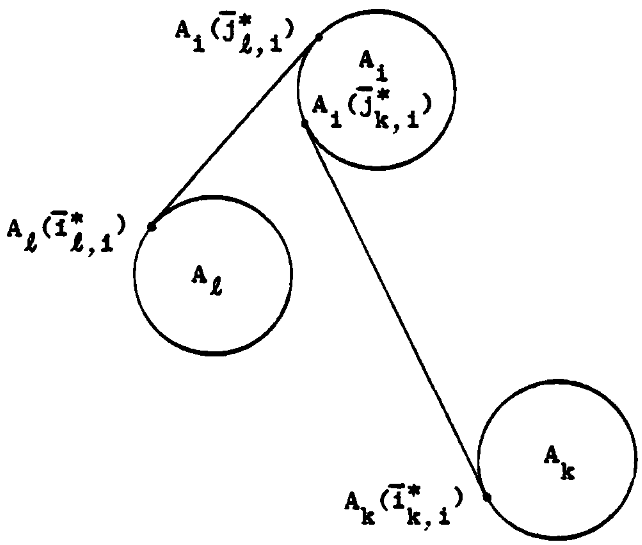
We associate with each A_i an index $\bar{b}(i)$ $(> i)$ which is the largest index such that $\delta_{\bar{b}(i),i} \leq \delta_{k,i}$, $i < k \leq N^\alpha-1$. Again using Lemma 5.8, we have this result.

Corollary 5.3. Among all edges $(A_i(\bar{j}^*_{k,i}), A_k(\bar{i}^*_{k,i}))$ $(i < k \leq N^\alpha-1)$, $(A_i(\bar{j}^*_{\bar{b}(i),i}), A_{\bar{b}(i)}(\bar{i}^*_{\bar{b}(i),i}))$ (to be referred to as edge candidate) is the only candidate for being an edge of the convex hull of $A_0,\ldots,A_{N^\alpha-1}$.

We are now able to determine if the edge candidates are edges of the convex hull of $A_0,\ldots,A_{\alpha-1}$ as follows.



(a) $k < \ell$



(b) $k > \ell$

Figure 30. Proof of Lemma 5.8.

Theorem 5.5. The edge candidates are edges of the convex hull of $A_0,\ldots,A_{N^\alpha-1}$ if and only if $\bar{i}^*_{i,\bar{t}(i)} > \bar{j}^*_{\bar{b}(i),i}$ or ($\bar{i}^*_{i,\bar{t}(i)} = \bar{j}_{\bar{b}(i),i}$ and $\bar{\omega} = \theta(A_i(\bar{j}^*_{\bar{b}(i),i}), A_{\bar{b}(i)}(\bar{i}^*_{\bar{b}(i),i})) - \theta(A_i(\bar{i}^*_{i,\bar{t}(i)}), A_{\bar{t}(i)}(\bar{j}^*_{i,\bar{t}(i)})) > \pi$).

Proof: Suppose $\bar{i}^*_{i,\bar{t}(i)} < \bar{j}^*_{\bar{b}(i),i}$ (refer to Figure 31(a)) or $\bar{i}^*_{i,\bar{t}(i)} = \bar{j}^*_{\bar{b}(i),i}$ and $\bar{\omega} \le \pi$ (refer to Figure 31(b)). We have $\delta_{\bar{b}(i),i} < \delta_{\bar{b}(i),\bar{t}(i)}$ and $\delta_{i,\bar{t}(i)} > \delta_{\bar{b}(i),\bar{t}(i)}$. Thus, by Lemmas 5.7 and 5.8, edges $(A_{\bar{b}(i)}(\bar{i}^*_{\bar{b}(i),i}), A_i(\bar{j}^*_{\bar{b}(i),i})$ and $A_{\bar{t}(i)}(\bar{j}^*_{i,\bar{t}(i)}), A_i(\bar{i}^*_{i,\bar{t}(i)}))$ are not edges of convex hull of $A_0,\ldots,A_{N^\alpha-1}$.

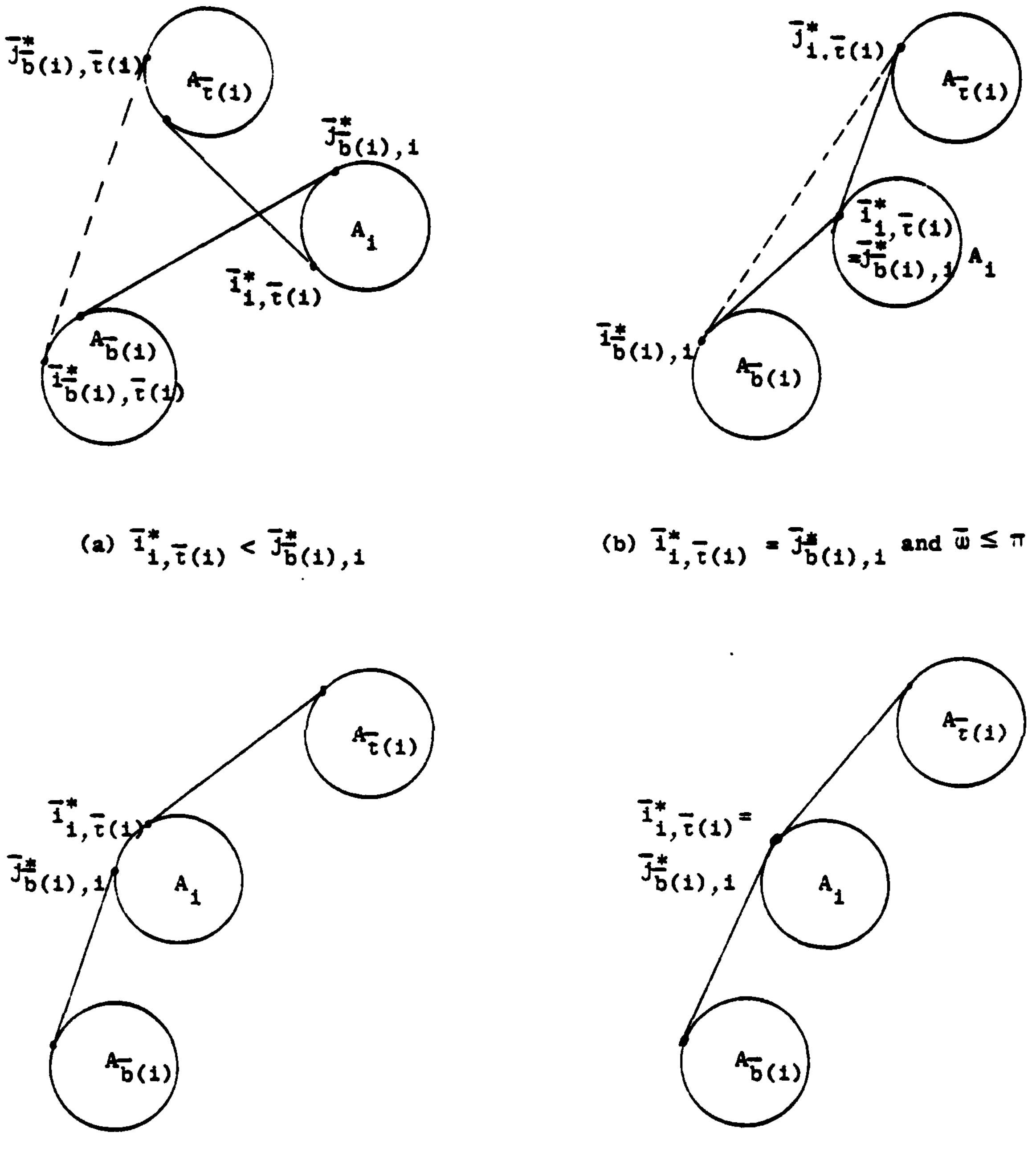
Suppose $\bar{i}^*_{i,\bar{t}(i)} > \bar{j}^*_{\bar{b}(i),i}$ (refer to Figure 31(c)) or $\bar{i}^*_{i,\bar{t}(i)} = \bar{j}^*_{\bar{b}(i),i}$ and $\bar{\omega} > \pi$ (refer to Figure 31 (d)). By the definitions of $\bar{t}(i)$ and $\bar{b}(i)$, all $A_0,\ldots,A_{N^\alpha-1}$ are on the same side of the edge candidates. Thus, the candidates are edges of convex hull of $A_0,\ldots,A_{N^\alpha-1}$. □

We now describe the analog for the right tangents. The index $t(i)$ is the smallest one such that $\phi_{i,t(i)} \le \phi_{i,k}$, $0 \le k < i$. And the index $b(i)$ is the largest such that $\phi_{b(i),i} \ge \phi_{k,i}$, $i < k \le N^\alpha-1$. We shall state without proof the analogous lemmas, corollaries, and theorems for the right tangents.

Lemma 5.9. If $\phi_{i,k} > \phi_{i,\ell}$ or $\phi_{i,k} = \phi_{i,\ell}$ and $\ell < k$, for $k,\ell < i$ then $(A_i(i^*_{i,k}), A_k(j^*_{i,k}))$ is not an edge of the resulting convex hull of $A_0,\ldots,A_{N^\alpha-1}$.

Corollary 5.4. Among all edges $(A_i(i^*_{i,k}), A_k(j^*_{i,k}))$ $(0 \le k < i)$, $(A_i(i^*_{i,t(i)}), A_{t(i)}(j^*_{i,t(i)}))$ is the only edge candidate.

Lemma 5.10. If $\phi_{k,i} < \phi_{\ell,i}$ or $\phi_{k,i} = \phi_{\ell,i}$ and $k < \ell$, for $k,\ell > i$ then $(A_i(j^*_{k,i}), A_k(i^*_{k,i}))$ is not an edge of the convex hull of $A_0,\ldots,A_{N^\alpha-1}$.



(a) $\bar{i}^*_{i,\bar{\tau}(i)} < \bar{j}^*_{\bar{b}(i),i}$

(b) $\bar{i}^*_{i,\bar{\tau}(i)} = \bar{j}^*_{\bar{b}(i),i}$ and $\bar{\omega} \le \pi$

(c) $\bar{i}^*_{i,\bar{\tau}(i)} > \bar{j}^*_{\bar{b}(i),i}$

(d) $\bar{i}^*_{i,\tau(i)} = \bar{j}^*_{\bar{b}(i),i}$ and $\bar{\omega} > \pi$

Figure 31. Illustration of proof of Theorem 5.5.

Corollary 5.5. Among all edges $(A_i(j^*_{k,i}), A_k(i^*_{k,i}))$ $(i < k \leq N^\alpha-1)$, $(A_i(j^*_{b(i),i}), A_{b(i)}(i^*_{b(i),i}))$ is the only edge candidate.

Theorem 5.6. The edge candidates are edges of the convex hull of $A_0, \ldots, A_{N^\alpha-1}$ if and only if $i^*_{i,t(i)} < j^*_{b(i),i}$ or $(i^*_{i,t(i)} = j^*_{b(i),i}$ and $\omega = \theta(A_i(j^*_{b(i),i}), A_{b(i)}(i^*_{b(i),i})) - \theta(A_i(i^*_{i,t(i)}), A_{t(i)}(j^*_{i,t(i)})) < \pi)$.

Before discussing how to obtain indices $t(i)$, $b(i)$, $\bar{t}(i)$, and $\bar{b}(i)$, etc., we present an example of merging five convex polygons in Figure 32. In Figure 32, $\delta_{20} > \delta_{21}$; therefore by Lemma 5.7 and Corollary 5.2, $(A_2(\bar{i}^*_{2,0}), A_0(\bar{j}^*_{2,0}))$ is an edge candidate while edge $(A_2(\bar{i}^*_{2,1}), A_0(\bar{j}^*_{2,1}))$ is eliminated. Also $\delta_{42} > \delta_{32}$, therefore by Lemma 5.8 and Corollary 5.3, $(A_2(\bar{j}^*_{32}), A_3(\bar{i}^*_{32}))$ is an edge candidate while edge $(A_2(\bar{j}^*_{42}), A_4(\bar{i}^*_{42}))$ is eliminated. However, by Theorem 5.5, both of these edge candidates will be eliminated because $\bar{i}^*_{2,0} < \bar{j}^*_{3,2}$. With similar arguments, all lines of support, except those shown in the figure, will be eliminated. The resulting convex hull is $(A_0(0), A_0(1), \ldots, A_0(j^*_{1,0}), A_1(i^*_{1,0}), A_1(i^*_{1,0}+1), \ldots, A_1(j^*_{4,1}), A_4(i^*_{4,1}), A_4(i^*_{4,1}+1), \ldots, A_4(\bar{i}^*_{4,3}), A_3(\bar{j}^*_{4,3})$ $A_3(\bar{j}^*_{4,3}+1), \ldots, A_3(\bar{i}^*_{3,1}), A_0(\bar{j}^*_{3,0}), A_0(\bar{j}^*_{3,0}+1), \ldots, A_0(n_0-1))$. We now discuss how to obtain the resulting convex hull in the general case. We first copy $A_0, \ldots, A_{N^\alpha-1}$ into the following pattern P3:

$$A_0A_1A_0A_2\cdots A_0A_{N^\alpha-1}A_1A_1A_1A_2\cdots A_1A_{N^\alpha-1}\cdots A_{N^\alpha-1}A_1A_{N^\alpha-1}A_2\cdots A_{N^\alpha-1}A_{N^\alpha-1}.$$

Therefore, pairs of polygon A_kA_i, $k < i$ and $i = 0, \ldots, N^\alpha-1$, are adjacent. We then use the procedure TANGENTS2(A_iA_k) in Section 5.4.1 to determine $j^*_{i,k}$, $i^*_{i,k}$, $\bar{j}^*_{i,k}$ and $\bar{i}^*_{i,k}$. The number of processors required in the copying is $N^\alpha 2(N^\alpha-1)\cdot N^{1-\alpha} < 2N^{1+\alpha}$, and it can be achieved in $O(\log N)$ parallel steps with some simple-minded algorithm. Determination of the

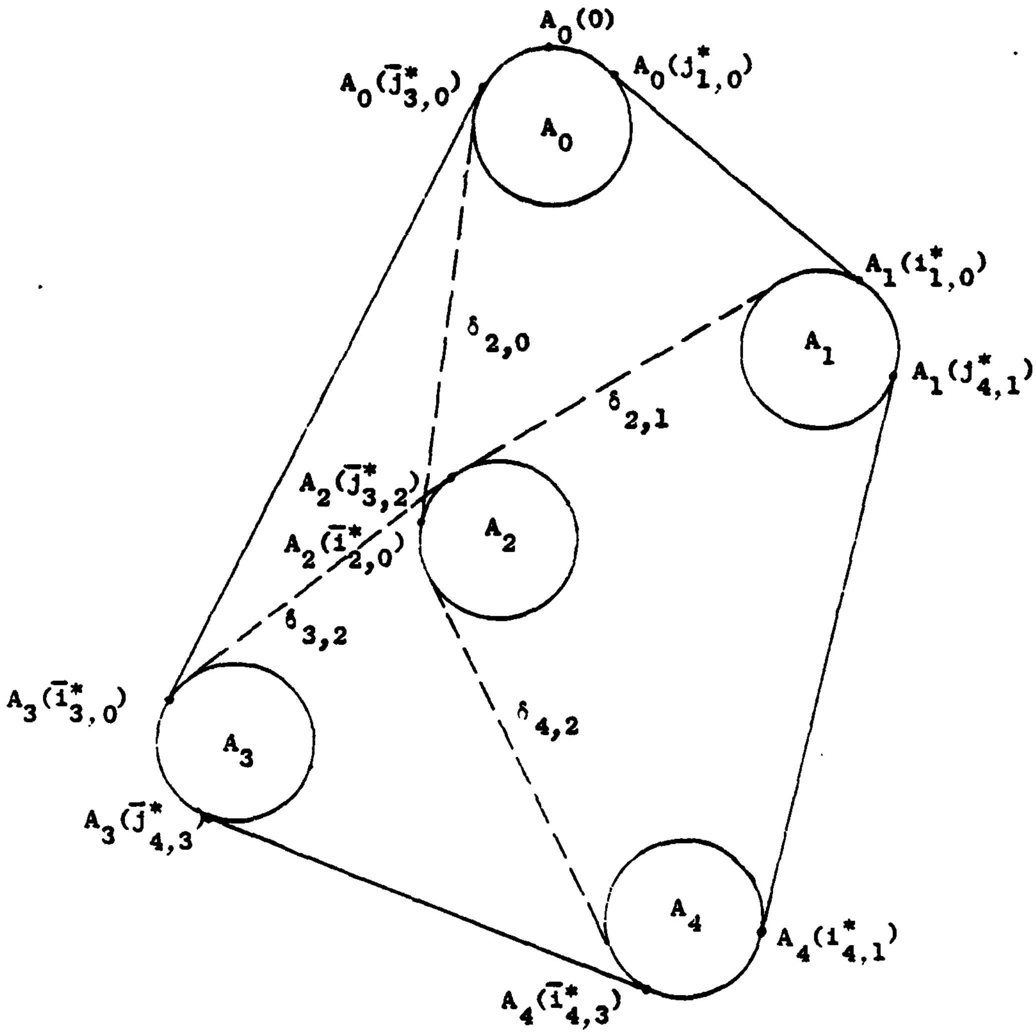


Figure 32. An example of merging five convex polygons.

indices $t(i)$, $b(i)$, $\bar{t}(i)$, and $\bar{b}(i)$ involves finding minimum and maximum of multisets of uniform size; so it can be achieved in $O(\log N)$ steps. Using Theorems 5.5 and 5.6, we can determine whether $A_i(i^*_{i,t(i)})$, $A_i(j^*_{b(i),i})$, $A_i(\bar{i}^*_{i,\bar{t}(i)})$, and $A_i(\bar{j}^*_{\bar{b}(i),i})$ are vertices of the convex hull of $A_0,\ldots,A_{N^\alpha-1}$. Rearranging vertices of the resulting convex hull involves order reversing and data extraction; both can be carried out in time $O(\log N)$. Although the details of this algorithm are a bit tedious to describe, it should be clear that merging N^α convex polygons, each having at most $N^{1-\alpha}$ vertices, can be performed on a CCC with $2N^{1+\alpha}$ processors in time $O(\log N)$.

The entire convex hulls algorithm is a "divide and conquer" program. The subproblems are solved recursively in parallel. Therefore, the running time of this algorithm is $O(\frac{1}{\alpha}\log N)$.

Theorem 5.7. The convex hull of a set of N points in the plane can be determined in time $O(\frac{1}{\alpha}\log N)$ on a CCC with $N^{1+\alpha}$ processors, $0 < \alpha \le 1$.

CHAPTER 6

CONVEX HULLS OF SETS OF POINTS IN THREE DIMENSIONS

The convex hull of a set of points in three dimensions is a convex polyhedron. A convex polyhedron is specified completely by its edges and faces. It is represented by the arrays of edges $E(0: |E| - 1)$ and of faces $F(0: |F| - 1)$. It is a crucial observation that the set of edges of a convex polyhedron forms a planar graph: if we exclude degeneracies, it forms a triangulation. Thus, we know that $|E|$ and $|F|$ are at most $3N-6$ and $2N-4$ respectively, by Euler's polyhedron theorem, where $N(\geq 3)$ is the number of vertices.

In [30], Preparata and Hong show that the convex hull of a set of N points in three dimensions can be determined serially with O(NlogN) operations. Their algorithm uses the "divide and conquer" technique and recursively applies a merge procedure for two nonintersecting convex hulls which consists of two major steps: (1) construction of a "cylindrical" triangulation $\mathcal{T}$, which is tangent to the convex hulls along two circuits; (2) removal from both convex hulls of the respective portions which have been "obscured" by $\mathcal{T}$. In this chapter, this solution is reorganized so that parallel operations are possible.

6.1 Definitions and Preliminaries

We consider a convex polyhedron with edges $E(0: |E|-1)$ and faces $F(0: |F|-1)$. Element $E(i)$ is a record consisting of fields: V_1 and V_2 which are the extremes of this edge; F_1 and F_2 which are indices of the two faces bounded by this edge. Each element $F(i)$ is also a record of three fields: E_1, E_2, and E_3 which are indices of the three bounding edges of $F(i)$.

We can represent face F_i by an equation $\alpha_i x + \beta_i y + \gamma_i z + \delta_i = 0$ with normal vector $\langle a_i, b_i, c_i \rangle$ pointing away from the polyhedron, where

$$a_i = \frac{\alpha_i}{\sqrt{\alpha_i^2 + \beta_i^2 + \gamma_i^2}}, \quad b_i = \frac{\beta_i}{\sqrt{\alpha_i^2 + \beta_i^2 + \gamma_i^2}}, \quad c_i = \frac{\gamma_i}{\sqrt{\alpha_i^2 + \beta_i^2 + \gamma_i^2}}.$$

The convex angle formed by faces F_i and F_j with normal vectors $\langle a_i, b_i, c_i \rangle$ and $\langle a_j, b_j, c_j \rangle$ respectively is $\cos^{-1}\langle a_i, b_i, c_i \rangle \cdot \langle a_j, b_j, c_j \rangle$ which is $\cos^{-1}(a_i a_j + b_i b_j + c_i c_j)$. In the range $0 \le \theta \le \pi$, the function $\cos\theta$ is decreasing from 1 to -1; so the inverse function $\cos^{-1} a$ decreases as a increases. Note that 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 and we conclude this discussion by the following theorem.

Theorem 6.1. The convex angle that face F_i with normal vector $\langle a_i, b_i, c_i \rangle$ forms with face F_j with normal vector $\langle a_j, b_j, c_j \rangle$ decreases as the distance between points (a_i, b_i, c_i) and (a_j, b_j, c_j) decreases.

6.2 Merging Two Convex Polyhedra

Consider two nonintersecting convex polyhedra A and B with edge sets $E_A(0{:}\ |E_A|-1)$ and $E_B(0{:}\ |E_B|-1)$ respectively, and with face sets $F_A(0{:}\ |F_A|-1)$ and $F_B(0{:}\ |F_B|-1)$ respectively. We obtain the convex hull CH(A,B) of A and B in two steps: removal from A and B of the faces which do not belong to CH(A,B) (these faces will be referred to as internal faces); and addition of faces which are tangent to A and B along two circuits (which will be defined later).

6.2.1 Removal of Internal Faces

Consider the half-spaces bounded by $F_A(i)$ of A; we denote the half-space that contains A by H(A,i) and denote the other one that does not contain A by $H(\bar{A},i)$. Face $F_A(i)$ belongs to CH(A,B) if B lies in the half-space H(A,i). Consider the pair of parallel planes of support $PL'_A(i)$ and $PL''_A(i)$, which are parallel to face $F_A(i)$ and bounding the convex polyhedron B. We define the two associated faces $F_B(i')$ and $F_B(i'')$ of $F_A(i)$ as follows: $F_B(i')$ is a face of B making the smallest angle with $PL'_A(i)$ among all the faces of B that intersect at the point of tangency with $PL'_A(i)$; and $F_B(i'')$ is a face of B making the smallest angle with $PL''_A(i)$ among all the faces of B that intersect at the point of tangency with $PL''_A(i)$. Due to convexity, every face of B is in H(A,i) if $F_B(i'')$ and $F_B(i')$ are in H(A,i). We demonstrate what we have just discussed by a two-dimensional analogy in Figure 33. $F_A(i)$ will belong to CH(A,B) because $F_B(i'')$ and $F_B(i')$ are in half-space H(A,i) while $F_A(j)$ will become internal to CH(A,B) because $F_B(j')$ is in $H(\bar{A},j)$.

We now describe how to determine the associated faces of $F_A(i)$. We first transform faces $F_B(0\colon |F_B|-1)$ of B into points $P_B(0\colon |F_B|-1)$ on the surface of the unit sphere, where $P_B(j) = (\bar{a}_j,\bar{b}_j,\bar{c}_j)$ and $\langle \bar{a}_j,\bar{b}_j,\bar{c}_j \rangle$ is the normal vector, pointing away from B, of $F_B(j)$. We search in $P_B(0\colon |F_B|-1)$ for the nearest neighbors $P_B(i'')$ and $P_B(i')$ of (a_i,b_i,c_i) and $(-a_i,-b_i,-c_i)$ respectively, where $\langle a_i,b_i,c_i \rangle$ is the normal vector of $F_A(i)$. By Theorem 6.1, $F_B(i'')$ and $F_B(i')$ are the

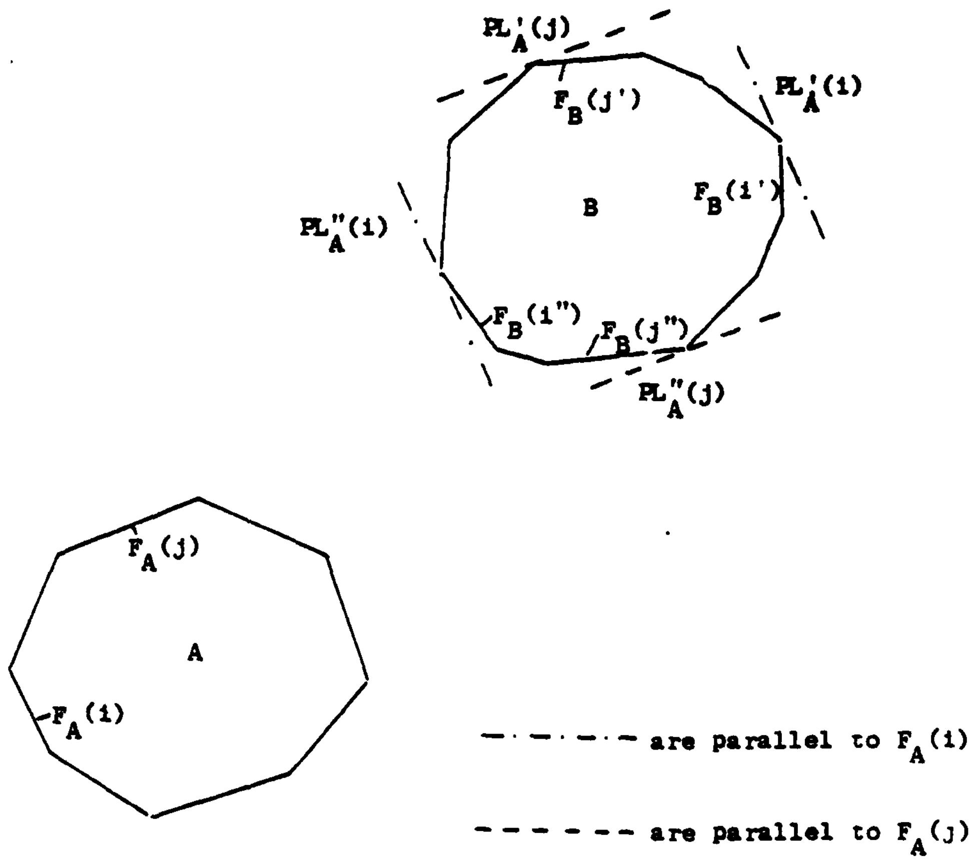


Figure 33. Two-dimensional analogy of associated faces.

associated faces of $F_A(i)$. We shall perform repeatedly nearest neighbor searches for all points $\pm (a_i, b_i, c_i)$ on $P_B(0: |F_B|-1)$; therefore, it is beneficial to arrange $P_B(0: |F_B|-1)$ into an organized structure to facilitate searching. Since $P_B(0: |F_B|-1)$ is on the surface of the unit sphere, we can construct a spherical Voronoi diagram [8] of $P_B(0: |F_B|-1)$. A spherical Voronoi diagram of a set of points P(0: n-1) on a sphere is a partition of the surface of the sphere into n regions: region i for P(i) is the locus of points on the surface of the sphere which are closer to P(i) than to any other point in P(0: n-1). The problem of all nearest neighbors searching is solved by performing point locations in the spherical Voronoi diagram.

In [8], Brown presents an algorithm for constructing the spherical Voronoi diagram of a set of n points P(0: n-1) on the surface of a sphere by intersecting half-spaces. For each point P(i) 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 the n half-spaces H(i) forms a convex body C. The spherical Voronoi diagram is now obtained by a simple projection of the edges of this polyhedron to the surface of the sphere. This projection is a "radial" projection: the projection of a point Q is the point where a line segment connecting the center of the sphere and point Q 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 faces of the polyhedron are mapped to spherical Voronoi regions.

Let $\bar{\alpha}_i x + \bar{\beta}_i y + \bar{\gamma}_i z + \bar{\delta}_i = 0$ be the equation of face $F_B(i)$ with normal vector $\langle \bar{a}_i, \bar{b}_i, \bar{c}_i \rangle$ pointing from B. Then the plane PL(i) tangent to the unit sphere at point $(\bar{a}_i, \bar{b}_i, \bar{c}_i)$ has equation

$\bar{\alpha}_i x + \bar{\beta}_i y + \bar{\gamma}_i z = \sqrt{\bar{\alpha}_i^2 + \bar{\beta}_i^2 + \bar{\gamma}_i^2}$, that is PL(i) is obtained from $F_B(i)$ by a translation. Figure 34 shows the two-dimensional analogy of the translation of faces of B. Therefore, the intersection of PL(i) and PL(j) is an edge of C if and only if $F_B(i)$ and $F_B(j)$ are adjacent.

6.2.2 <u>Addition of New Faces</u>

In addition to the removal of internal faces, we have to construct faces which are tangent to A and B along two circuits C_A and C_B (refer to Figure 35). The circuit C_A is composed of edges $E_A(i)$ of A such that $E_A(i)[F_1]$ is an internal face and $E_A(i)[F_2]$ is not or vice versa. The edges in C_B are determined in the same manner. We have to describe a criterion for uniquely ordering the edges in C_A and C_B. We define observer B as an observer placed at any point of B and oreinted like the negative z-axis; and observer A as an observer placed at any point of A and oriented like the positive z-axis. The edges in C_A are numbered in ascending order so that they form a clockwise sequence for an observer B. And the edges in C_B are numbered in ascending order so that they form a counterclockwise sequence for an observer A. We start both sequences at the vertices with largest y-coordinates in C_A and C_B accordingly. Let $C_A(i)[V_1]$ and $C_B(j)[V_1]$ be the vertices at which edges $C_A(i)$ and $C_B(j)$ originate respectively. Then $(C_A(0)[V_1], C_A(1)[V_1], \ldots)$ and $(C_B(0)[V_1], C_B(1)[V_1], \ldots)$ are the sequences

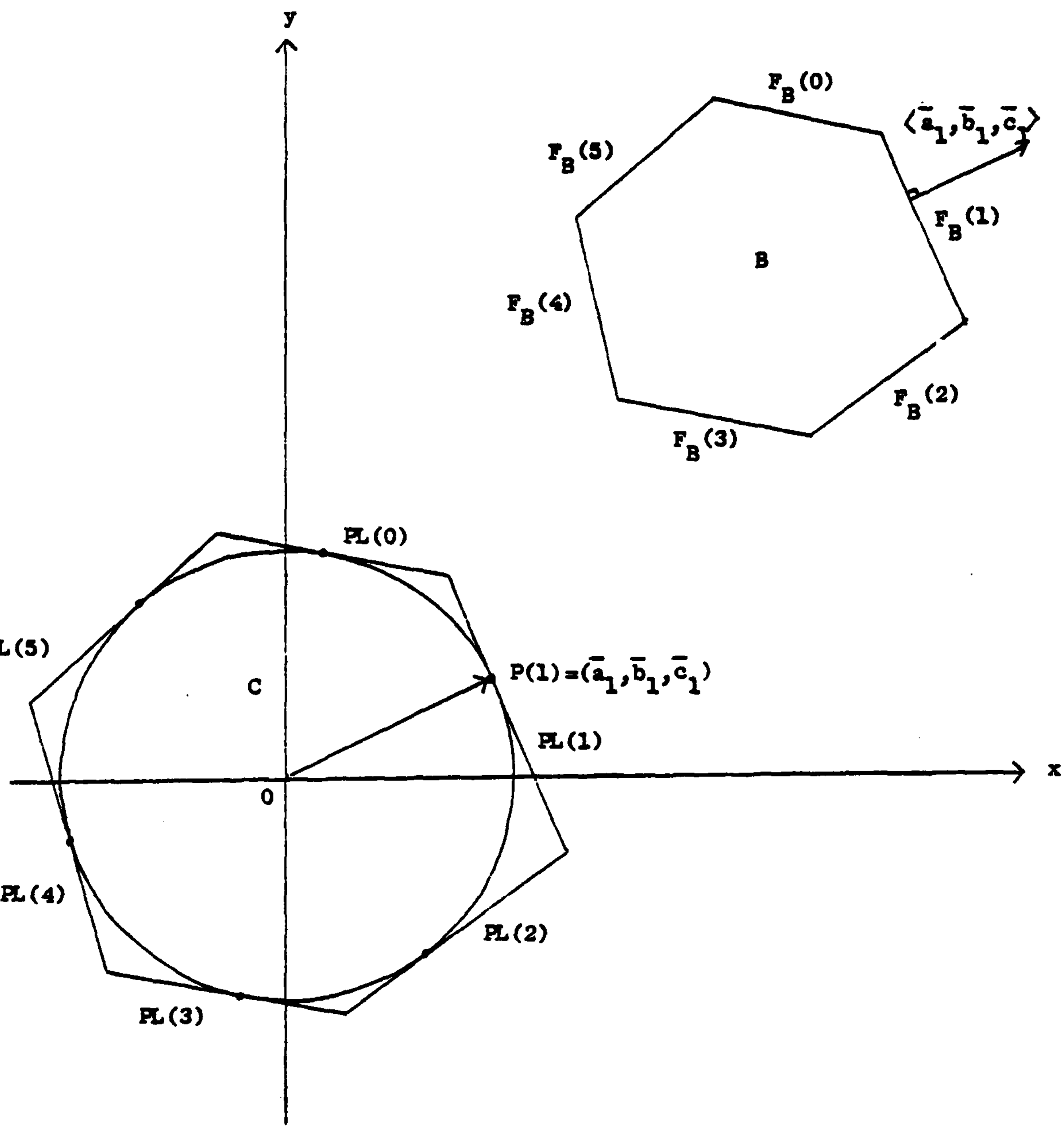


Figure 34. Two-dimensional analogy of the transformation from B to C.

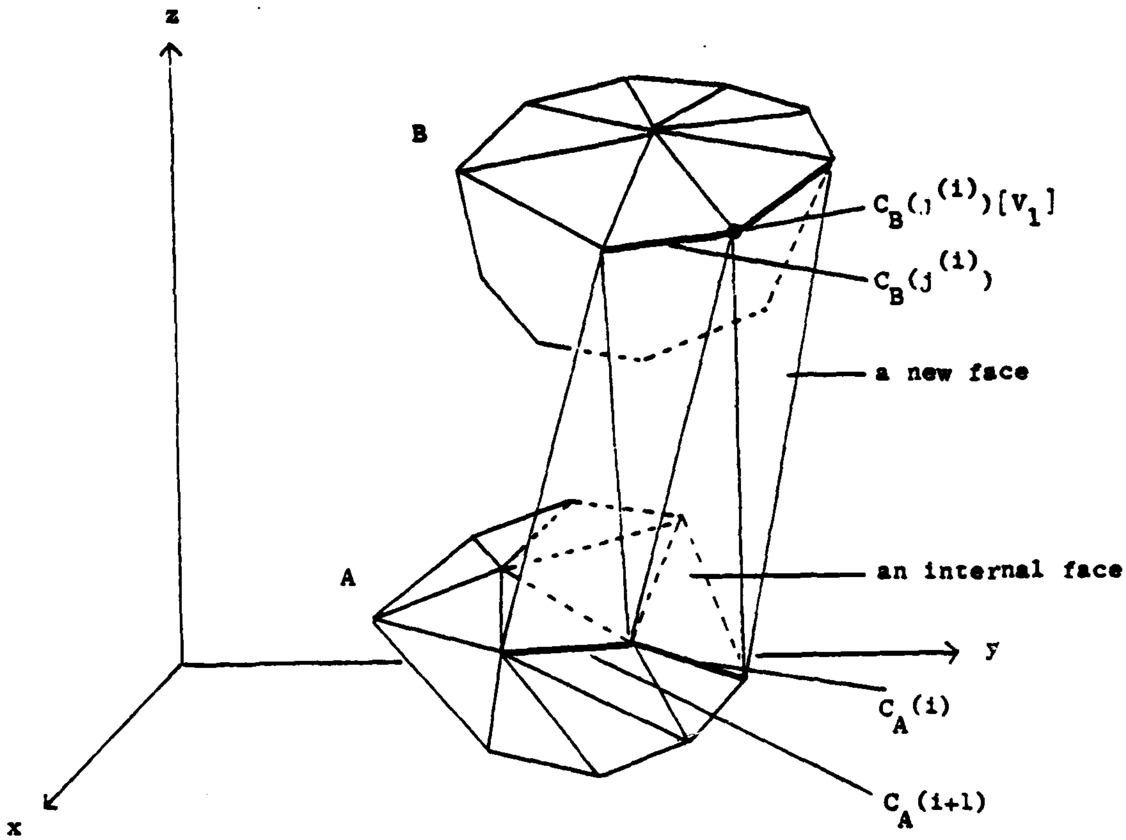


Figure 35. Merging two convex hulls in three dimensions.

of vertices of C_A and C_B respectively. Due to convexity, the convex angle formed by $(C_A(0)[V_1],C_A(i)[V_1])$ and $(C_A(0)[V_1],C_A(j)[V_1])$ is clockwise for an observer B, where $i < j$; the convex angle formed by $(C_B(0)[V_1]],C_B(i)[V_1])$ and $(C_B(0)[V_1],C_B(j)[V_1])$ is counterclockwise for an observer A, where $i < j$. Therefore, edges in C_A can be ordered by some simple sorting algorithm, and so those in C_B.

We define an angle measure $\theta_A(i,j)$,[1] associated with edge $C_A(i)$ and vertex $C_B(j)[V_1]$, as the convex angle formed by the plane determined by $C_A(i)$ and $C_B(j)[V_1]$ and the face bounded by $C_A(i)$, which belongs to CH(A,B). In an analogous manner, we define $\theta_B(j,i)$ as the convex angle formed by the plane determined by $C_B(j)$ and $C_A(i)[V_1]$ and the face bounded by $C_B(j)$, which belongs to CH(A,B). We also define $j^{(i)}$ as the smallest index such that $\theta_A(i,j^{(i)})$ is a maximum among all $\theta_A(i,j)$, $0 \leq j < |C_B|$; $i^{(j)}$ as the largest index such that $\theta_B(j,i^{(i)})$ is a maximum among all $\theta_B(j,i)$, $0 \leq i < |C_A|$. It is observed that $(j^{(0)},j^{(1)},\ldots)$ and $(i^{(0)},i^{(1)},\ldots)$ are nondecreasing sequences. The faces determined by $C_A(i)$ and $C_B(j^{(i)})[V_1]$ (or $C_B(j)$ and $C_A(i^{(j)})[V_1]$) are tangent to A and B. They are faces of CH(A,B).

6.3 On the SMM with N Processors

In this section we discuss the entire convex hulls algorithm in three dimensions on the SMM. The crucial step is the implementation of the merging of two convex polyhedra as described in the previous section. We show that the merging runs in time $O((\log N)^2 \log\log N)$ with N processors, which gives us an $O((\log N)^3 \log\log N)$ three-dimensional convex hulls algorithm on a SMM with N processors.

---

[1] In the actual implementation, the operation of comparing two angles will be replaced by the operation of comparing the negative values of their cotangents.

6.3.1 Implementing the Merge Algorithm

We now present a top-down implementation of the merge algorithm on the SMM. First we have to determine the internal faces. The following procedure determines which faces of the convex polygon A are internal.

<u>procedure</u> INTERNALA(A,B,t_A)

/* Given two nonintersecting convex polyhedra A and B, for each face $F_A(i)$ of A, determines if it is internal to the convex hull of A and B; it sets $t_A(i)$ to 1 if $F_A(i)$ is internal and 0 otherwise */

1. transform each face $F_B(j)$ with normal vector $\langle \bar{a}_j, \bar{b}_j, \bar{c}_j \rangle$ into a point $P_B(j) = (\bar{a}_j, \bar{b}_j, \bar{c}_j)$.
2. construct the spherical Voronoi Diagram G_B for the set P_B.
3. transform each face $F_A(i)$ with normal vector $\langle a_i, b_i, c_i \rangle$ into two points $P''_A(i) = (a_i, b_i, c_i)$ and $P'_A(i) = (-a_i, -b_i, -c_i)$
4. for each i, determine the nearest neighbors $P_B(i'')$ and $P_B(i')$ of the points $P''_A(i)$ and $P'_A(i)$ respectively by point location in G_B.
5. for each i, if both $F_B(i'')$ and $F_B(i')$, the associated faces of $F_A(i)$, are in H(A,i) (i.e., $F_A(i)$ is internal) set $t_A(i)$ to 1; otherwise set $t_A(i)$ to 0.

The transformations in steps 1 and 3 of procedure INTERNALA can be done in constant time with $|F_B|$ and $|F_A|$ processors respectively. As discussed in Section 6.2.1, the construction of the spherical Voronoi Diagram for P_B is just a simple transformation from B, which can be done in constant time. In Section 4.1, we have given a point location algorithm which runs in time $O((\log n)^2 \log\log n)$ on a SMM with max(n,m) processors, where n is the number of vertices in the graph and m is the number to be located. Therefore, all the nearest neighbors in step 4 can be determined in time $O((\log|F_B|)^2\log\log|F_B|)$ with $\max(|F_A|,|F_B|)$ processors. Finally, step 5 runs in constant time. Thus, the internal faces of A are determined in time $O((\log|F_B|)^2\log\log|F_A|)$ on a SMM with $\max(|F_A|,|F_B|)$ processors. Similarly, we can have a procedure INTERNALB(A,B,t_B) which set $t_B(j)$ to 1 if face $F_B(j)$ is internal; and set to 0 otherwise.

Knowing the internal faces, the circuits C_A and C_B as defined in Section 6.2.2, can be determined in time $O(\log|E_A|\log\log|E_A|)$ and $O(\log|E_B|\log\log|E_B|)$ respectively as follows.

```
procedure CIRCUITS(A,B)
    /* determine the two circuits C_A and C_B for A and B  */
    begin
        /* C_A contains edges of A, each of which is shared by an internal
           face and an external face */
        C_A ← E_A
        foreach i, 0 ≤ i < |E_A| do
            if E_A(i)[F_1] is internal and E_A(i)[F_2] is external
                then t(i) ← 1
                else t(i) ← 0
        call EXTRACT1(C_A,t)
        order the edges in C_A as defined in Section 6.2.2

        /* C_B contains edges of B each of which is shared by an internal
           face and an external face  */
        C_B ← E_B
        foreach i, i < |E_B| do
            if E_B(i)[F_1] is internal and E_B(i)[F_2] is external
                then t(i) ← 1
                else t(i) ← 0
        call EXTRACT1(C_B,t)

        order the edges in C_B as defined in Section 6.2.2
    end
```

The face determined by the edge $C_A(i)$ and the vertex $C_B(j^{(i)})[V_1]$ is a new face of the convex hull. Since $j^{(i)}$ is the smallest index such that $\theta_A(i,j^{(i)})$ is a maximum among all $\theta_A(i,j)$, $0 \le j < |C_B|$ and using the result in Section 2.1.2, $j^{(i)}$, for a particular i, can be determined in time $O(\log|C_B|)$ on a SMM. Since $(j^{(0)},j^{(1)},\ldots)$ is a nondecreasing sequence, we can first find $j^{(|C_A|/2)}$; then find, in parallel, $j^{(|C_A|/4)}$ in the intervals $[0,j^{(|C_A|/2)}]$ and $[j^{(|C_A|/2)},|C_B|-1]$ respectively, and so on. It is straightforward to see that it takes $\log|C_A|$ iterations to obtain all $j^{(i)}$'s. We can obtain all $j^{(i)}$'s by invoking the following procedure with a single call FIND_$j^{(i)}$1$(0,|C_A|-1,0,|C_B|-1)$:

```
procedure FIND_j^(i)1(a,b,c,d)
    /* determine j^(i) in the range [c,d] for each i in [a,b]  */
    begin if b-a = 0 then return
        /* determine j^(i) where i is in the middle of [a,b]  */
        i ← (a+b)/2
        j^(i) ← MINIMUM ({j|c ≤ j ≤ d and θ_A(i,j) =
                 MAXIMUM ({θ_A(i,k), c ≤ k ≤ d})})
        /* partition the ranges at i and j^(i), and apply the procedure
           recursively to these sub-ranges  */
        call FIND_j^(i)1(0,i-1,c,j^(i))
        call FIND_j^(i)1(i+1,b,j^(i),d)
    end
```

Similarly, we can have an $O(\log|C_B|\log|C_A|)$ time procedure FIND_$i^{(j)}$1 to produce all $i^{(j)}$'s. We are now about to present the entire merge procedure which runs in time $O((\log N)^2\log\log N)$ with N processors.

```
procedure MERGING1(A,B)
    /* merge A and B, store the resulting convex hull in C */
    begin F_C ← F_A ∪ F_B; E_C ← E_A ∪ E_B
        /* determine internal faces */
        call INTERNALA(A,B,t_A)
        call INTERNALB(A,B,t_B)
        /* determine the new faces formed by C_A(i) and C_B(j^(i))[V_1]
           or C_B(j) and C_A(i^(j))[V_1] */
        call FIND_j^(i)1(0,|C_A|-1,0|C_B|-1)
        call FIND_i^(j)1(0,|C_A|-1,0,|C_B|-1)
        /* remove all internal faces and edges bounding two internal
           faces */
        remove, from F_C, faces with t_A or t_B = 1
        remove, from E_C, edges E_A(i) such that both
            E_A(i)[F_1] and E_A(i)[F_2] have tag t_A = 1,
            and edges E_B(i) such that E_B(i)[F_1] and
            E_B(i)[F_2] have t_B = 1.
        /* add new faces and edges */
        add, to F_C, faces determined by C_A(i) and C_B(j^(i))[V_1]
            and faces determined by C_B(j) and C_A(i^(j))[V_1]
        add, to E_C, edges (C_A(i)[V_1],C_B(j^(i))[V_1]),
            (C_A(i)[V_2],C_B(j^(i))[V_1]),(C_B(j)[V_1],C_A(i^(j))[V_1]),
            and (C_B(j)[V_2],C_A(i^(j))[V_1]) .
    end
```

6.3.2 Three-Dimensional Convex Hulls Algorithm

As a preliminary step, we sort the set S of N points by their y coordinates in ascending order. This can be done in time O(logNloglogN) with N processors. We now present the recursive program for determining the three-dimensional convex hull of S.

```
function CH3(S)
    /* return CH(S) where S is a set of N points in three dimensions */
    begin if N ≤ 2 then return (S)
        else return (MERGING1(CH3(S(0:N/2-1)),CH3(S(N/2:N-1)))
    end
```

The running time $T(N)$ of function CH31 can be obtained from the recurrence relation $T(N) \le T(N/2) + M(N)$, where $M(N)$ is the running time of function MERGING1. In the previous section, we have shown that $M(N)$ is $O((\log N)^2 \log\log N)$ with N processors, thence, $T(N) = O((\log N)^3)\log\log N)$.

Theorem 6.1. The convex hull of a set of N points in the three dimensional space can be determined in time $O((\log N)^3 \log\log N)$ on a SMM with N processors and N memory units.

6.4 On the CCC with N Processors

The main purpose of this section is to discuss the implementation of the merge algorithm on a CCC. We shall first develop a parallel algorithm for finding the maxima of several sets of numbers. This will be used in the implementation.

6.4.1 Finding Maxima of Multiple Sets

Given an array $D(0: n-1)$ of numbers, which is partitioned into m subarrays $D_0, D_1, \ldots, D_{m-1}$ such that the concatenation $D_0 \cdot D_1 \cdot \ldots \cdot D_{m-1} = D(0: n-1)$, we want to find the maximum of each D_i. We assume n is a power of 2. We logarithmically partition each D_i into at most 2 logn-1 segments by means of a segment tree $T(0,n)$ [28], which consists of a root V representing an integer interval $[0,n]$, and of a left subtree $T(0,\lfloor n/2 \rfloor)$ and a right subtree $T(\lfloor n/2 \rfloor + 1,n)$ (refer to Section 4.1 for more details). For example, $D_i = \{D(7),D(8),\ldots,D(13)\}$, a subarray of $D(0: 31)$, is partitioned into $\{\{D(7)\},\{D(8),\ldots,D(11)\},\{D(12),D(13)\}\}$. We first find the maximum of each of these segments (to be referred to as submaxima). We then find the maximum $M(i)$ among the submaxima of the same array D_i.

We now outline the procedure that determines the maxima $M(j)$ of D_j for $0 \le j < m$ (we shall present the program in the appendix).

1. Logarithmically segment each subarray by means of a segment tree $T(0,n)$.
2. Determine the maxima of the segments by an ASCEND program: at iteration k, $k = 0,\dots,\log n-1$, if $D(i)$ and $D(i+(1-BIT_k(i))2^k)$ belong to the same segment, change $D(i)$ to the larger of the two; at the end of logn iterations, every position of a segment contains the maximum of that segment.
3. Extract the submaxima obtained in step 2.
4. Determine the maxima of the sets of submaxima of same subarray.

As discussed in the planar point location algorithm, the intervals can be logarithmically segmented in time O(logn) on a CCC with n processors. Step 2 is an ASCEND program which runs in logn steps. Data extraction discussed in Section 2.2.1 runs in time O(logn) on a CCC with n processors. Since each subarray is segmented into at most 2logn-1 segments, there are at most 2logn-1 submaxima in each subarray. Therefore, the maxima of the of the same subarray can be determined in time O(logn).

<u>Theorem 6.2</u>. The maxima of each of subarrays $D_0, D_1, \dots, D_{m-1}$ of D where the concatenation $D_0 \cdot D_1 \cdot \dots \cdot D_{m-1}$ is the array D of n elements, can be found in time O(logn) on a CCC with n processors.

6.4.2 Implementing the Merge Algorithm

We now discuss how the merge algorithm can be implemented on a CCC with N processors in time $O((\log N)^3)$. The procedures INTERNALA and INTERNALB in Section 6.3.1 for determining the internal faces of polyhedra A and B can be implemented on a CCC with N processors. The most time-consuming step is determining all nearest neighbors which involves the point location algorithm in Section 4.2. With the result in Section 4.2, the internal faces can be determined in time $O((\log N)^3)$.

We have to modify slightly the procedure CIRCUITS in Section 6.3.1, for determining the two circuits C_A and C_B, so that it can be implemented on a CCC. We have to use procedure EXTRACT2 in Section 2.2.1 for data extraction and the ordering takes $O((\log N)^2)$ time on a CCC. Therefore, the circuits are determined in time $O((\log N)^2)$ on a CCC with N processors.

In implementing the procedure for finding the $j^{(i)}$ and $i^{(j)}$ for the circuits, we have to use the algorithm in the previous section for finding the maximums of multiple sets on a CCC. Therefore, $j^{(i)}$ and $i^{(i)}$ can be determined in time $O((\log N)^2)$ on a CCC with N processors.

The steps in the procedure MERGING1 (Section 6.3.1) can be modified according to the above discussion and be implemented on a CCC with N processors in time $O((\log N)^3)$. Using the same recursive program CH3 in Section 6.3.2 with this modified merge procedure, we have an $O((\log N)^4)$ time algorithm for determining the three-dimensional convex hull.

Theorem 6.3. The convex hull of a set of N points in the three-dimensional space can be determined in time $O((\log N)^4)$ on a CCC with N processors.

6.5 On the CCC with $N^{1+\alpha}$ Processors

In the process of merging two convex hulls, the point location used in determining all nearest neighbors is the most time-consuming step. It can be done in time $O(\frac{1}{\alpha}(\log N)^2)$ on a CCC with $N^{1+\alpha}$ processors (refer to Section 4.3), where $0 < \alpha \leq 1$. Therefore, we have a $O(\frac{1}{\alpha}(\log N)^2)$ time merging algorithm which yields an $O(\frac{1}{\alpha}(\log N)^3)$ time algorithm for finding the three-dimensional convex hull.

Theorem 6.4. The convex hull of a set of N points in the three-dimensional space can be determined in time $O(\frac{1}{\alpha}(\log N)^3)$ on a CCC with $N^{1+\alpha}$ processors, where $0 < \alpha \leq 1$.

CHAPTER 7

VORONOI DIAGRAMS FOR POINTS IN THE EUCLIDEAN PLANE

A Voronoi diagram of a set S(0: N-1) of N points in the Euclidean plane is a partition of the plane into N convex polygonal regions R(0: N-1) (refer to Figure 36). For each point S(i), the convex polygonal region R(i) is the locus of points closer to S(i) than the other N-1 points of S. The vertices of the diagram are called Voronoi points; and the line segments are Voronoi edges. The polygonal boundaries of the regions are called Voronoi polygons.

The problem of the construction of planar Voronoi diagrams arises in many areas; one of the most important applications is in nearest neighbor problems. Shamos and Hoey [35] present an O(NlogN) "divide and conquer" algorithm for construction of a planar Voronoi diagram. Brown [8] describes an O(NlogN) time algorithm which can be extended to higher dimensions. His result is that a two-dimensional Voronoi diagram of N points can be constructed by transforming the points to three-dimensional space, constructing the convex hull of the transformed points, and then transforming back to two-dimensional space.

In this chapter we use Brown's technique to develop parallel algorithms for constructing planar Voronoi diagrams on the SMM and on the CCC.

7.1 Definitions and Preliminaries

In this section we describe how to represent a Voronoi diagram, review some important properties of the Voronoi diagram, and define the inversion transform which will be used in the construction of the Voronoi diagram.

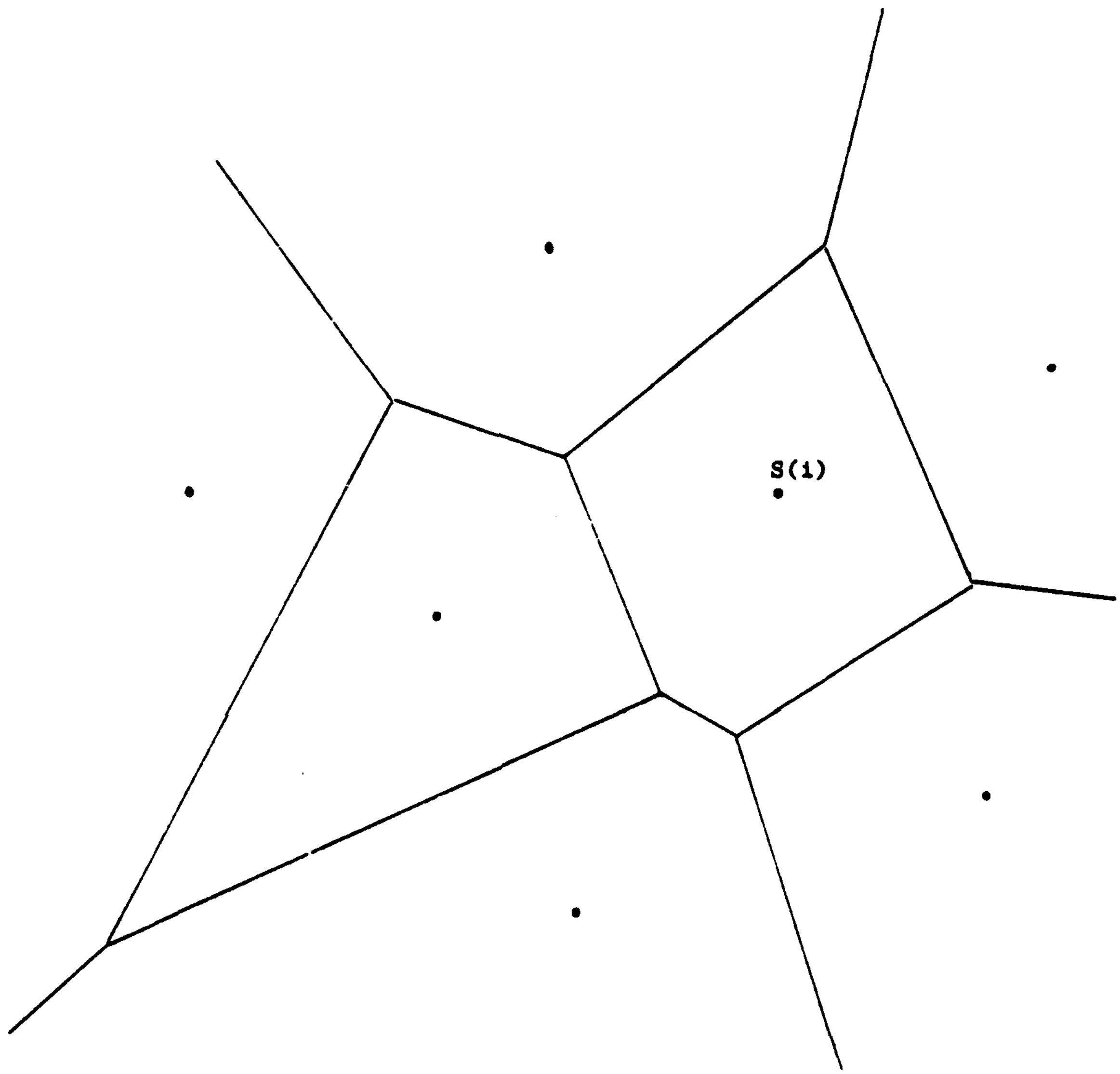


Figure 36. The Voronoi diagram of points in the plane.

7.1.1 Representation of Voronoi Diagrams

Let V(0: |V|-1) and E(0: |E|-1) be the sets of Voronoi points and of Voronoi edges, respectively, of the Voronoi diagram of S(0: N-1), where $|V| \leq 2N-4$ and $|E| \leq 3N-6$. Each element V(i) contains the following information: V(i)[x], V(i)[y] which are the coordinates of the Voronoi points V(i), and V(i)[ADJ], the adjacency list of V(i). Elements E(i) contains the two original points that determine Voronoi edge E(i). By constructing the Voronoi diagram, we also mean obtaining the set of Voronoi polygons in standard form; $P_i(0: |P_i|-1)$ is the Voronoi polygon relative to point S(i).

7.1.2 Properties of Voronoi Diagrams

We now review some important properties of Voronoi diagrams which are exploited in the algorithm of Brown. Each Voronoi point V(i) of the Voronoi diagram for S is equidistant from the three points of S which are closest to V(i). The circle determined by these three points is centered at V(i) and contains no other points of S. Furthermore, if the circle determined by any three points of S does not contain any other points of S (these three points are said to be satisfying the circumcircle property), then the center of the circle is a Voronoi point. A Voronoi edge is the perpendicular bisector of the line segment joining two points of S, which are on the same circumcircle.

7.1.3 The Inversion Transform

The geometric transform used by the algorithm is called inversion. The inversion is an involutory point-point transformation determined by two parameters, the center of inversion P_0 and the radius of inversion r. The image of a point Q under the inversion is another point Q', where $\vec{P_0Q}$ and $\vec{P_0Q'}$ are in the same direction and the magnitude $|\vec{P_0Q'}| = r^2/|\vec{P_0Q}|$. For example, that the center of inversion is the origin and that the radius of inversion is one, then under this inversion, in the plane, the image of a point with polar coordinates (R,θ) is $(1/R,\theta)$; and in the space, the image of (R,θ,ϕ) is $(1/R,\theta,\phi)$. The inversion transforms any sphere which passes through the center of inversion to a plane which does not pass through the center of inversion, and vice versa. For example with the center of inversion at a point P_0 not on the xy-plane and radius > 0, the xy-plane transforms to a sphere with P_0 at the apex. Another property of inversion is that the interior of the sphere transforms to a half-space bounded by the plane which is the image of the sphere, and the exterior of the sphere transforms to the other half-space.

7.2 The Voronoi Diagram Algorithm

In this section, we shall describe how the techniques of embedding into three dimensions, inversion, and the three-dimensional convex hull algorithm are used to construct the Voronoi diagram of a set S of points in the xy-plane.

Let S' be the set of inversion points of S with center at an arbitrary point P_0 not in the xy-plane and radius 1. Since all points of the xy-plane are mapped to a sphere with P_0 at the apex, all points of S' are on this sphere and they will be on the convex hull of S'. Observe that any three points of S satisfying the circumcircle property determine a face F of the convex hull. This happens because the other N-3 points of S are exterior to the circle determined by these three points, that is, exterior to the sphere with P_0 at the apex and intersecting the xy-plane in that circle (refer to Figure 37). Therefore, after the inversion, the other N-3 points will be in the same half-space bounded by the plane F. Therefore, we can find the Voronoi points as follows: we invert each face F_i of the convex hull of S' into the corresponding sphere, which will intersect the xy-plane in a circle. The center V_i of this circle is a Voronoi point if P_0 and the convex hull are in the same half-space whose boundary plane contains face F_i.

The Voronoi edges are constructed by connecting appropriate pairs of Voronoi points. Suppose faces F_i and F_j of the convex hull meet at an edge of the hull, then there will be a Voronoi edge from V_i to V_j when both V_i and V_j are Voronoi points. However, if one and only one of V_i and V_j, say V_i, is a Voronoi point, then there will be an infinite ray starting at V_i in the direction of $\overrightarrow{V_iV_j}$ (unbounded Voronoi polygon).

We now present the entire Voronoi diagram algorithm as follows:

```
procedure CONSTRUCT_VD(S)

    /* construct the Voronoi diagram of a set S(0: N-1) of points
       in the xy-plane  */
    begin

         /* embed each point (x,y) of S into (x,y,0)  */
1.       foreach i, 0 ≤ i < N do begin
                                       S*(i)[x] ← S(i)[x]
                                       S*(i)[y] ← S(i)[y]
                                       S*(i)[z] ← 0
                                 end
```

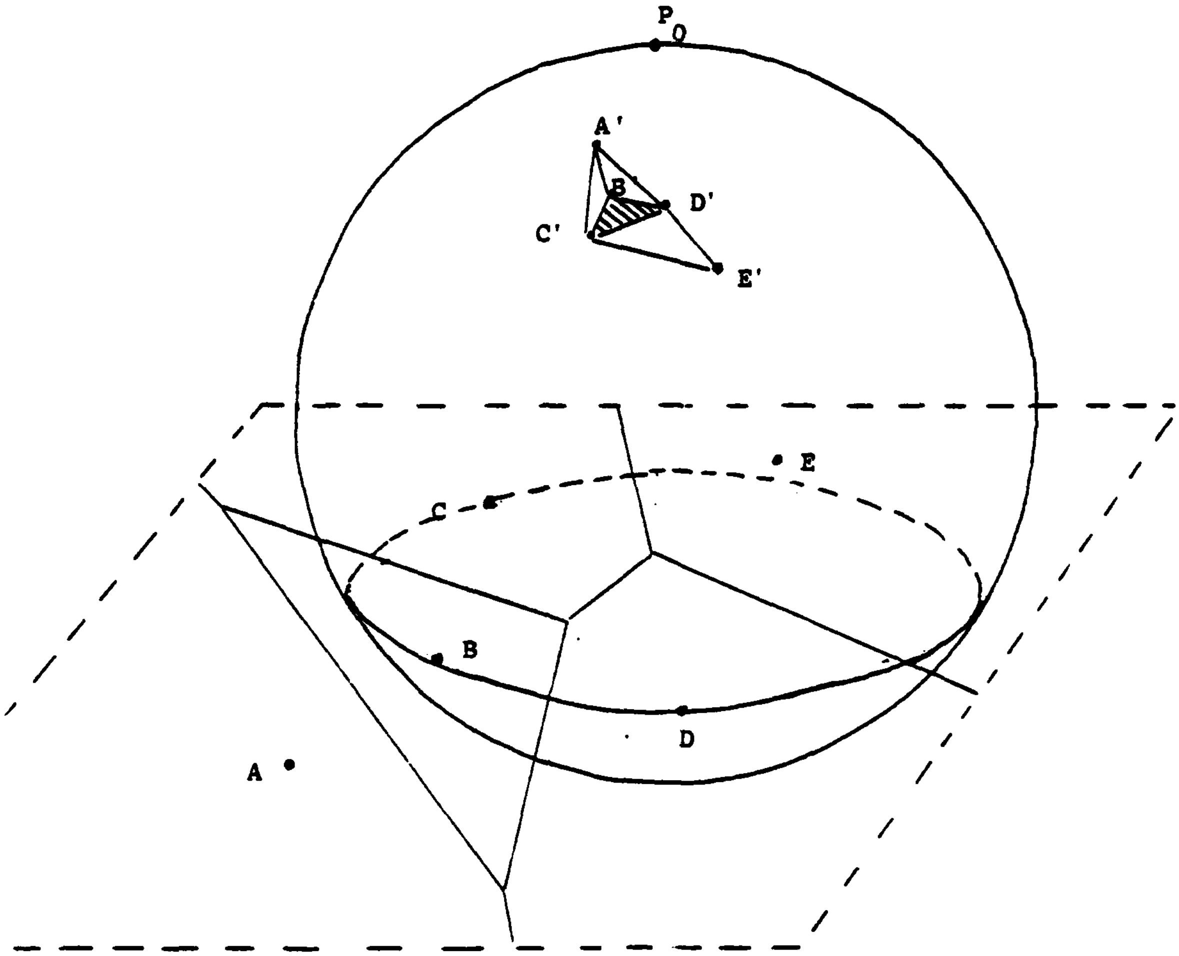


Figure 37. Planar Voronoi diagram and corresponding convex hull.

```
        /* choose the center and radius of inversion */
        P_0 ← some arbitrary point not on the xy-plane
        r ← 1

        /* invert points in S* w.r.t. P_0 and r */
2.      foreach i, 0 ≤ i < N do S'(i) ← inversion of S*(i) w.r.t.

3.      construct the convex hull CH of S'

        /* determine the Voronoi points */
4.      foreach face F_i of CH do
            begin A_i ← inversion of F_i
                V_i ← center of the circle which is the intersection of
                    A_i and the xy-plane.
                if P_0 and CH are in the same half-space bounded by F_i
                    then V_i is a Voronoi point
            end

        /* determine Voronoi edges and rays */
5.      foreach each edge E_ij, bounding F_i and F_j of CH do
            if V_i is a Voronoi point
                then if V_j is a Voronoi point
                        then (V_i,V_j) is a Voronoi edge
                        else there is a ray starting at V_i
                            in the direction of →V_iV_j
                else if V_j is a Voronoi point
                        then there is a ray starting at V_j
                            in the direction of →V_jV_i

6.      obtain the set of Voronoi polygons.
    end
```

We shall show, in the next section, that this algorithm can be implemented on a SMM and a CCC.

7.3 Implementing the Voronoi Diagram Algorithm on the SMM and the CCC

We first show that the algorithm in Section 7.2 can be implemented on a SMM with N processors and N memory units in time $O((\log N)^3 \log\log N)$. The embedding into three dimensions is clearly achievable in constant time with N processors and N memory units. Each independent inversion transform can be done in constant time on one processor. Therefore, steps 1, 2 and 5 of the algorithm run in constant time. It is not difficult to show that step 4 also runs in constant time. The most time-consuming step is step 5 of the algorithm which requires the construction of the convex hull. We have shown in Section 6.3 that the three-dimensional convex hull can be constructed on a SMM with N processors and N memory units in time $O((\log N)^3 \log\log N)$. The final step which obtains all the Voronoi polygon involves grouping and sorting the edges. This can be done in time $O(\log N \log\log N)$. Therefore, we have the following result.

Theorem 7.1. The Voronoi diagram of a set of N points in the plane can be constructed in time $O((\log N^3 \log\log N)$ on a SMM with N processors and N memory units.

As we discussed in the previous paragraph, the construction of the convex hull in three dimensions is the most time-consuming step of the algorithm. In Sections 6.4 and 6.5, we have presented an $O((\log N)^4)$ and an $O(\frac{1}{\alpha}(\log N)^3)$ three-dimensional convex hull algorithms for the CCC with N processors and $N^{1+\alpha}$ processors, respectively. And it is straightforward to show that all other steps of the algorithm require at most $O((\log N)^2)$ for N processors and $O(\frac{1}{\alpha} \log N)$ for $N^{1+\alpha}$ processors. Therefore, we have the following results.

Theorem 7.2. The Voronoi diagram of a set of N points in the plane can be constructed in time $O((\log N)^4)$ on a CCC with N processors.

Theorem 7.3. The Voronoi diagram of a set of N points in the plane can be constructed in time $O(\frac{1}{\alpha}(\log N)^3)$ on a CCC with $N^{1+\alpha}$ processors, where $0 < \alpha \leq 1$.

CHAPTER 8

CONCLUSION

It has been demonstrated in this thesis that in solving certain geometric problems, operations can be performed in parallel to substantially reduce the computation time. Using the Shared Memory Machine of Section 1.1.1, parallel algorithms have been developed to solve the problems of reporting all intersecting pairs of rectangles in time $O((\log N)^2)$, planar points location in time $O((\log N)^2 \log\log N)$, constructing two-dimensional convex hulls in time $O((\log N)^2)$, three-dimensional convex hulls in time $O((\log N)^3 \log\log N)$, and constructing planar Voronoi diagram in time $O((\log N)^3 \log\log N)$. Using the Cube-Connected-Cycles with a number of processors linear in problem size, the parallel algorithms developed for all of these problems, except reporting intersecting pairs of rectangles and constructing two-dimensional convex hull, have time complexity only increased by a factor of $\log N/\log\log N$. The algorithms for the two exceptional problems have time complexity $O((\log N)^2)$ which is the same as that on the SMM. With an increase in the number of processors of the CCC to $N^{1+\alpha}$ $(0 < \alpha \leq 1)$, all of the problems can be solved with parallel algorithms of time complexity improved by a factor of $1/(\alpha \log N)$ with respect to the time complexity of the algorithms on the CCC with N processors. In contrast, the best sequential algorithms for all of these problems, except planar point location, have a worst case time complexity of $O(N\log N)$. The best sequential algorithms for locating M points in a graph of N vertices has time complexity $O((M+N)\log N)$.

In parallel computation, it is possible that some processors are not always busy. It has been shown that the algorithms presented here for finding the two-dimensional convex hulls and reporting intersecting pairs of rectangles are not only fast, but involve relatively little waste as well.

The results in this thesis indicate that geometric problems are susceptible of being solved efficiently on parallel computer systems. Moreover, once again, the Cube-Connected-Cycles is shown to be suitable for implementing algorithms for an expanding class of problems.

We conclude this thesis by presenting the results in Table 1.

problems \ time complexity \ models of computation	Uniprocessor	SMM N processors	CCC N processors	CCC $N^{1+\alpha}$ processors $0<\alpha\leq 1$
intersections of N rectangles	$O(N\log N+k)$[1]	$O((\log N)^2+k')$[2]	$O((\log N)^2+k')$	$O(\frac{1}{\alpha}\log N+k')$
locating M points in a planar graph with N vertices	$O((M+N)\log N)$	$O((\log N)^2\log\log N)$	$O((\log(M+N)^3)$	$O(\frac{1}{\alpha}\log(N+M))$
convex hull of N points in the plane	$O(N\log N)$	$O((\log N)^2)$	$O(\log N)^2)$	$O(\frac{1}{\alpha}\log N)$
convex hull of N points in the space	$O(N\log N)$	$O((\log N)^3\log\log N)$	$O((\log N)^4)$	$O(\frac{1}{\alpha}(\log N)^3)$
Voronoi diagram for N points in the plane	$O(N\log N)$	$O((\log N)^3\log\log N)$	$O((\log N)^4)$	$O(\frac{1}{\alpha}(\log N)^3)$

Table 1.

(1) k is the number of intersecting pairs.

(2) k' is the maximum number of intersections per rectangle.

REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

2. Akl, S. G. and Toussaint, G. T., "A fast convex hull algorithm," Information Processing Letters, vol. 7, no. 5, August 1978, pp. 219-222.

3. Arjomandi, E., "A study of parallelism in graph theory," Ph.D. thesis, Department of Computer Science, University of Toronto, December 1975.

4. Baird, H. S., "Fast algorithms for LSI artwork analysis," Design Automation & Fault-Tolerant Computing, 1978, pp. 179-209.

5. Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. K., and Stoker, R. A., "The Illiac IV computer," IEEE Trans. on Computers, vol. C-17, 1968, pp. 746-757.

6. Bentley, J. L. and Shamos, M. I., "Divide-and-conquer in multidimensional space," Proc. 8th ACM Symp. on Theory of Computing, May 1976, pp. 220-230.

7. Bentley, J. L. and Wood, D., "An optimal worst-case algorithm for reporting intersection of rectangles," Computer Science Technical Report, McMaster University, 1979.

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

9. Bykat, A., "Convex hull of a finite set of points in two dimensions," Information Processing Letters, vol. 7, no. 6, October 1978, pp. 296-298.

10. Cray Research, Inc., "Cray-1 computer," Chippewa Falls, Wisconsin, 1975.

11. Csanky, L., "Fast parallel matrix inversion algorithms," SIAM J. Computing, vol. 5, No. 4, December 1976, pp. 618-623.

12. Eckstein, D., "Parallel graph processing using depth-first search and breadth-first search," Ph.D. thesis, Department of Computer Science, University of Iowa, Iowa City, 1977.

13. Hartigan, J. A., Clustering Algorithms, John Wiley & Sons, New York, 1975.

14. Heller, D., "A determinant theorem with applications to parallel algorithms," Department of Computer Science Tech. Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1973.

15. Hintz, R. G. and Tate, D. P., "Control data STAR-100 processor deisgn," COMPCON-72 Digest of Papers, IEEE Comp. Soc., 1972, pp. 1-4.

16. Hirschberg, D. S., "Fast parallel sorting algorithms," CACM, vol. 21, no. 8, August 1978, pp. 657-661.

17. Hirschberg, D. S., "Parallel algorithms for the transitive closure and the connected component problems," Proc. 8th ACM Symp. on Theory of Computing, May 1976, pp. 55-57.

18. Kuck, D. J., The Structure of Computers and Computations, Department of Computer Science, University of Illinois, Urbana.

19. Lauther, U., "4-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits," Design Automation & Fault-Tolerant Computing, 1978, pp. 241-247.

20. Lee, D. T., "Proximity and reachability in the plane," Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, November 1978.

21. Lee, D. T. and Preparata, F. P., "Location of a point in a planar subdivision and its applications," SIAM J. Computing, vol. 6, 1977, pp. 594-606.

22. Mead, A. M. and Convay, L. A., Introduction to VLSI Systems, 1978, Textbook in preparation.

23. Muller, D. E. and Preparata, F. P., "Restructuring of arithmetic expressions for parallel evaluation," J. ACM, vol. 23, no. 3, July 1976, pp. 534-543.

24. Munro, I. and Paterson, M., "Optimal algorithms for parallel polynomial evaluation," J. of Computer and System Sciences, vol. 7, no. 2, 1973.

25. Muraoka, Y., "Parallelism exposure and exploitation in programs," Department of Computer Science Tech. Report No. 424, University of Illinois, Urbana, 1971.

26. Nassimi, D. and Sahni, S., "Parallel permutation and sorting algorithms and a new generalized-connection-network," Computer Science Department, Tech. Report 79-8, University of Minnesota, Minneapolis, April 1979.

27. Preparata, F. P., ed., Steps into Computational Geometry, Coordinated Science Laboratory Report R-760, University of Illinois, Urbana, March 1977.

28. Preparata, F. P., "A new approach to planar point location," submitted for publication, 1979.

29. Preparata, F. P., "New parallel sorting schemes," IEEE Trans. on Computers, vol. C-27, no. 7, July 1978, pp. 669-673.

30. Preparata, F. P. and Hong, S. J., "Convex hulls of finite sets of points in two and three dimensions," CACM, vol. 20, no. 2, February 1977, pp. 87-93.

31. Preparata, F. P. and Vuillemin, J., "The cube-connected-cycles: A versatile network of parallel computation," Proc. 20th Annual IEEE Symp. on Foundations of Computer Science, October 1979.

32. Rudolph, J. A., "A production implementation of an associative array processor-staran," AFIPS Fall 1972, AFIPS Press, Montvale, N. J., vol. 41, pt. 1, pp. 229-241.

33. Savage, C. D., "Parallel algorithms for graph theoretic problems," Ph.D. thesis, Department of Mathematics, University of Illinois, Urbana, August 1978.

34. Shamos, M. I., Computational Geometry, Department of Computer Science, Yale University, 1977, to be published by Springer-Verlag.

35. Shamos, M. I. and Hoey, D., "Closest-point problems," Proc. 16th Annual IEEE Symp. on Foundations on Computer Science, October 1975, pp. 151-162.

36. Stone, H. S., "An efficient parallel algorithm for the solution of a tridiagonal system of equations," J. ACM, vol. 20, no. 1, 1973, pp. 27-38.

37. Valiant, L. G., "Parallelism in comparison problems," SIAM J. on Computing, vol. 4, no. 3, September 1975, pp. 348-355.

38. Wulf, W. A. and Bell, C. G., "C. mmp, a multi-mini-processor," AFIPS Fall 1972, AFIPS Press, Montval, N. J., vol. 41, pt. 2, pp. 765-777.

APPENDIX

```
procedure CONSTRUCT_𝓕(S)
    /* determine F_logN,...,F_0 for the points in S */
    begin
        /* the root F_logN is the set S sorted by their x-values */
        F_logN ← S
        sort F_logN by their x-values
        foreach j, 0 ≤ j < n do N#_logN(j) ← 0
        /* determine F_logN-1,...,F_0 one at a time */
        for i ← logN downto 1 do
            begin
                /* determine the node numbers N#_i-1 in the next level
                   i-1 for each point */
                foreach j, 0 ≤ j < n do
                    begin F_i-1(j) ← F_i(j)
                        TEMP(j) ← F_i(j)
                        N#_i-1(j) ← N#_i(j)
                        t_1(j) ← t_2(j) ← 0
                        if y-value of F_i(j) ≤ B_i-1(N#_i(j))
                            then t_1(j) ← 1
                            else begin t_2(j) ← 1
                                     TEMPN#(j) ← N#_i(j) + 2^(logN-i)
                                 end
                    end
                /* rearrange the points according to their node number */
                call EXTRACT2(F_i-1,t_1); call EXTRACT2(N#_i-1,t_1)
                call EXTRACT2(TEMP,t_2); call EXTRACT2(TEMPN#,t_2)
                foreach j, 0 ≤ j < |TEMP| do
                    begin F_i-1(j + |F_i-1|) ← TEMP(j)
                        N#_i-1(j) + |F_i-1|) ← TEMPN#(j)
                    end
            end
    end
```

```
procedure INTERSECT3(V,H):

    /* search all intersecting pairs of horizontal line segments in H and
       vertical line segments in V */
    begin

        /* construct the search structures D_{1/α},D_{1/α-1},...,D_0 for V */
        call CONSTRUCT_β1(V)

        /* H', the set of horizontal line segments, is maintained sorted
           lexicographically by their node numbers and the x-values of
           their left endpoints. */
        H' ← H
        sort H' by x-values of left endpoints
        foreach j, 0 ≤ j < m do NN(j) ← 0
        foreach j, m ≤ j < 2mN^α do H'(j) ← null

        /* search in β beginning at D_{1/α} */
        for i ← 1/α downto 0 do
            begin call RANGE_SEARCH_1D(d_i,H')

            /* determine node numbers of the horizontal line segments
               and reorder H' according to these node numbers */
            for k ← log 2m to log 2mN^α-1 do /* duplicate H' N^α times */
                if BIT_k(j) = 0 then begin H'(j+2^k) ← H'(j)
                                           NN(j+2^k) ← NN(j)
                                     end
            foreach j, 0 ≤ j < 2mN^α do /* determine node numbers */
                begin t(j) ← 0
                      NN(j) ← NN(j) + ⌊j/2m⌋N^{1-iα}
                      if B_{i-1}(NN(j)) ≤ y-value of H'(j) ≤ T_{i-1}(NN(j))
                          then t(j) ← 1
                end
            call EXTRACT2(H',t); call EXTRACT2(NN,t) /* reordering */
        end
    end
```

```
procedure CONSTRUCT_𝒥(S)
    /* construct the arrays G_{1/α},...,G_0 for the set S of points */
    begin
        /* the root, G_{1/α}, is the set S sorted by x-coordinates */
        G_{1/α} ← S
        sort G_{1/α} by their x-values
        foreach j, 0 ≤ j < n do N#_{1/α}(j) ← 0
        /* determine G_{1/α-1},...,G_0 one by one in descending order */
        for i ← 1/α downto 1 do
            begin
                /* G_{i-1} is obtained by reorder G_i as follows */
                foreach j, 0 ≤ j < nN^α do
                    begin G_{i-1}(j) ← N#_i(j)
                          N#_{i-1}(j) ← N#_i(j)
                          t(j) ← 0
                    end
                /* duplicate G_i into N^α copies */
                for k ← logn to lognN^α-1 do
                    if BIT_k(j) = 0 then begin G_{i-1}(j + 2^k) ← G_{i-1}(j)
                                               N#_{i-1}(j + 2^k) ← N#_{i-1}(j)
                                         end
                /* determine node numbers of each point in G_{i-1} */
                foreach j, 0 ≤ j < nN^α do
                    begin N#_{i-1}(j) ← N#_{i-1}(j) + ⌊j/n⌋N^{1-iα}
                          if B_{i-1}(N#_{i-1}(j)) ≤ y-value of G_{i-1}(j) ≤
                                                           T_{i-1}(N#_{i-1}(j))
                                then t(j) ← 1
                    end
                /* reorder the points according to their node numbers
                   and x-coordinates */
                call EXTRACT2(G_{i-1},t)
                call EXTRACT2(N#_{i-1},t)
            end
    end
```

```
procedure RANGE_SEARCH3(S,Q)

    /* report all points a ∈ S such that Q(i)[L] ≤ x(a) ≤ Q(i)[R]
       and Q(i)[B] ≤ y(a) ≤ Q(i)[T] for every Q(i) */
    begin

        /* construct the search arrays 𝒢:G_{1/α},...,G_0 for S */
        call CONSTRUCT_𝒢(S)

        /* Q' is the set Q sorted by Q(i)[L] */
        Q' ← Q
        sort Q' by x-values of left endpoints
        foreach j, 0 ≤ j < m do NN(j) ← 0
        foreach j, m ≤ j < 2mN^α do Q'(j) ← null

        /* search in D_{1/α},...,D_0 one at a time */
        for i ← 1/α downto 0 do
            begin

                /* determine Q" which is a subset of queries which can
                   be answered at this level.  The remaining queries
                   determine the node numbers in the next level */
                foreach j, 0 ≤ j < 2mN^α do
                    begin t_1(j) ← t_2(j) ← 0
                        Q"(j) ← Q'(j)
                        NN"(j) ← NN(j)
                        if Q'(j_ ≠ null
                            then if Q'(j) [B]≤B_i(NN(j)) and
                                                   T_i(NN(j)) ≤ Q'(j)[T]
                                then t_1(j) ← 1
                                else t_2(j) ← 1
                    end
                call EXTRACT2(Q",t_1); call EXTRACT2(NN",t_1)

                /* answer queries in Q" by performing a one-dimensional
                   range searching on i */
                call RANGE_SEARCH_1D(G_i,Q")

                /* extract Q'-Q" from Q' and reorder the queries
                   according to their node number */
                call EXTRACT2(Q',t_2); call EXTRACT2(NN,t_2)
                for k ← log2n to log2mN^α-1 do
                    foreach j, 0 ≤ j < 2mN^α do
                        if BIT_k(j) = 0 then
                                    begin Q'(j + 2^k) ← Q'(j)
                                          NN(j + 2^k) ← NN(j)
                                    end
```

```
foreach j, 0 ≤ j < 2mN^α do
     begin t(j) ← 0
          NN(j) ← NN(j) + ⌊j/2m⌋N^(1-iα)
          if (Q'(j)·[B] < T_(i-1)(NN(j)) or
                              Q'(j)·[T] > B_(i-1)(NN(j)))
               and Q'(j) ≠ null
               then t(j) ← 1
     end
call EXTRACT2(Q',t); call EXTRACT2(NN,t)
end
end
```

```
procedure CONSTRUCT_𝒯2(E)
    /* construct the point location tree for the set (0:|E|-1) of edges */
    begin
        /* C_{i,j} is a subset of edges which may belong to NODE_i(j) */
        foreach k, 0 ≤ k < |E| do C_{logN,0}(k) ← E(k)

        /* determine the nodes of 𝒯, level by level */
        for i ← logN downto 0 do

            /* extract the appropriate edges from C_{i,j} to form NODE_i(j);
               then form C_{i-1,2j} and C_{i-1,2j+1} from the remaining edges */

            foreach j, 0 ≤ j < 2^i 1 do
                begin NODE_i(j) ← ∅
                    C_{i-1,2j} ← C_{i-1,2j+1} ← C_{i,j}
                    if C_{i,j} ≠ ∅ then
                        begin

                            /* extract from C_{i,j} edges that belong
                               to NODE_i(j) */
                            foreach k, 0 ≤ k < |C_{i,j}| do
                                if C_{i,j}(k)[B] ≤ B_i(j) and
                                        T_i(j) ≤ C_{i,j}(k)[T]
                                    then t(k) ← 1
                                    else t(k) ← 0
                            call EXTRACT1(C_{i,j}t)
                            NODE_i(j) ← C_{i,j}
                            sort edges in NODE_i(j) in the
                                positive x direction

                            /* determine C_{i-1,2j} and C_{i-1,2j+1} */
                            foreach k, 0 ≤ k < |C_{i-1,2j}| do
                                begin if t(k) = 0 and
                                        C_{i-1,2j}(k)[B] < B_{i-1}(2j)
                                        then t_1(k) ← 1 else t_1(k) ← 0
                                    if t(k) = 0 and C_{i-1,2j}(k)[T]
                                            > B_{i-1}(2j+1)
```

```
                                        then t_2(k) ← 1 else t_2(k) ← 0
                                 end
                            call EXTRACT1(C_{i-1,2j}, t_1)
                            call EXTRACT1(C_{i-1,2j+1}, t_2)
                        end
            end
    end


procedure LOCATE1(G,P)

    /* locate the set of points P(0: M-1) in the planar subdivision
       induced by the graph G = (V,E) */
    begin

        /* construct the point location tree 𝒯 for the edges of G */
        call CONSTRUCT_𝒯2(E)

        /* J_0(k) and J_1(k) are the indice of the nodes which we have
           to search for point P(k); L(k) and R(k) are edges on the
           left and right, respectively of P(k) */

        foreach k, 0 ≤ k < M do
            begin J_0(k) ← 0; J_1(k) ← -1
                L(k) ← Ē_{-∞}
                R(k) ← Ē_∞
            end

        /* search in 𝒯 one level at a time */
        for i ← lcgN downto 0 do
            for ℓ ← 1 to 1 do
                foreach k, 0 ≤ k < M do
                    if J_ℓ(k) ≥ 0 then
                        begin TEMPL(k) ← edge in NODE_i(J_ℓ(k)) that is
                                         closest to and left of P(k)
                            TEMPR(k) ← edge in NODE_i(J_ℓ(k)) that is
                                         closest to and right of P(k)
                            if TEMPL(k) is right of L(k) then
                                                   L(k) ← TEMPL(k)
                            if TEMPR(k) is left of R(k) then
                                                   R(k) ← TEMPR(k)
```

```
                                   if L(k) and R(k) bound the same region
                                      then begin P(k) is the region bounded
                                                             by L(k) and R(k)
                                                 Jℓ(k) ← Jℓ⊕1(k) ← -1
                                           end
                                      else if y-value of P(k) = T_{i-1}(2Jℓ(k))
                                           then begin Jℓ⊕1(k) ← 2Jℓ(k)+1
                                                end
                                           else if y-value of
                                                       P(k) < T_{i-1}(2Jℓ(k))
                                                     then Jℓ(k) ← 2Jℓ(k)
                                                     else Jℓ(k) ← 2Jℓ(k)+1
                             end
          end
```

```
procedure CONSTRUCT_δ2(E):

    /* determine the search structure E_logN,...,E_0 for the set E of edges */
    begin

        /* S is the set of edges from which E_1 is formed */
        foreach j, 0 ≤ j < |E| do begin S(j) ← E(j); π(j) ← 0; end
        foreach j, |E| ≤ j < 4|E| do S(j) ← null

        /* determine E_logN,...,E_0 one by one */
        for i ← logN downto 0 do
            begin

                /* determine the edges in E_i */
                foreach j, 0 ≤ j < 4|E| do
                    begin t_1(j) ← t_2(j) ← 0
                        E_i(j) ← S(j); N#_i(j) ← π(j)
                        if S(j) ≠ null then
                            if S(j)[B] ≤ B_i(π(j)) and T_i(π(j)) ≤ S(j)[T]
```

```
                    then t1(j) ← 1
                    else t2(j) ← 1
     end
call EXTRACT2(Ei,t1); call EXTRACT2(N#i,t1)
sort both Ei and N#i lexicographically by values of
N#i(j) and positions of Ei(j) in the direction of
positive x.

/* determine edges which may belong to the next level
   of T */
call EXTRACT2(S,t2); call EXTRACT2(π,t2)
foreach j, 0 ≤ j < 4|E| do
     begin TEMP(j) ← S(j)
          t1(j) ← t2(j) ← 0
          if S(j)[B] < Ti-1(π(j)) then t1(j) ← 1
          if S(j)[B] > Ti-1(π(j))
               then begin t2(j) ← 1
                          TEMPπ(j) ← 2^(logN-i) + π(j)
                    end
     end
call EXTRACT2(S,t1); call EXTRACT2(π,t1)
call EXTRACT2(TEMP,t2); call EXTRACT2(TEMPπ,t2)
foreach j, 0 ≤ j < |TEMP| do begin S(j + |S|) ← TEMP(j)
                                  π(j + |S|) ← TEMPπ(j)
                            end
          end
end
```

```
procedure LOCATE2 (G,P):

    /* locate the set of points P(0: M-1) in the planar subdivision
       induced by G = (V,E) */
    begin

        /* construct the search structure E_logN, E_logN-1, ..., E_0 for the
           set E of edges */
        call CONSTRUCT_β2(E)

        /* P' is the set of points to be located; they are sorted by
           their node numbers and then x-coordinates */
        sort P by x coordiantes
        foreach k, 0 ≤ k < 2M do
            begin NN(k) ← 0; P'(k) ← P(k)
                  L(k) ← Ē_-∞
                  R(k) ← Ē_-∞
            end
        foreach k, M ≤ k < 2M do P'(k) ← null

        /* search in E_logN, ..., E_0 one at a time until edges L(k) and
           R(k), for each k, bound the same region */
        for i ← logN downto 0 do
            begin call SEARCH(E_i,P',TEMPL) /* parallel searching in
                                               Section 2.2.3 */
                call SEARCH1(E_i,P',TEMPR) /* modified SEARCH */
                foreach k, 0 ≤ k < 2M do
                    begin if TEMPL(k) is right of L(k) then
                                                L(k) ← TEMPL(k)
                        if TEMPR(k) is left of R(k) then
                                               R(k) ← TEMPR(k)
                        if L(k) and R(k) bound the same region
                            then begin P'(k) is in the region bounded
                                              by L(k) and R(k)
                                       P'(k) ← null
                                 end
                        t_1(k) ← t_2(k) ← 0
                        TEMP(k) ← P'(k)
                        TEMPNN(k) ← 2^(logN-i) + NN(k)
                        if P'(k) ≠ null then
                            begin if y-value of P'(k) ≤ T_i-1(NN(k))
                                      then t_1(k) ← 1
```

```
                                    if y-value of P'(k) ≥ T_{i-1}(NN(k))
                                        then t_2(k) ← 1
                                end
                            call EXTRACT2(P',t_1)
                            call EXTRACT2(NN,t_1)
                            call EXTRACT2(TEMP,t_2)
                            call EXTRACT2(TEMPNN,t_2)
                            foreach k, 0 ≤ k < |TEMP| do
                                begin P'(|P'| + k) ← TEMP(k)
                                    NN(|P'| + k) ← TEMPNN(k)
                                end
                            end
                    end
            end

procedure CONSTRUCT_β2(E):
    begin
        foreach j, 0 ≤ j < |E| do begin S(j) ← E(j); π(j) ← 0 end
        foreach j, |E| ≤ j < 2|E|N^α do S(j) ← null
        for i ← 1/α downto 0 do
            begin
                foreach j, 0 ≤ j < 2|E|N^α do
                    begin t_1(j) ← t_2(j) ← 0; D_i(j) ← S(j);N#_i(j) ← π(j)
                        if S(j) ≠ null
                            then if S(j)[B] ≤ B_i(π(j)) ≤ S(j)[T]
                                then t_1(j) ← 1
                                else t_2(j) ← 1
                    end
                call EXTRACT2(D_i,t_1); call EXTRACT2(N#_i,t_1)
                sort both D_i and N#_i by lexicographically by values of
                    N#_i(j) and position of D_i(j) in positive x direction
                call EXTRACT2(S,t_2); call EXTRACT2(π,t_2)
                for k ← log 2|E| to log 2|E|N^α - 1 do
                    for j, 0 ≤ j < 2|E|N^α do
                        if BIT_k(j) = 0 then begin S(j + 2^k) ← S(j)
                                                   π(j + 2^k) ← π(j)
                                             end
```

```
                    foreach j, 0 ≤ j < 2|E|N^α do
                        begin π(j) ← π(j) + ⌊ j/2n⌋ N^(1-iα)
                              t(j) ← 0
                              if S(j) ≠ null and (S(j)[B] < T_(i-1)(π(j)) or
                                                   S(j)[T] > B_(i-1)(π(j)))
                                  then t(j) ← 1
                        end
                    call EXTRACT2(S,t); call EXTRACT2(π,t)
                end
        end
```

```
procedure LOCATE3(G,P):

    /* locate the set of points P(0: M-1) in the planar subdivision
       induced by G */
    begin call CONSTRUCT_β2(E)
        sort P by x coordinates
        foreach 0 ≤ k < |E| do
            begin P'(k) ← P(k)
                NN(k) ← o
                L(k) ← Ē_∞
                R(k) ← Ē_∞
            end
        foreach |E| ≤ k < 2|E|N^α do P'(k) ← null
        for i ← 1/α downto 0 do
            begin call SEARCH(D_i,P',TEMPL) /* parallel searching in
                                                Section 2.2.3 */
                call SEARCH1(D_i,P',TEMPR) /* modified SEARCH */
                foreach k, 0 ≤ k < 2|E|N^α do
```

```
if P(k) ≠ null then
    begin if TEMPL(k) is right of L(k) then
                                  L(k) ← TEMPL(k)
          if TEMPR(k) is right of R(k) then
                                  R(k) ← TEMPR(k)
          if L(k) and R(k) bound the same region
              then begin P'(k) is in the region
                                bounded by L(k)
                                and R(k)
                         P'(k) ← null
                   end
    end
for j ← log 2|E| to log 2|E|N^α-1 do
    if BIT_j)k) = 0 then begin P'(K+2^j) ← P'(k)
                               NN(k+2^j) ← NN(k)
                               L(k+2^j) ← L(k)
                               R(k+2^j) ← R(k)
                         end
foreach k, 0 ≤ k < 2|E|N^α do
    begin t(k) ← 0
          NN(k) ← NN(k) + ⌊k/2|E|⌋ N^(1-α)
          if P'(k) ≠ null and
             B_(i-1)(NN(k)) ≤ y-value of
                               P'(k) ≤ T_(i-1)(NN(k))
             then t(k) ← 1
    end
call EXTRACT2(P',t); call EXTRACT2(NN,t)
        end
end
```

function TANGENTS1(A,B)

/* returns the indices of the extremes of the left tangent and right tangent of A,B where A and B are two non-intersecting convex polygons and y-coordinates of vertices in B > those in A */

begin

/* determine the ranges in which j* and i* lie */

if x-value of $B(r_B)$ < x-value of $A(r_A)$

then begin $a \leftarrow 0$; $b \leftarrow r_A$; $c \leftarrow 0$; $d \leftarrow r_B$; **end**;

else begin $a \leftarrow r_A$; $b \leftarrow s_A$; $c \leftarrow r_B$; $d \leftarrow s_B$; **end**;

/* determine $j^{(i)}$ at selected values of i */

foreach i, $i \in \{a+k, a+2k, \ldots, a+(k-1)k\}$ **do**

$j^{(i)} \leftarrow$ MIN_V_BITONIC $(\{\gamma_{i,c}, \gamma_{i,c+1}, \ldots, \gamma_{i,d}\})$

/* i* is in the range $[\bar{i}-k+1, \bar{i}+k-1]$, determine i* and j* in this range */

$\bar{i} \leftarrow$ MINIMUM1$(\{i \mid j^{(i)} \le j^{(h)},\ h = a+k, a+2k, \ldots, a+(k-1)k\})$

foreach i, $i \in \{\bar{i}-k+1, \bar{i}-k+2, \ldots, \bar{i}+k-1\}$ **do**

$j^{(i)} \leftarrow$ MIN_V_BITONIC $(\{\gamma_{i,c}, \gamma_{i,c+1}, \ldots, \gamma_{i,d}\})$

$j^* \leftarrow$ MINIMUM1$(\{j^{(i)} \mid i = \bar{i}-k+1, \ldots, \bar{i}+k-1\})$

foreach i, $i \in \{\bar{i}-k+1, \ldots, \bar{i}+k-1\}$ **do**

if $\gamma_{i,j^*-1} > \gamma_{i,j^*+1}$ /* test $j^* = j^{(i)}$ */

and $\alpha_{i,i-1} > \gamma_{i,j^*}$ **and** $\alpha_{i,i+1} - \gamma_{i,j^*} < \pi$ /* property (2) */

then $i^* \leftarrow i$

/* determine the ranges in which $\bar{j}^*$ and $\bar{i}^*$ lie */

if x-value of $B(\ell_B)$ < x-value of $A(\ell_A)$

then begin $a \leftarrow s_A$; $b \leftarrow \ell_A$; $c \leftarrow s_B$; $d \leftarrow \ell_B$; **end**;

else begin $a \leftarrow \ell_A$; $b \leftarrow n$; $c \leftarrow \ell_B$; $d \leftarrow m$; **end**;

/* determine $j^{(i)}$ at selected values of i */

$k \leftarrow \sqrt{b-a+1}$

foreach i, $i \in \{a+k, a+2k, \ldots, a+(k-1)k\}$ **do**

$\bar{j}^{(i)} \leftarrow$ MAX_Λ_BITONIC $(\{\gamma_{i,c}, \gamma_{i,c+1}, \ldots, \gamma_{i,d}\})$

```
    /* ī* is in the range [ī-k+1,ī+k-1], determine j̄* and ī*
       in this range */
    ī ← MAXIMUM1({i|j^(i) ≥ j^(h), h = a+k,a+2k,...,a+(k-1)k})
    foreach i, i ∈ {ī - k + 1,...,ī + k - 1} do
        j̄^(i) ← MAX_∧_BITONIC ({γ_{i,c},γ_{i,c+1},...,γ_{i,d}})
    j̄* ← MAXIMUM1({j̄^(i)|i - k + 1,...,ī + k - 1})
    foreach i, i ∈ {
        if γ_{i,j̄*-1} ≤ γ_{i,j̄*} > γ_{i,j̄*+1} and α_{i,i+1} < γ_{i,j̄*} and
           α_{i,i-1} - γ_{i,j̄*} > π then ī* ← i
    return (j*, i*, j̄*, ī*)
end


function R_TANGENT_INDEX(A,B):

    /* returns j* /
    begin   /* determine the appropriate range for j* */
1.      if x-value B(r_B) < x-value of A(r_A)
            then begin a ← 0; b ← r_A; c ← 0; d ← r_B; end
            else begin a ← r_A; b ← s_A; c ← r_B; d ← s_B; end
2.      k ← √(b-a+1)
3.      h ← √(d-c+1)

        /* determine j = min{J(i), where γ_{iJ(i)} = min{γ_{i,c+h},γ_{i,c+2h},...,
           γ_{i,c+(h-1)h}} for i = a+k,a+2k,...,a+(k-1)k} */
4.      duplicate {A(a+k),A(a+2k),...,A(a+(k-1)k)} into pattern P2(h-1)
5.          let the resulting array be c(0: (h-1)(k-1)-1);
6.      duplicate {B(c+h),B(c+2h),...,B(c+(h-1)h)} into pattern P1(k-1)
            let the resulting array be D(0: (h-1)(k-1)-1);
7.      foreach i, 0 ≤ i < (h-1)(k-1) do GAMMA(i) ← θ(C(i),D(i))
8.      foreach i, 0 ≤ i < (h-1)(k-1) do
            begin J(i) ← ∞
                case i mod (h-1) of
                    0: if GAMMA(i) < GAMMA(i+1) then
                                                  J(i) ← C+((i mod h-1)+1)h
                    h-2: if GAMMA(i-1) > GAMMA(i) then
                                                  J(i) ← C+((i mod h-1)+1)h
                    else : if GAMMA(i-1) > GAMMA(i) < GAMMA(i+1)
                        then J(i) ← C+((i mod h-1)+1)h
            end
```

```
          /* determine ī ∈ {a+k,a+2k,...,a+(k-1)k} such that γ_{ī,j̄} is the
             smallest among γ_{i,ℓ} for i ∈ {a+k,...,a+(k-1)k} ℓ ∈
             {ℓ-h+1,ℓ-h+2,...,ℓ+h-1} and for some j̄ ∈ {ℓ-h+1,ℓ-h+2,...,ℓ+h-1} */
 9.       j ← min{J(0: (h-1)(k-1)-1)}
10.       duplicate {B(j-h+1,B(j-h+2),...,B(j)} into pattern P1(k-1)
11.           let the resulting array be D(0: (h-1)(k-1)-1);
12.       foreach i, 0 ≤ i < (h-1)(k-1) do GAMMA(i) ← θ(C(i),D(i))
13.       foreach i, 0 ≤ i < (h-1)(k-1) do
              begin J'(i) ← ∞
                  case i mod(h-1) of
                      0: if GAMMA(i) < GAMMA(i+1) then
                                         J'(i) ← j - h + 1 + (i mod h-1)
                      h-2: if GAMMA(i-1) > GAMMA(i) then
                                         J'(i) ← j - h + 1 + (i mod h-1)
                      else: if GAMMA(i-1) > GAMMA(i) < GAMMA(i+1)
                          then J'(i) ← j - h + 1 + (i mod h-1)
              end
14.       j' ← min {J'(0: (h-1)(k-1)-1)}
15.       i' ← min {i|J'(i) = j'}

16.       duplicate {B(j),B(j+1),...,B(j+h-1)} into pattern P1(k-1)
17.           let the resulting array be D(0: (h-1)(k-1)-1)
18.       foreach i, 0 ≤ i < (h-1)(k-1) do GAMMA(i) ← θ(C(i),D(i))
19.       foreach i, 0 ≤ i < (h-1)(k-1) do
              begin J'(i) ← ∞
                  case i mod (h-1) of
                      0: if GAMMA(i) < GAMMA(i+1) then
                                         J'(i) ← j + (i mod(h-1))
                      h-2: if GAMMA(i-1) > GAMMA(i) then
                                         J'(i) ← j + (i mod(h-1))
                      else: if GAMMA(i-1) > GAMMA(i) < GAMMA(i+1)
                          then J'(i) ← j + (i mod(h-1))
              end
20.       j" ← min{J'(0: (h-1)(k-1)-1)}
21.       i" ← min{i|J'(i) = j"}
22.       if j' = j" then ī ← a + (⌊min(i',i")/h-1⌋ + 1)k
                  else if j' < j" then ī ← a + (⌊i'/h-1⌋ + 1)k
                                 else ī a + (⌊i"/h-1⌋ + 1)k

          /* j* = j^(i) for some i ∈ {ī-k+1,ī-k+2,...,ī+k-1}  */
23.       duplicate {A(ī-k+1),A(ī-k+2),...,A(ī)} into pattern P2(h-1)
              let the resulting array be C(0: (h-1)(k-1)-1)
24.       repeat steps 6-20
25.       j* ← min (j',j")
26.       duplicate {A(ī),A(ī+1),...,A(ī+k-1)} into pattern P2(h-1)
              let the resulting array be C(0: (h-1)(k-1)-1)
27.       repeat steps 6-20
28.       j* ← min(j*,j',j")
29.       return (j*)
      end
```

```
procedure MULTI_MAX (D,π,FIRST,LAST)

     /* D(0: n-1) is an array of numbers.  π(i) is the index of the subarray
        to which d(i) belongs, that is D(i) ∈ D_π(i).  Partition D into sub-
        sets such that elements in each subset have the same π-values; find
        i ∈ [|D_j-1|,D_j-1| + |D_j|1] and |D_-1| = 0, FIRST(i) and LAST(i) are the
        indices of the first and the last elements of the subset D_π(i)  */

     begin

          /* logarithmically partition each subset: first determine the
             first element of each partition  */
          foreach i, 0 ≤ i < n do
               if FIRST(i) = i then t(i) ← 1
                    else begin  L ← 0
                                R ← n-1
                                t(i) ← 0
                                while FIRST(i) > L or LAST(i) < R do
                                   if i = ⌊(L+R)/2⌋ + 1
                                        then begin t(i) ← 1
                                                   L ← FIRST(i)
                                                   R ← LAST(i)
                                             end
                                        else if i ≤ ⌊(L+R)/2⌋
                                                   then R ← ⌊(L+R)/2⌋
                                                   else L ← ⌊(L+R)/2⌋ + 1
                         end

          /* classify each partition  */
2.        call RANK(D,t,CLASS)
3.        foreach i, 0 ≤ i < n do
               if t(i) ≠ 1 then CLASS(i) ← CLASS(i)-1

          /* determine the submaximum in each partition, i.e., maximum
             of the elements in the same class  */
4.        foreach i, 0 ≤ i < n do begin SM(i) ← D(i), π'(i) ← π(i) end
5.        for j ← 0 to log n-1 do
               foreach i, 0 ≤ i < n do
                    if CLASS(i) = CLASS(i + (1-2BIT_j(i))2^j)
                         then SM(i) ← max(SM(i),SM(i + (1-2BIT_j(i))2^j))

          /* concentrate the submaximums into consecutive processors */
6.        call CONCENTRATE(SM,CLASS,t)
7.        call CONCENTRATE(π',CLASS,t)
```

```
        /* determine sequentially the maximum of the (at most 2 logn-1)
           of the same subset  */
8.      for j ← 1 to 2 logn-1 do
            begin j ← j+1
                foreach i, 0 ≤ i < n-1 do
                    if π'(i) = π'(i+1)
                        then SM(i) ← max(SM(i),SM(i+(1-2BIT(i))2^j))
            end

        /* concentrate the maximums into consecutive processors  */
9.      foreach i, 1 ≤ i < n do
            if π'(i-1) < π'(i) then t(i) ← 1
                               else t(i) ← 0
        t(0) ← 1
10.     call CONCENTRATE(SM,π',t)
    end
```

LME

-8