M.Tech. (Computer Science) Dissertation Series

Parallel Architecture and Algorithms for Fast Detection of Basis Polygons

A dissertation submitted in partial fulfilment of the requirements for the M. Tech. (Computer Science) degree of the Indian Statistical Institute, Calcutta.

By
Arijit Laha

Under the supervision of
Prof. Bhabani Prasad Sinha

INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road
Calcutta - 700 035

July 1997

Indian Statistical Institute

203, B.T. Road

Calcutta- 700 035

Certificate of approval

This is to certify that the thesis titled, **Parallel Architecture and Algorithms for Fast Detection of Basis Polygons** submitted by **Arijit Laha** towards partial fulfillment of the requirements for the degree of M. Tech. in Computer Science at the Indian Statistical Institute, Calcutta, embodies the work done under our supervision.

Bhabani P. Sinha
Professor and Head
Advanced Computing and Microelectronics Unit
Indian Statistical Institute
Calcutta- 700 035

Abstract

Though a human brain is inherently slower than digital computer, in the computations involving vision it usually has an edge over the later due to its massive parallelism and ability to deal with analog data directly. However, with the advent of parallel processing it is possible to devise algorithms following human reasoning closely. In this dissertation we choose the problem of polygon detection which is actually a subclass of the general problem problem of simple geometrical figure recognition. For such problems digital computer is quite efficient because the description of a scene comprising of geometrical figures can be represented with simple numerical data. Here we try to devise some efficient algorithms for parallel processors that follows the sort of reasoning a human being makes when trying to comprehend a visual scene. Here we develop two O(n) algorithms using an architecture comprising of an n x n array of processors. This makes the cost of computation $O(n^3)$, which is optimal.

Acknowledgement

I am deeply grateful to Prof. Bhabani P. Sinha for his guidance, advice and encouragement in this project, without which this project could not have been successful. I also thank Prof. Bhargab B. Bhattacharya for providing constructive criticism that led to more generalized approach to the problem.

Lastly, I would like to thank all my classmates, especially Mr. Debashish Sarkar and Mr. Manoj Baruah who helped me to remain sane during two maddening years of this M. Tech. course.

Calcutta,
July, 1997.

(Arijit Laha)

Contents

Chapter 1

Introduction

If we put aside the issue of actual mechanism, from a generalized viewpoint the human brain and the digital computer both are in the business of computation (in fact, this supposition forms the basis of the study of artificial intelligence). Hence, at the heart of every job performed by both systems there are some algorithms.

However, as we try to examine and compare two systems more closely, numerous differences and almost complementary nature in terms of handling the type of jobs efficiently becomes evident. The human brain is an analog device, capable of interacting with the environment by means of sensory organs. These sensory organs pick up analog stimuli from the environment and sends appropriate electrochemical signal to the central nervous system (brain and spinal chord) through the network of neurons making up the nervous system. The central nervous system (which is again an aggregation of billions of neurons) analyzes the received signal, performs necessary computations and produces appropriate response. All these steps are performed using analog electrochemical signals.

On the other hand, a digital computer is a "number cruncher", whatever computation it does, it does with numbers. When it works with some analog signal, before the actual computation starts the analog signals must be transformed into some numerical representation.

Owing to the distinction mentioned above a digital computer is orders of magnitude faster than human brain when it comes to numerical computation. However, when it comes to a "pattern matching" problem, especially those involving natural stimuli that can be captured by human sensory organ, human brain beats digital computer in most of the cases quite comfortably. To emphasize the point let us inspect the problem of vision. Among all the sensory inputs a human being receives vision is the richest and most important. Usually vision comprises 60% of the sensory input a person receives. The field of study "computer vision (CV)" deals with the problem of perception, memorizing and comprehension of visual scenes by digital computers. Any CV algorithm that comes nominally close to the human ability is bound to perform millions of transformation operations, which when implemented with the fastest available sequential digital computer today, will require several seconds, if not minutes to process a single scene. But though human brain works with its basic building block neurons which are a few orders of magnitude slower than electronic hardware, the job is done within a few milliseconds (a neuron need about 2 milliseconds to generate a response, but the perception and interpretation of a time-varying complex scene takes between 70 to 200 milliseconds). For an illustration of the point , consider the problem of recognizing a person whose face from frontal direction is known to a viewer and an image of the same frontal view is stored in a computer running a face recognition algorithm.

Now a side view of the person is presented to the viewer as well as the computer. Usually the human viewer would come up with the correct identification within a few milliseconds, but for the computer it is a formidable task with considerable possibility of failure.

The above observation gives us a sense of a paradox unless we remember that human brain employs incredibly massive parallelism. Each neuron is a processing unit and there are billions of them working together in a cohesive and unified fashion whose precise mechanism is still a great (perhaps the Greatest!) mystery. With the advent of parallel architecture we can enhance the performance of a digital computer greatly over that of a mere sequential machine, but even that can not beat human brain in the sort of problems as described above.

There is also another snag. In a digital computer data are saved in memory locations and retrieved by addressing the location. However, in human brain information is stored in form of neural pathway , i.e. a collection of neurons connected by synaptic connections. A large collection of such neural pathways forms a particular memory (say, of some object, some event etc.). The stored information is retrieved using retrieval cue. The better the cue is the faster is the retrieval. (this resolves the paradox that the bigger the knowledge-base of a digital computer the slower is its performance while for human brain greater knowledge-base means swifter performance.) The only thing in the realm of digital computer technology that can come nominally close to such mechanism is "content-addressable memory".

Nevertheless, there is atleast one case of visual scene recognition where the digital computer works faster than human brain. This is when the scene contains geometrical figures and nothing else. Upto a moderate level of complexity of the figures human brain works well, but as complexity of the figures increases digital computer beats human brain hands down. This is so due to the fact that when low-level image processing is over, the scene can be represented by simple numerical data.

Here, in this dissertation we shall try to develop some parallel algorithms to detect some simple geometrical figures. In doing so we shall try to follow the sort of reasoning a human being seems to follow, i.e., try to combine the best of two worlds. To illustrate the point, think of a child who can recognize a polygon drawn on a paper. Give him a paper with several polygons drawn on it and tell him to show the polygons. He will almost invariably put his finger at the vertex of a polygon and trace around the boundary of the polygon,, then do the same for another polygon and so on.

We specifically go for the problem of detection of the basis polygons formed by a set of straight-lines in a two-dimensional plane whose endpoint co-ordinates are supplied as input data. We use MIMD (Multiple Instruction Multiple Data) SM (Shared Memory) parallel processing architechture. A small part of the shared memory is content-addressable memory.

Our goal is to develop some efficient algorithms for solving the problem and study the maximum possible speed-up under the afore-described architecture and the bottlenecks arising in the attempt for further speed-up.

Chapter 2

Problem specification

Problem: Given n straight lines in a two-dimensional plane in terms of their endpoints, detect all the basis polygons formed by them.

Specification: Since a set of straight-lines can in general produce a lot of varied situations, before proceeding further, we must state clearly what we consider as a polygon. Here we formalize two specifications. The first one somewhat restricts the randomness of the input but allows the development of a very easy-to-understand and elegant algorithm. The second one demands no restriction on the input data but the algorithm developed to deal this case is a bit involved.

The specifications are given below.

(1) The polygons are simple (contains no nested structure), has no degenerate part (no line has an endpoint inside a polygon. In addition, polygons are convex (this condition is not strictly necessary, however this makes the algorithm simpler).

(2) The polygons may have internal structures, degenerate portions and may be non-convex.

Analysis: Given n straight lines in a two-dimensional plane we can make following observations.

(a) An intersection point involves at least two straight lines. Thus nunber of maximum possible intersection points is ${}^{n}C_{2} = n(n - 1)/2$.

(b) Computing the maximum nunber of basis polygons goes as follows,

We consider an ensemble of straight lines initially empty, in each step we add a new straight line to the ensemble and compute the maximum number of new basis polygons that may be created due to inclusion of the new line.

As it is evident in i-th step the i-th line is added to an ensemble of (i - 1) lines, and the new line can intersect at most (i - 1) already existing lines. Now, for creating a polygon a line must intersect atleast two other lines. For (i - 1) already existing lines the i-th line can find at most (i - 2) such line pairs. Therefore, the inclusion of i-th line to the ensemble can increase the number of basis polygons by at most (i -2). Thus, by counting the number of new basis polygons generated until n-th straight line is added to the ensemble, we have maximum number of polygons that can be created by n straight lines is given by,

$$1 + 2 + 3 + \ldots + i + \ldots + (n - 2) = (n - 1)(n - 2)/2 = {}^{(n-1)}C_{2}$$

Again, a polygon may have at most n vertices (i.e. each of the n sraight lines is one side of the polygon).

Thus, the optimal cost of detecting $O(n^2)$ polygons each having O(n) vertices is $c(n) = O(n^3)$.

Chapter 3

A closer look at the problem

Before we proceed further with the development of the algorithms, we have shall describe here the strategy employed and various conventions used in the later part of the discussion. We also shall try to develop a deeper understanding of the problem.

Strategy: Here we recall the child in chapter (1) showing the polygons drawn on a paper. He puts his finger at a vertex of a polygon, traces its boundary with his finger until his finger points to the vertex from which he started. The strategy employed in the following algorithms is precisely the same as that used by the child.

Now we state a few observations .

(1) Each vertex (except a degenerate turning point) of a polygon is an intersection point of atleast two lines (Fig. 1).

(2) Each polygon can be described by a list of vertices, the first in the list being the same as the last (Fig. 1) .

(3) Any two consecutive vertices in the list are joined by a single line and there are no other intersection point between these two, i.e. the consecutive vertices are closest neighbors on the line joining them along the direction of traversal of the polygon.

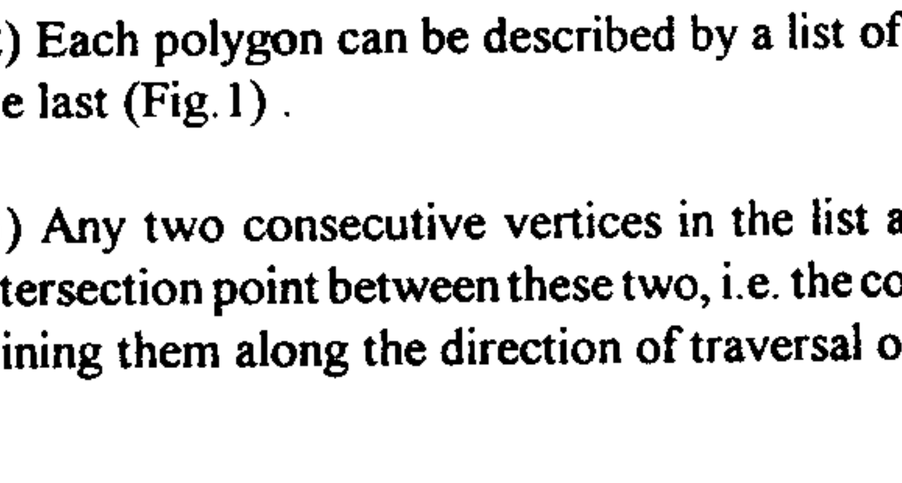

Fig. 1

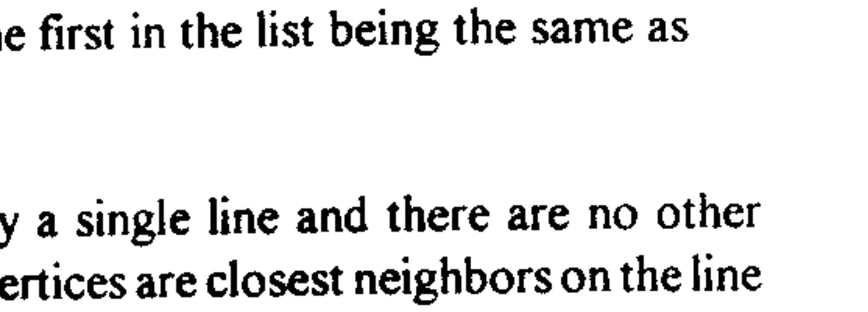

Fig. 2

To traverse each polygon systematically we have to define a sense of direction on the 2-D plane and also impose some sense of ordering over the 2-D points.

The points are described by their (x,y) co-ordinates. We define a point $P_1(x_1, y_1)$ is in left of another point $P_2(x_2, y_2)$ if either $x_1<x_2$ or $x_1 = x_2$ and $y_1<y_2$. We express this as $P_1 <P_2$. The sense of rightness is expressed accordingly. However, to allow some amount of approximation in the computation, we define the equality of two points as the difference of their x-value and/or y-value being less than |2|.

A line $\overline{A(x_1, y_1)B(x_2, y_2)}$ has a point $C(x_3, y_3)$ at its left-hand side (upper half-plane) (Fig.2) if the vector cross-product of the vectors $\overline{AB}$ and $\overline{AC}$ is positive i.e. $(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$ is greater than zero.

We also use the concept of vector dot product to compute the angle between two lines. The angle between two lines $\overline{AB}$ and $\overline{AC}$ is computed as $\overline{AB} \bullet \overline{AC} / |\overline{AB}||\overline{AC}|$.

Now we are in a position to discuss the strategy of the traversal over a polygon in more concrete terms. Consider Fig. 3. There we have five lines forming four basis polygons. The point a is the intersection of lines 1 and 2. We start from a along line 1, i.e., a is the first vertex in the vertex list. We make a convention of looking rightwards from the first vertex along the starting line (here line 1) to find the second vertex, i.e., we pick up the closest right neighbor of a on line 1 as second vertex, in this case b. Here b is the intersection of three lines 1,3 and 4. To continue the traversal of the basis polygon further we have to choose one of the lines between 3 and 4 to traverse along. However for any one choice of line there are two possibility for next vertex, namely d and f if we choose line 3, c and e if we choose line 4. This is where we have to adopt some conventions in order to facilitate an unambiguous and systematic method of choosing the next line and next vertex to be covered by the traversal.

First, we adopt a convention called "move-left strategy", which dictates that while choosing among two closest neighbors on the line to turn into, choose the neighbor on the left-hand side of the current line. Thus to choose between d and f on line 3 we compute the vector cross-product of ab and ad and choose d if the result is positive, otherwise we choose f. If there are more than one lines to turn into, choose the closest neighbor on each lines conforming with move-left strategy. In our case we choose d on line 3 and c on line 4.

Now we know what will be the next vertex if the traversal chooses a particular line. We choose among all possible next lines invoking a strategy "move-leftmost". This chooses among all prospective next lines the one that makes smallest angle with the current line on turning into. Thus here the line 3 is chosen because the angle abd is less than angle abc.

Thus the traversal reaches d by line 3. The point d is the intersection point of lines 3 and 1. There is only one line at d to turn into. So we invoke move-left strategy to choose between c and a (on line 1). Point a is chosen, which is the starting vertex. So the traversal terminates producing the vertex list < abda >.

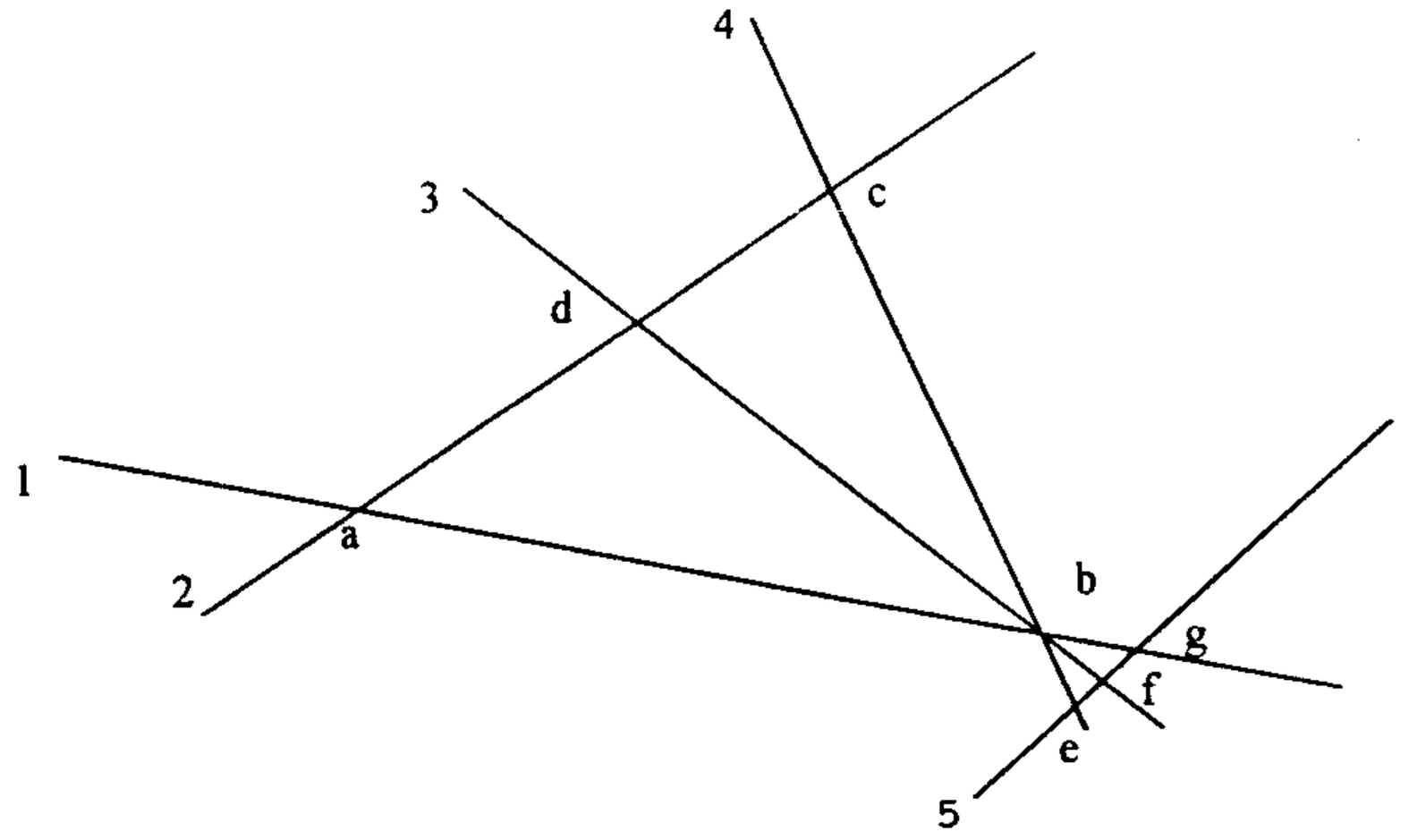

Fig. 3

Similarly, starting from d along line 3 we detect polygon < dbcd >, starting from b along line 3 we detect polygon < bfgb > and starting from b along line 4 we detect polygon < befb >.

The procedure described above is adequate to deal with all the cases arising out of specification (1). The modification required for dealing with cases arising for specification (2) will be described before we develop the algorithm for it.

Chapter 4

Algorithms

4.1

About the data structures:
Following data structures are used in the algorithms.

(1) Point: (int x, int y) // the structure of integer co-ordinate of a point.

(2) fPoint: (float x, float y) // the structure of floating point co-ordinate of a point.

(3) Line:(Point leftend, Point rightend) // the structure containing the description of a line.

(4) Neighb:(Point leftN, Point rightN) // the structure containing neighborhood information of a point on a line.

(4) Line lines[n] // an array of length n of structure Line. lines[i] contains the i-th input line.

(5) fPoint ipoints[n][n] // an n x n matrix of structure fPoints in shared memory and is content-addressable memory (CAM) which can be searched one row at a time. ipoints[i][j] contains the intersection point of i-th line and j-th line.

(6) Neighb inb[n][n] // an n x n matrix of structure Neighb in the shared memory. inb[i][j] contains the closest neighbors of the intersection point of i-th line and j-th line (the point ipoints[i][j]) on i-th line.

(7) fPoint a[n][n-1] // an n x n-1 matrix of structure fPoints

(8) int b[n][n-1] // an n x n-1 matrix of integers

(9) int llist // a list of integers

(10) Point vlist // a list of structure Point

Note: Following operations can be performed on both the lists.

(a) First(): returns the first element of the list.

(b) Last(): returns the last element of the list.

(c) Next(): returns the element next to the one last accessed.

(d) Empty(): returns TRUE if the list is empty, otherwise returns FALSE.

(e) Islast(): returns TRUE if the last access was to the last element of the list, otherwise returns FALSE.

(d) Insert(): inserts an element at the head of the list (new element becomes the header).

(e) Append(): appends an element at the end of the list.

(f) InsertList(): inserts a list at the beginning of another list.

(g) AppendList(): appends one list at the end of another list.

4.2

About the architecture:

The architecture used employs n^2 processors organized as an n x n array. The processors are identified by their array indices (P_{ij} is the processor at i-th row and j-th column). All the processors can address a shared memory which includes a block of CAM. The shared memory is accessible in CREW fashion.

4.3

Procedures:

Now we describe some functions that will be used in our algorithms.

(1) fPoint Intersection(int i, int j) // computes the intersection point of i-th and j-th lines(the i-th and j-th element of input array lines []) and returns the floating point co-ordinates of the intersection point. If the lines are non-intersecting it returns an invalid value.

(2) Sort(int i) // this is a standard O(n) parallel sorting procedure employing n processors to sort n elements. Here the elements of ipoints[i] are sorted by the processors of i-th row in increasing order of rightwardness. The sorted list output in the array a[i], while the b[i][j] contains the column no. of a[i][j] in ipoints[i].

(3) Compute_neighbors(int i) // this procedure is executed by the processors P_{ii} . For each point a[i][j] it computes the left neighbor of the point in row a[i] from columns preceding j-th column such that the point chosen is closest to a[i][j] but not equal. For a[i][1] if the point is not equal to the left endpoint of lines[i], the later is chosen as the left neighbor, otherwise the left neighbor is set INVALID (this is the case when the left endpoint is also an intersection point). Similarly, the right neighbor of a[i][j] is one from a[i] in a column succeeding j-th column, closest to a[i][j] and different from it. For the rightmost point in a[i], if it is different from the right endpoint of lines[i] then the later is chosen as the right neighbor otherwise the right neighbor is set INVALID (right endpoint is also an intersection point). The neighborhood information of a[i][j] is stored at inb[i][b[i][j]]. The procedure takes O(n) time.

(4) SearchCAM(int i, fPoint P) // this function searches the i-th row of ipoints for the point P, returns the list llist containing the no. of the columns containing P in increasing order. This is an O(1) operation.

(5) int Isleft(fPoint A, fPoint B, fPoint C) // this function computes the vector cross-product of vectors AB and AC, returns 1 (TRUE) if the result is positive, returns 0 (FALSE) otherwise.

(6) int Find_Imost(fPoint A, fPoint B, fPoint C, fPoint D) // this function computes the angles ∠ABC and ∠ABD, returns 1 if ∠ABC < ∠ABD, returns 2 otherwise.

4.4

Algorithm 1

4.4.1

Algorithm:

Now we are in a position to write down the first algorithm.

Algorithm 1:

```
Each processor has the following local variables,
int l_on, l_with
fPoint position

Input: An array of structure Line of length n lines[n].
Output: If a processor succeeds to detect a polygon, it outputs a
list of vertices( points ) vlist.

Step 1: Initialization.
          for i, j = 1 to n all processors P_ij do in parallel
              l_on ← i
              l_with ← j
              ipoints[l_on][l_with] ← INVALID
              inb[l_on][l_with] ← INVALID

Step 2: Computation of intersection points.
          for i, j = 1 to n and i ≠ j all processors P_ij do in
          parallel
              position ← Intersection(lines[ i ], lines[ j ])
              ipoints[l_on][l_with] ← position

Step 3: Sorting the intersection points on a line.
          for i = 1 to n all processors in i-th row do in parallel
              Sort(i)

Step 4: Computing the neighbors, only the diagonal processors
        P_ii are at work.
          for i = 1 to n all processors P_ii do in parallel
              Compute_neighbors(i)

Step 5: Detecting the polygons.
          for i, j = 1 to n and i ≠ j all processors P_ij do in parallel
              Detect_polygon()
```

The procedure Detect_polygon() is the heart of the whole algorithm and given below.

```
Procedure Detect_polygon()
   {
   if( position = INVALID ) // checkpoint (1)
      abort                 // no intersection point to start with
   llist ← SearchCAM(l_on, position)
   if ( First(llist) ≠ l_with )  // checkpoint (2)
      abort                      // avoid redundant computation
   fPoint Startvertex ← position
   fPoint Current_v ← Startvertex
   Neighb nb ← inb[l_on][l_with]
   fPoint Next_v ← rightN(nb)    // choose the right neighbor
                                 // of Startvertx as the next_v
   fPoints pNext_v ← Next_v
   int Current_l ← l_on

   while( Next_v ≠ Startvertex )
        {
        Append( vlist, Current_v )
        llist ← SearchCAM(Current_l, Next_v)
        if( Empty(llist) )        // checkpoint (3)
          abort                   // the Next_v is an endpoint of
                                     a line
      int pNext_l ← First(llist)
      int notFound ← FALSE
      while( notFound = FALSE )
           {
            nb ← inb[pNext_l][curr_l]
            if( leftN(nb) ≠ INVALID )
                {
                if( Isleft( Current_v, Next_v, leftN(nb) )
                    pNext_v ← leftN(nb)
                    break       // break from while loop
                 else
                   if( rightN(nb) ≠ INVALID )
                     pNextv ← rightN(nb)
                     break           // break from while loop
           }
      else
        {
        if( rightN(nb) ≠ INVALID )
          if( Isleft( Current_v, Next_v, rightN(nb) )
             pNext_v ← rightN(nb)
             break       // break from while loop
        }
```

```
if( Islast(llist) )
   notFound ← TRUE
else
   pNext_l ← Next(llist)
}
```

// end of while loop. If the flag notFound is still FALSE then we have found one possible left move from Next_v. However, if Next_v is an intersection point of more than two lines then there may be other possible left moves. We have to employ move left-most strategy to resolve the conflict.

```
if( Islast(llist) )
   notFound ← TRUE

int ppNext_l ← Next(llist)

while( notFound = FALSE )
  {
  nb ← inb[ppNext_l][Current_l]
  if( leftN(nb) ≠ INVALID )
     {
     if( Isleft( Current_v, Next_v, leftN(nb) )
       {
      if(Find_lmost(Current_v,Next_v,leftN(nb),pNext_v)=1)
         Next_v ← leftN(nb)
         pNext_l ← ppNext_l
       }
     else
       {
       if( rightN(nb) ≠ INVALID )
      if(Find_lmost(Current_v,Next_v,rightN(nb),pNext_v)=1)
        Next_v ← rightN(nb)
        pNext_l ← ppNext_l
       }
else
  {
  if( rightN(nb)≠ INVALID )
   if(Find_lmost(Current_v,Next_v,rightN(nb),pNext_v)=1)
          Next_v ← rightN(nb)
          pNext_l ← ppNext_l
  }
if( Islast(llist) )
  notFound ← TRUE
else
  ppNext_l ← Next(llist)
}
```

```
     // end of second while loop. At this stage  if a legitimate
left move from Next_v exists then the vertex is pNext_v and the
next line to turn into is pNext_l. Otherwise we have Next_v =
pNext_v.

          if( Next_v = pNext_v )    // checkpoint (4)
             abort                  // no possible left move,
                                       failure to detect polygon
          else
             {
             Current_v ← Next_v
             Next_v ← pNext_v
             if( Next_v = rightN(inb[pNext_l][Curr_l]) )
                if( position < Next_v ) // checkpoint (5)
                   abort                // avoid redundant
                                           computation
             Curr_l ← pNext_l
             }
           }

     // the main while loop ends. Now the Next_v  is same as Startvertex,
i.e., the traversal of the polygon is complete. The list of vertex vlist
contains all the vertices of the polygon traversed in the order of
traversal. We only need to append Next_v to vlist to close the traversal.

     Append( vlist, Current_v )
     Append( vlist, Next_v )
     }

     // end of procedure Detect_polygon.
```

4.4.2

An example:

To understand the working of the algorithm let us work out with a simple example (Fig. 4).

Fig. 4 has 4 lines, constituting two basis polygons. Let us now observe the performance of the algorithm on this figure. Table 1 is the input array lines.

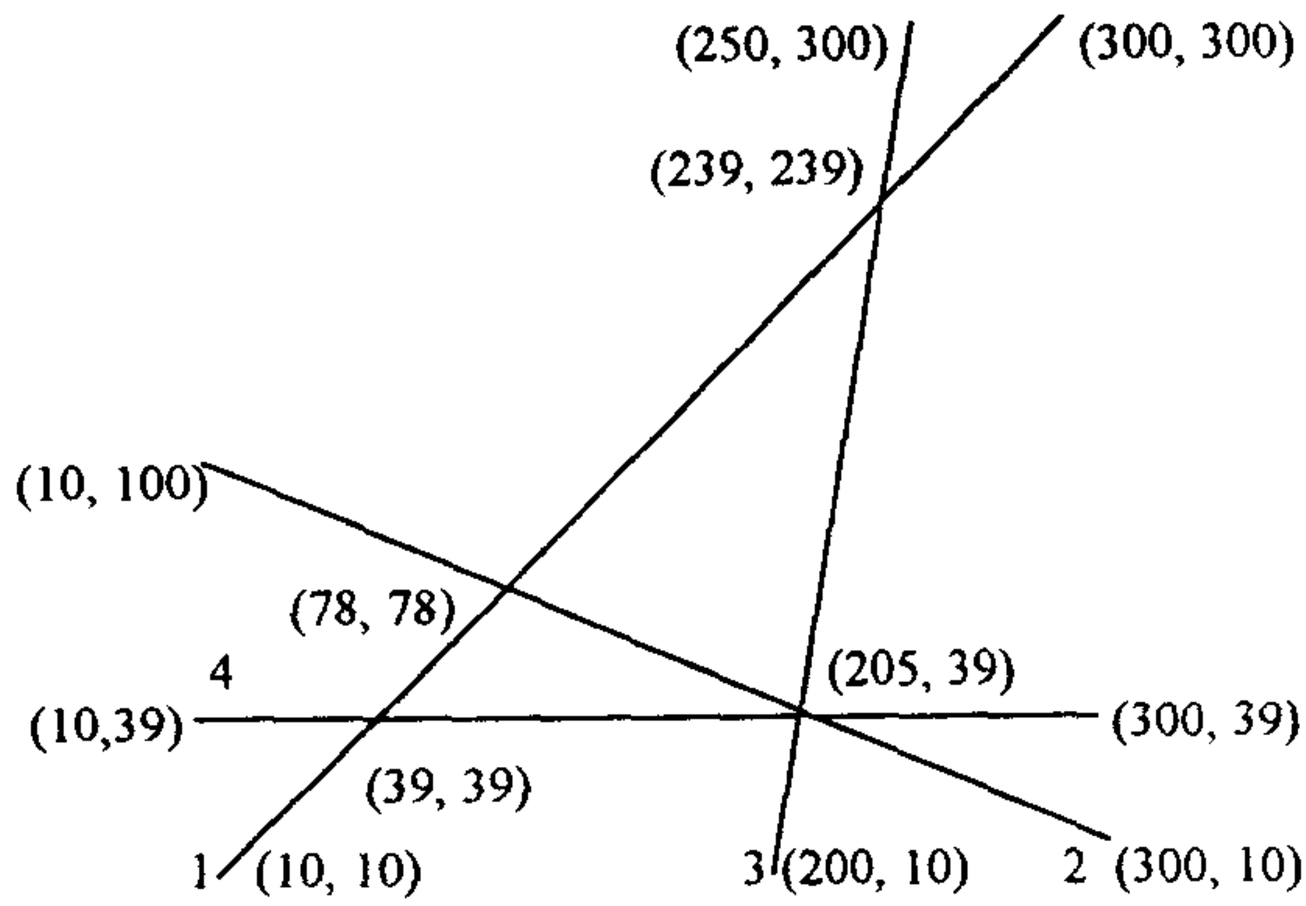

Fig. 4

lines:

	left endpoint	right endpoint
1.	(10, 10)	(300, 300)
2.	(10, 100)	(300, 10)
3.	(200, 10)	(250, 300)
4.	(10, 39)	(300, 39)

Table 1

After the completion of step 2 (O(1) time) the matrix ipoints contains the intersection point information. Table 2 below shows the ipoints.

ipoints:

	1.	2.	3	4.
1.	INVALID	(78, 78)	(239, 239)	(39, 39)
2.	(78, 78)	INVALID	(205, 39)	(205,39)
3.	(239, 239)	(205, 39)	INVALID	(205,39)
4.	(39, 39)	(205, 39)	(205, 39)	INVALID

Table 2

After the sorting in step 3 is over the matrices a and b contains the following data,

a:

1.	(39, 39)	(78, 78)	(239, 239)
2.	(78, 78)	(205, 39	(205, 39)
3.	(205, 39	(205, 39	(239, 239)
4.	(39, 39)	(205, 39)	(205, 39)

b:

1.	4	2	3
2.	1	3	4
3.	4	2	1
4.	1	3	2

Table 3

In step 4 the procedure Compute_neighbors uses the matrices a and b to produce the neighborhood information in the matrix inb[][]. This is an O(n) operation. The result of step 4 for the given example is shown in the following table.

inb:

	1.	2.	3.	4.
1.	LN: INVALID RN: INVALID	LN: (39, 39) RN: (239, 239)	LN: (78,78) RN: (300, 300)	LN: (10, 10) RN: (78, 78)
2.	LN: (10, 100) RN: (205, 39)	LN: INVALID RN: INVALID	LN: (78, 78) RN: (300, 10)	LN: (78, 78) RN: (300, 10)
3.	LN: (205, 39) RN: (250, 300)	LN: (200, 10) RN: (239, 239)	LN: INVALID RN: INVALID	LN: (200, 10) RN: (239, 239)
4.	LN: (10, 39) RN: (205, 39)	LN: (39, 39) RN: (300,39)	LN: (39, 39) RN: (300, 39)	LN: INVALID RN: INVALID

LN: left neighbor
RN: right neighbor

Table 4

Now all processors P_{ij} ($i \neq j$) can invoke the procedure Detect_polygon() in step 5.

Let us review what each processor comes up with.

P_{12} : Starts from point (78, 78) along line 1, goes to right neighbor (239, 239), turns left to (250, 300) along line 3. But (250, 300) is an endpoint of line 3 and no intersection occurs at it. So no possible left move is possible from there on. The procedure terminates without success (at checkpoint (3) in Detect_polygon).

P_{13} : Starts from (239, 239) along line 1, goes to right neighbor (300, 300), which is an endpoint of line 1 and no intersection. The procedure terminates without success (checkpoint (3)).

P_{14} : Suffers same fate as P_{12}.

P_{21} : Starts from (78, 78) along line 2, next is (205, 39). Therefrom it turns left to (239, 239) along line 3. However, the point chosen is the right neighbor of (205, 39), which is the intersection point of line 2, 3 and 4. Hence, the processors P_{32} and P_{34} are entitled to start from (205, 39) along line 3 towards (239, 239) and they will follow precisely the same path as the one to be taken by P_{21} afterwards. Thus we can see that if in a particular traversal, there are more than one (i.e., at the start) instances of choosing the next vertex that is a right neighbor of an intersection point, then there will be more than one processors traversing the same polygon (provided the traversal closes), with different starting points. This amounts to redundant computation. To avoid this we invoke a convention that, if there are several processors traversing the same polygon, only the one with rightmost "position" value among them is allowed to complete the traversal. Now, position

of P_{21} (78, 78) is left to (205, 39). Hence, P_{21} aborts computation (marked as checkpoint (5) in Detect_polygon).

P_{23}: Starts from (205, 39) along 2 towards (300, 10) and terminates (checkpoint (3)).

P_{24}: The searchCAM(2, (205, 39)) produces the llist for which First(llist) = 3. Hence the process terminates (checkpoint (2)). Otherwise it would have followed the same path as P_{23}.

P_{31}: Suffers the same fate as P_{13}.

P_{32}: Starts from (205, 39) along line 3 towards (239, 239) chooses successive two left move at (239, 239) and (78, 78), reaches (205, 39) closing the traversal. Hence it terminates with success, the result being the vertex list of the polygon < (205, 39), (239, 239), (78, 78), (205, 39) > .

P_{34}: Suffers the same fate as P_{24} .

P_{41}: Starts from (39, 39) along line 4 towards (205, 39). Here it has two option of left move, to (239, 239) along line 3 and to (78, 78) along 2. According to move leftmost strategy it invokes the procedure Find_lmost() to make a proper choice and chooses (78, 78). Therefrom the traversal turns left to (39, 39) closing it. Hence the process terminates with success, the result < (39, 39), (205, 39), (78, 78), (39, 39) > .

P_{42}: Suffers the same fate as P_{13}.

P_{43}: Suffers the same fate as P_{24}.

As evident from above discussion all redundant computation is avoided and only the polygons (as per specification (1)) are detected successfully by as many processor. Each of these processors need as many no. of steps as the no of vertices in the polygon detected by it, which can be at most n for n straight lines. Thus the procedure Detect_polygon( ) takes atleast O(n) time.

There are two more checkpoints in Detect_polygon( ), (1) and (4). (1) deals with the case when the lines l_on and l_with are non-intersecting, i.e., there is no vertex to start from. (4) deals with the case when there is no possible left move, i.e., the intersection point is also the endpoint of some line in the right half-plane of the current line.

4.4.3

Analysis:

Now it is time for an estimation of the space and time complexities of the algorithm developed. The largest data structures used in the algorithms are the n x n matrices ipoints, inb, a and b. They occupy $O(n^2)$ space and shared by all the processors.

The other data structures used (apart from local variables) are llist, vlist, and processor queue, all takes O(n) space in worst case. However, since each processor maintains all these data structures of its own, the total space consumed by them is $O(n^3)$.

Hence, the worst case space complexity is $O(n^3)$.

Let us turn our attention to the time complexity.

Step (1) *(initialization)* and step (2) *(computation of intersection points)* takes O(1) time.

Step (3) (sorting) involves sorting of parallel sorting of n elements using n processors. There are several O(n) algorithms for it(one described in appendix). n set of n elements are sorted parallely by n set of n processor. So it is an O(n) step.

Step (4) (computing neighborhood information) uses n processors in parallel, each processor computing neighborhood information from a sorted array of O(n) elements. This requires O(n) time(a C++ implementation of the routine is given in the appendix).

In step (6) n(n-1) processors execute the procedure Detect_polygon in parallel.

If the procedure succeeds to detect a polygon with k sides it must traverse k vertices in k steps. The largest number of sides that a polygon can have is the same as the no. of straight lines present i.e., n. To detect a polygon with n sides the procedure must traverse n vertices. Hence the minimum time required is O(n). Let us see whether the maximum time required is also the same.

When the traversal reaches a vertex that is intersection of only two vertices, it finds the next vertex in 1 step. However, if the vertex is an intersection of more than two lines, say i lines, then the process must choose one of (i-1) lines (the vertex is reached by along one of by the traversal, hence the next line chosen is one of the rest) as the next line. This takes (i-1) steps. However, in this process the line corresponding to leftmost move is chosen. Since the same strategy of leftmost move will be followed subsequently (the polygon is convex) the (i-2) lines rejected at this vertex can never be sides of the polygon. Thus the maximum possible number of lines that can be sides of the polygon is reduced by (i-2) precisely the no. of extra step needed at the vertex to choose the next line. Thus we can conclude that the worst-case run time of the procedure is O(n).

Therefore, the time complexity of algorithm (1) is O(n).

4.5

Algorithm 2

4.5.1

New hurdles towards generalization and their solution:

Now with full understanding of Algorithm 1 at our disposal we can venture into the development of the more generalized algorithm to take care of the general situation as allowed by specification (2).

First of all let us list the new situations we need to deal with.

(*i*) Non-convexity of the polygons (Fig. 5).

(*ii*) Lines with endpoints inside polygons (degenerate structures) (Fig. 6).

(*iii*) Nested polygons (Fig. 7).

(*iv*) Any combination of above three situations

To deal with non-convexity, we observe that our left move strategy in its original form is not adequate. At some point of the traversal of a non-convex polygon there will be no possible left move (as at point c of Fig. 5). Hence to continue the traversal we must allow the traversal to seek a right move when no left move is possible.

To confront degeneracy we have to allow the traversal to turn around from an endpoint. As shown in Fig.6, the polygon < abcdcecfcbgha > has a degenerate portion < bcdcecfcb >. To allow the traversal to capture the degenerate portion it is imperative to modify our strategy to allow turn around in case neither a left move nor a right move is possible.

In Fig. 7 the polygon < cdec > is nested within < abcfa > and we propose to detect it as one polygon < abcdecfa > . As it turns out, the above-mentioned facilities are enough for the job.

We leave the job of detecting the degenerated part or the nested part from the vertex list, to some postprocessing algorithm, which should be easy enough to develop with the aid of the information provided by this algorithm about the kind of move performed at each vertex to reach the next vertex

So we can summarize the new strategy of choosing next vertex of a traversal. The options are listed by priority. A particular move is executed after attempt to execute all moves preceding it has failed.

(1) Left move

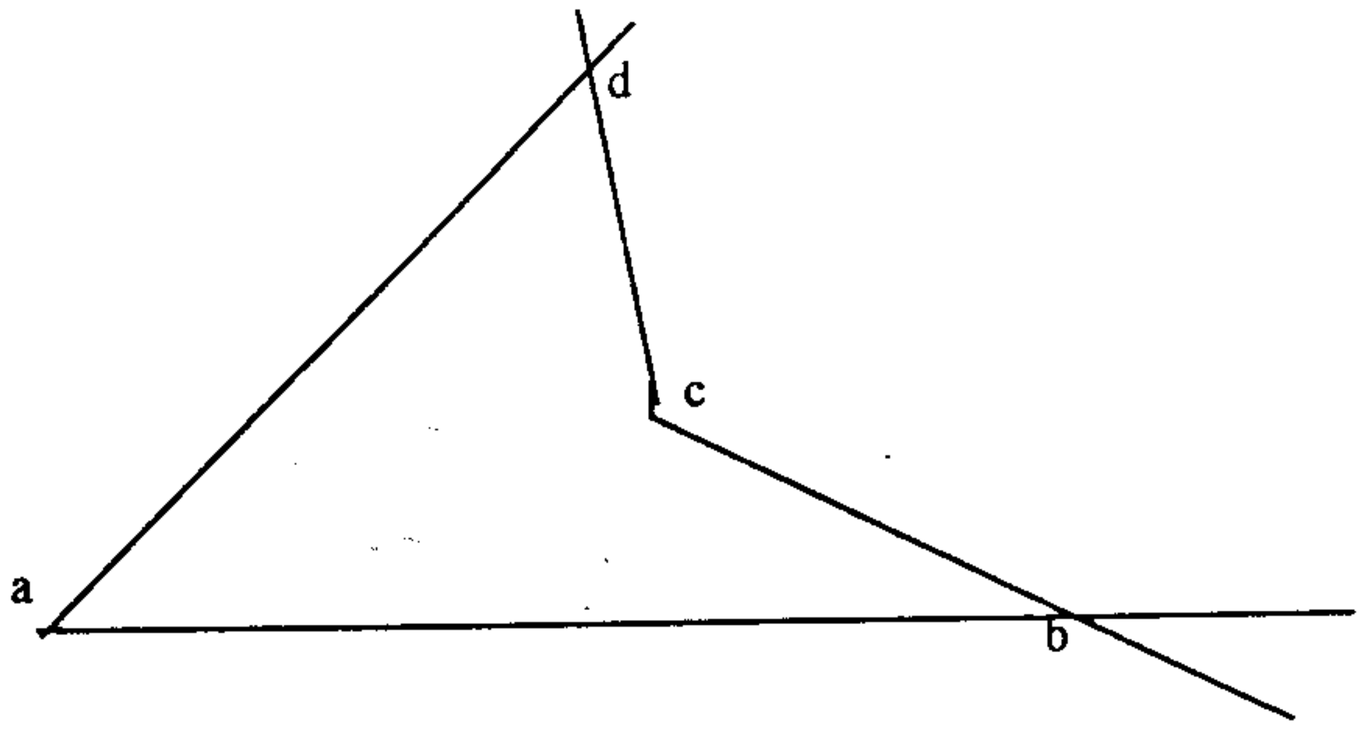

Fig. 5

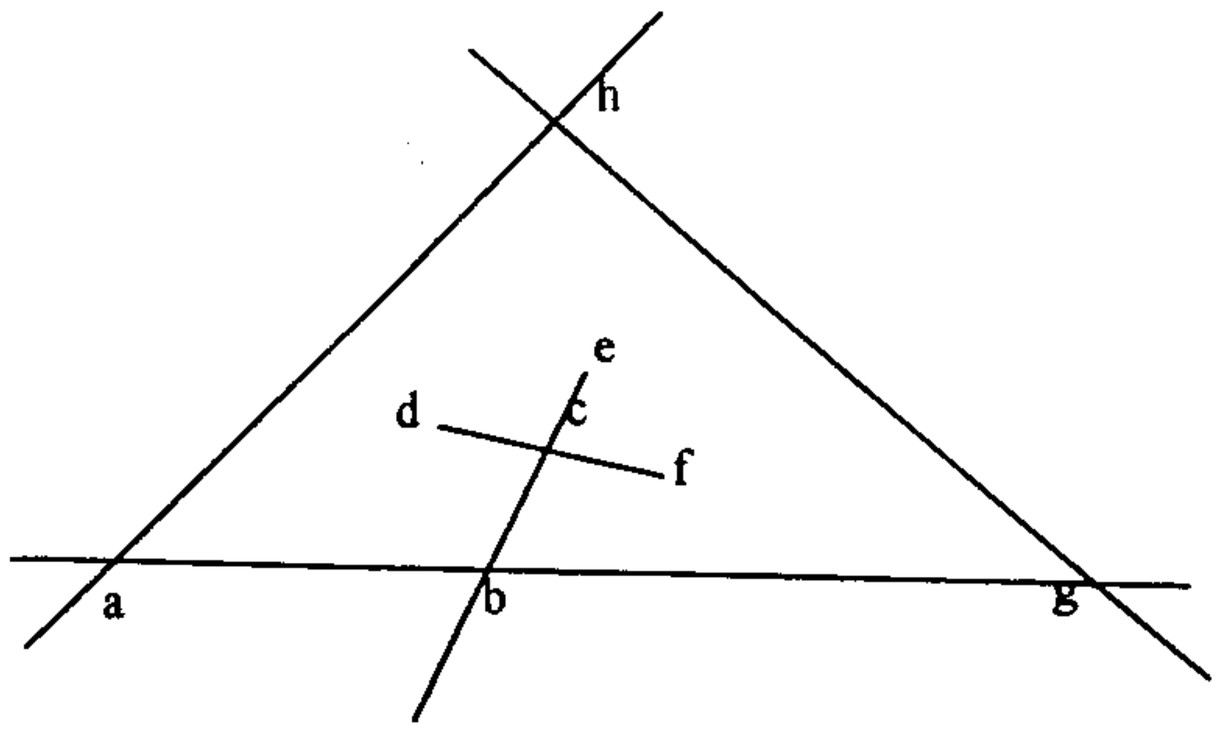

Fig. 6

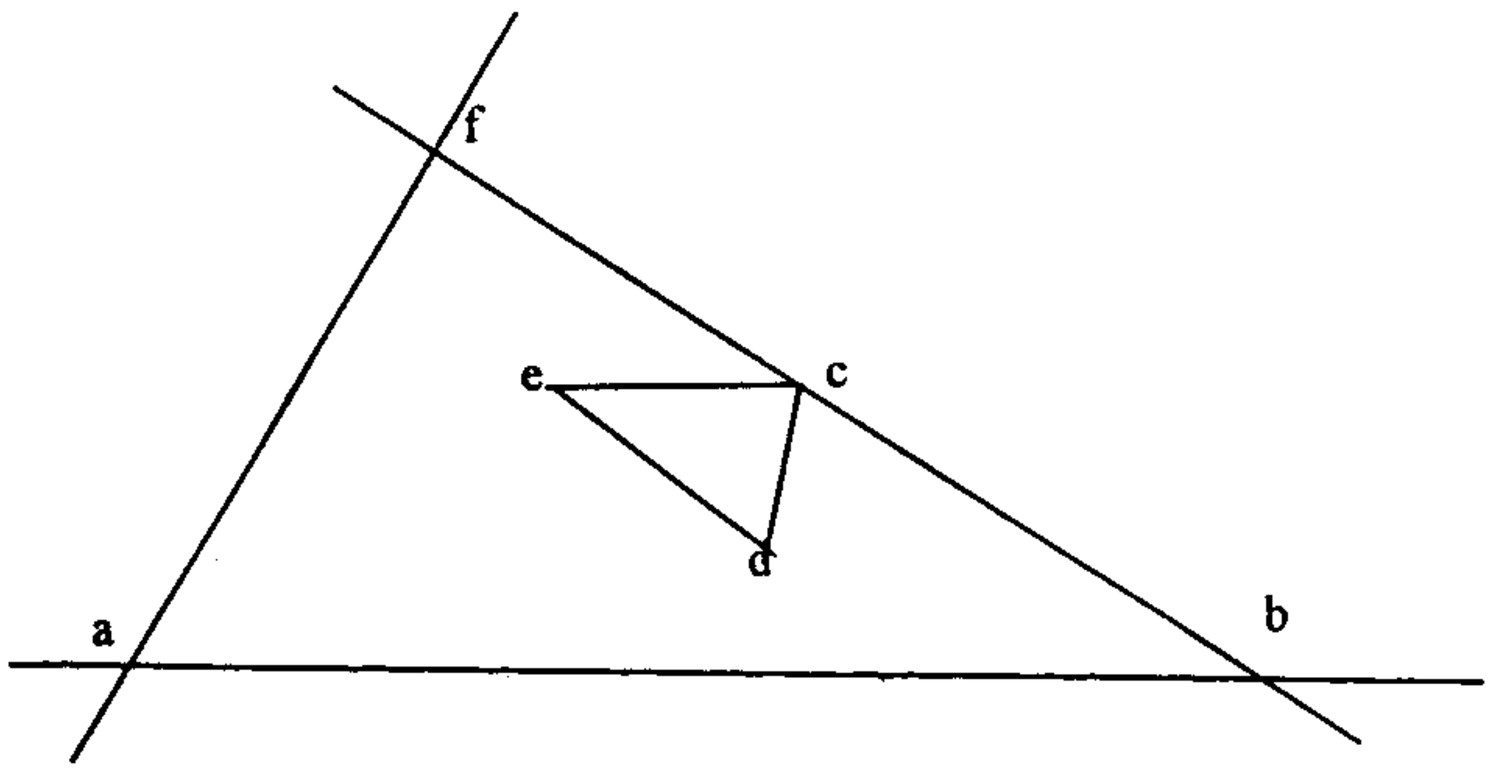

Fig. 7

(2) Continue along the same line.

(3) Right move.

(4) Turn around.

To understand the situation leading to option (2) we look at Fig. 8.

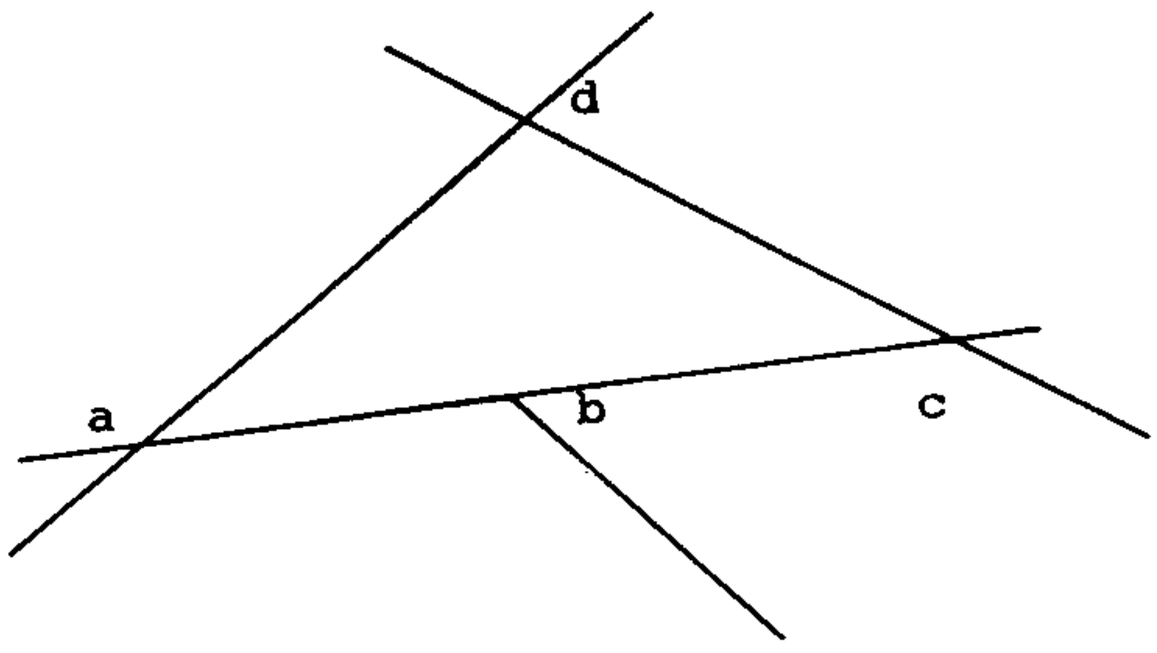

Fig. 8

In Fig. 8 the polygon is < abcda >. However at vertex b there is no possible left move. To traverse the polygon we must allow the algorithm to choose c on the same line as next vertex.

Even with all the modifications discussed so far, there still remains some problems. Consider Fig. 9 below.

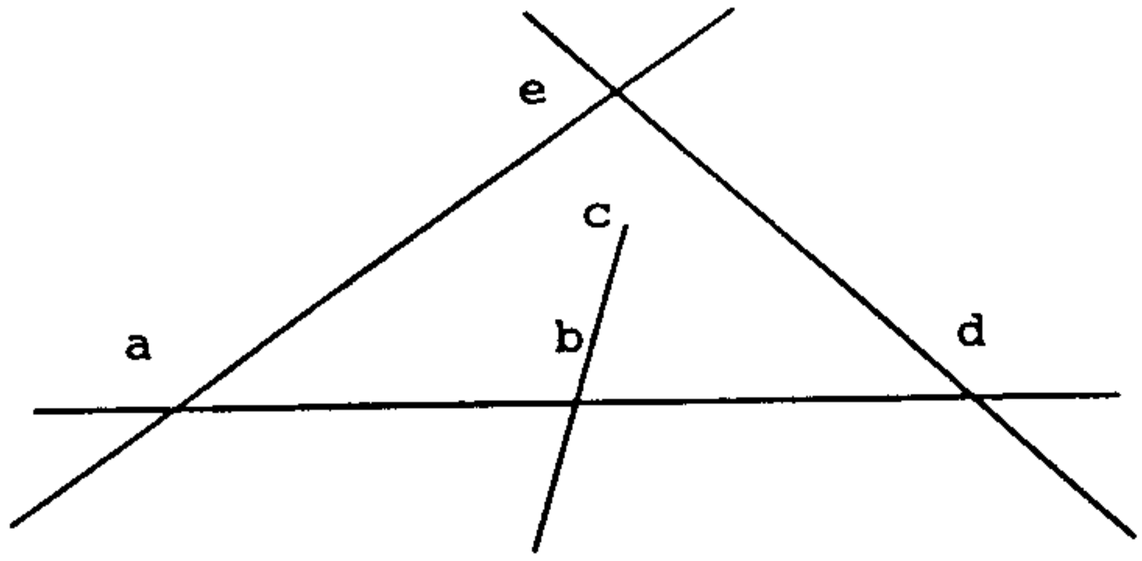

Fig. 9

If we allow all the traversals to continue, then the traversals starting from points a (towards b)and b (towards c) come up with the correct lists of vertices < abcbdea > and < bcbdeab >. But the traversal starting from b towards d comes up with < bdeab >. The previous convention of allowing the traversal with rightmost starting point is of no help, since it would produce < bdeab >. So we have to adopt a new approach.

Our choice is dictated by demand for the correctness, irredundancy and efficiency of computation. We can use the parallelism of the architecture to achieve this end. Since our original strategy demands that each processor positioned at intersection points starts a traversal towards its right neighbor, whenever in a process has to choose a right neighbor, it can be sure that another process has started computing from that point and since all processes use same strategy to choose next vertex, the later process is traversing the same path as the one that the former process would have followed if it continued further. Thus, if several processors are positioned in a same traversal, each of them computes a part of the traversal before encountering a vertex from which it has to choose a right neighbor as the next vertex. If the processors stop further traversal then, we have a chain of processors each having partial vertex list of the polygon. If these lists can be somehow merged into a single list then we can have the vertex list of the whole polygon. For example consider Fig 9. There are three processors traversing the polygon abcbdea. The processor starting at a towards b computes < ab >, processor starting at b towards c computes < bcb > and processor starting at b towards d computes < bdea >. If we could somehow merge these three vertex lists we can obtain the complete vertex list of the polygon.

However, to do this we have to this first we have to set up some convention which dictates unambiguously which one of the processors should be entrusted with the compilation of the final vertex list and secondly we have to devise a mechanism by which the chosen one can know which lists are to be merged.

The first problem can be solved by establishing the convention that only the rightmost (leftmost will do as well) processor in the processor chain will compile the complete vertex list.

Addressing the second problem is tricky one. We define a data structure Processor_info{ int, int, fPoint} and each processor maintains a data structure Identity{ l_on, l_with, position } of this type. Each processor also maintains a queue of Processor_info. Processors can read (but not delete) from the front of their own queue and write at the rear of any other processor's queue.

As soon as a processor chooses a right move it abstains from further traversal since the processor positioned at the next vertex is traversing the next part. Instead, it inserts its identity to the queue of the next vertex. Then it reads its own queue and inserts the information read to the queue of the next vertex, until it gets its own identity back from the queue. Since identity of each processors in the traversal passes through the queue, it can compare the positions of all the processors in the chain with its own position. If all processor positions in the chain are in left to its own position it knows that it is responsible for compiling the complete list. The processors whose vertex lists are to be used are already in proper order in the queue. So all it has to do is to append the lists of the processors in the queue to its own list in the reverse order of their appearance in the queue.

4.5.2

An example:

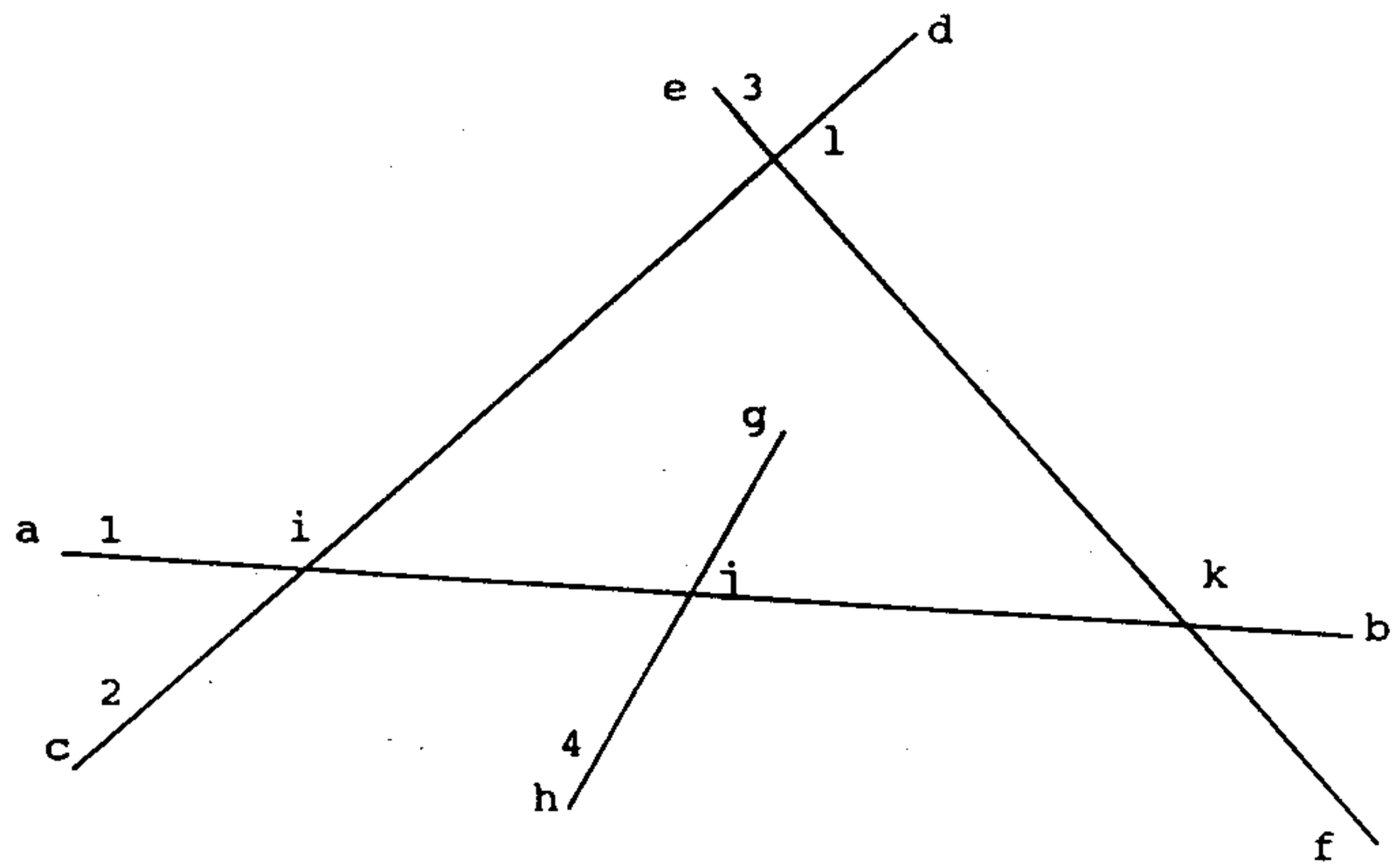

Fig. 10

In Fig. 10 there are 4 lines constituting the polygon ijgjkli. Three processors P_{12}, P_{41} and P_{14} forms the chain of processors detecting the polygon. Let us examine the vertex lists computed by the processors and their queues.

P_{12} : vertex list: < i >
queue: (1, 4, j)
(4, 1, j)
(1, 2, i)

P_{41} : vertex list: < jg >
queue: (1, 2, i)
(1, 4, j)
(4, 1, j)

P_{14} : vertex list < jkl >
queue: (4, 1, j)
(1, 2, i)
(1, 4, j)

In the example the point j is the rightmost vertex and starting vertex of both P_{14} and P_{41}. In such case we resolve by choosing the processor having largest value of l_on. This criterion allows P_{41} to produce the complete vertex list. As per entries of its queue it will take the vertex lists < jg >, < jkl > and insert before< i > (in the given order) and produce the complete vertex list < jgjklij>.

Though the measures developed above enables us to detect all legitimate polygons, one unwanted polygon, we call it the outer polygon is also detected. In Fig.10 the outer polygon is < ileldlkbkfkjhjiciai > whose detection involves the processors P_{21}, P_{23}, P_{32}, P_{13} and P_{31}. However, the computation of this polygon can be avoided by observing the fact that this polygon incudes the leftmost intersection point and from this point it traverses along the line with largest gradient value. The leftmost point can be detected by detecting the leftmost element of the first column of matrix a[][] after the sorting step is over and the line of largest gradient value can be detected using corresponding entry of the matrix b[][]. Then we can set the ipoints[][] entries in the row corresponding to the line, with leftmost intersection to INVALID. In that case the processors those would have started traversal from the point along the line, can not start, also the other processors in the chain will report failure.

4.5.3

Algorithm 2:

We need a few more data structures in addition to those used in Algorithm 1. They are as follows,

(1) Vertex_info:(int line_from, int line_to, fPoint vertex,int Status)

// this structure contains informations about a vertex of a polygon. line_from contains the no. of line along which the vertex is reached, line_to is the no. of the line along which the next vertex is to be reached, vertex is the co-ordinates of the vertex, Status contains the information abut the type of the move made at the vertex (it may have values OK (left move or continuation along the line), RTURN (right move) and TURNA (turn around)).

(2) Processor_info: (int l_on, int l_with, fPoint position)

// this structure contains the information about a processor (its position in the processor array and the co-ordinate of the intersection point to which it correspond (position).

(3) Vertex_info vlist

// a list of structure Vertex_info that will be produced as the vertex list of a polygon detected. The informations stored in the structures can be used for postprocessing the list. All the operations on a list as given before algorithm 1 are applicable to vlist, except the operation Append. Here Append on vlist is redefined as

Append(vlist, int a, int b, fPoint v, int flag)

This operation creates a Vertex_info structure with l_from = a, l_to = b, vertex = v and Status = flag. Then it is appended to the vlist.

(4) Processor_info Queue

// this is a queue of the structure Processor_info. Each processor maintains its own queue. Following operations can be performed on the queue.

(i) ReadQ(): each processor can read its own queue using this operation. It returns the element next to the one last accessed. It does not perform remove the element.

(ii) WriteQ(int i, int j, Processor_info info): a processor can insert the structure info at the rear of the queue of processor P_{ij}.

(iii) RearQ(): a processor uses it to know whether the last ReadQ() operation accessed the element at the rear of its queue.

An additional procedure is used in this algorithm.

Invalidate_outerpolygon(): employs all P_{ii} processors to compute the leftmost intersection point and the line through the point with largest value of slope, then invalidates all the entries in the row corresponding to the line having leftmost intersection.

Algorithm 2 is given below.

Algorithm 2:

```
Each processor has the following local variables,
int l_on, l_with
fPoint position
Processor_info Identity

Input: An array of structure Line of length n lines[n].
Output: If a processor succeeds to detect a polygon, it outputs a
list of vertices( Vertex_info ) vlist.

Step 1: Initialization.
          for i, j = 1 to n all processors P_ij do in parallel
               l_on ← i
               l_with ← j
               ipoints[l_on][l_with] ← INVALID
               inb[l_on][l_with] ← INVALID
               l_on(Identity) ← l_on
               l_with(Identity) ← l_with

Step 2: Computation of intersection points.
          for i, j = 1 to n and i ≠ j all processors P_ij do in
          parallel
              position ← Intersection(lines[ i ], lines[ j ])
              ipoints[l_on][l_with] ← position
              position(Identity) ← position

Step 3: Sorting the intersection points on a line.
          for i = 1 to n all processors in i-th row do in parallel
              Sort(i)

Step 4:  Computing the neighbors, only the diagonal processors
         P_ii are in work.
           for i = 1 to n all processors P_ii do in parallel
               Compute_neighbors(i)
               Invalidate_outerpolygon()
```

Step 5: Detecting the polygons.

```
for i, j = 1 to n and i ≠ j all processors P_ij do in parallel
    Detect_polygon()
```

```
procedure Detect_Polygon()( modified )
   {
   if( position = INVALID ) // checkpoint (1)
      abort                 // no intersection point to start with

   int passerOnly ← FALSE;
   int makeList ← TRUE;

   llist ← SearchCAM(l_on, position);
   if ( First(llist) ≠ l_with )  // checkpoint (2)
       {
        passerOnly ← TRUE;
        l_with(Identity) ← First(llist);
        position(Identity) ← ipoints[l_on][First(llist)];

   // the processor does not start traversal, however it acts as a
dummy transit point for the messages coming to its queue and transfer
the messages to the  queue of the processor that starts the traversal
that it was to start. Its Identity is also changed.
           }
      if( NOT passerOnly)
        {
        int notFound ← FALSE;

        int rightMove ← FALSE;

        int turnAround ← FALSE;

        fPoint start_v ← position;
                  // starting vertex of the polygon to be traversed

        fPoint curr_v ← start_v;
                  // the vertex upto which traversal is over now

   line_no curr_l ← l_on;
                  // the line along which traversal will proceed
                  // from curr_v

   line_no prev_l ← 0;
                  // the line leading to current vertex

   line_no pnext_l;
```

```
            // prospective next line on the traversal

fPoint next_v ← rightN(inb[l_on][l_with]);
            // the next vertex to be traversed, initialized with
            // the right neighbor of start vertex

if( next_v = INVALID )
   abort;
            // if the starting vertex has no right neighbor
            // i.e. start_v is the right end point of curr_l

fPoint pnext_v ← next_v;
            // prospective next vertex, initialized with next vertex

do
  {
  if(turnAround)
    {
    Append(vlist,prev_l,curr_l,curr_v,TURNA);
    turnAround ← FALSE;
    }
  else if(rightMove)
    {
    Append(vlist,prev_l,curr_l,curr_v,RTURN);
    rightMove ← FALSE;
    }
 else
  Append(vlist,prev_l,curr_l,curr_v,OK);
                // build up vertex list

  fPoint right_v ← INVALID;
        // possible next vertex in case of right move

  line_no right_l;
        // the line to turn to after right move

  llist ← SearchCAM(curr_l, next_v);

  if(Empty(llist))      // checkpoint (3)
     abort;             // next_v is the leftmost intersection

  int i ← First(llist);
  int notFound ← FALSE;
  while( notFound = FALSE )
       {
       Neighb nnb ← inb[i][curr_l];
       // nearest neighbors of next vertex on the
```

```
// intersecting line

if(leftN(nnb) ≠ INVALID)
  {
  if(Isleft(curr_v,next_v,leftN(nnb)))
// check whether left neighbor is on the left of
// current line

    {
    pnext_v ← leftN(nnb);
    pnext_l ← i;
    rightMove ← FALSE;
    break;
    }
  else
    {
    if(rightN(nnb) ≠ INVALID )

// if left neighbor is not on left side of current
// line and right neighbour exists then it is on the
// left side of current line

      {
      pnext_v ← rightN(nnb);
      pnext_l ← i;
      rightMove ← FALSE;
      break;
      }
    else

// right neighbour does not exist, hence possibility
// of a right move is imminent

      {
      if(right_v ≠ INVALID)

// there already exists a candidate for possible
// right move

        {
  if(Find_lmost(curr_v,next_v,right_v,leftN(nnb))=1)

// choose between right_v and leftN(nnb) as per
// leftmost move strategy
          {
          right_v ← leftN(nnb);
          right_l ← i;
```

```
                }
              }
            else
      // no previous candidate for right move
              {
              right_v ← leftN(nnb);
              right_l ← i;
              }
              rightMove ← TRUE;
              }
            }
          }

        else
          {
          if(Isleft(curr_v,next_v,rightN(nnb)))

         // if there is no left neighbor, and right neighbor
         // is on the left of current line

            {
            pnext_v ← rightN(nnb);
            pnext_l ← i;
            rightMove ← FALSE;
            break;
            }
         else

        // left neighbour does not exist, right neighbour is not
        // in on left of curr_l, hence possibility
        // of a right move is imminent

            {
            if(right_v ≠ INVALID)

        // there already exists a candidate for possible
        // right move

              {

          if(Find_lmost(curr_v,next_v,right_v,rughtN(nnb))=1)

        // choose between right_v and nnb.RN as per `leftmost
        // move' strategy
                  {
                  right_v ← rightN(nnb);
                  right_l ← i;
```

```
        }
    }
else
    {
    right_v ← rightN(nnb);
    right_l ← i;
    }
rightMove ← TRUE;
                }
            }
            if( Islast(llist) )
                notFound ← TRUE
            else
                pNext_l ← Next(llist)
        }
// end of while loop to detect first possible left move

    if(notFound)
            // no left move is possible

    {
    if(rightMove)
        {
            // a right move is possible, however first try
            // to avoid it by continuing straight along
            // curr_l

Neighb a ← inb[curr_l][right_l];
            // curr_v is one of the neighbors

if((leftN(a) ≠ INVALID) AND (rightN(a) ≠ INVALID))
    {
            // both neighbors exist, so continuation along the same
            // line is possible
    if( curr_v = leftN(a))
            // next vertex is right a right neighbour, so stop
            // traversal
             break;
            }
        else
            {
            pnext_v ← leftN(a);
            pnext_l ← curr_l;
            rightMove ← FALSE;
            }
        }
else
```

```
{
      // next_v is an endpoint of curr_l and a right
      // move is to be done

pnext_v ← right_v;
pnext_l ← right_l;
}
  }
else
  {
        // next_v is an endpoint of curr_l and not an
        // intersection point, hence turn around
  pnext_v ← curr_v;
  pnext_l ← curr_l;
  turnAround ← TRUE;
  }
}
```

// at this point the first possible left move or if no left move is possible then possible continuation or right move or turn around is detected. If a left move is detected and there are several other possible left moves (intersection of more than two lines), the left most move is detected in following step.

```
while(notFound = FALSE)
        // finds other prospective next vertices and
        // determine true prospective next vertex conforming
        // with `leftmost move' strategy

      {
      i ← Next(llist);
      Neighb pnnb ← inb[i][curr_l];
      if(leftN(pnnb) ≠ INVALID)
        {
        if(Isleft(curr_v,next_v,leftN(pnnb)))
          {
        fPoint opnext_v ← leftN(pnnb);
        int opnext_l ← i;
        if( Find_lmost(curr_v,next_v,pnext_v,opnext_v) = 2)
          {
          pnext_v ← opnext_v;
          pnext_l ← opnext_l;
          }
        }
      else if( rightN(pnnb) ≠ INVALID)
        {
        fPoint opnext_v ← rightN(pnnb);
```

```
            int opnext_l ← i;
            if( Find_lmost(curr_v,next_v,pnext_v,opnext_v) = 2)
              {
              pnext_v ← opnext_v;
              pnext_l ← opnext_l;
              }
            }
        }
      else if(Isleft(curr_v,next_v,rightN(pnnb)))
            {
            fPoint opnext_v ← rightN(pnnb);
            int opnext_l ← i;
            if( Find_lmost(curr_v,next_v,pnext_v,opnext_v) = 2)
              {
              pnext_v ← opnext_v;
              pnext_l ← opnext_l;
              }
            }
          if(Islast(llist))
            notFound ← TRUE;
        }

// end while. If a left move is possible, then the proper left move
is chosen at this point.

        curr_v ← next_v;
        next_v ← pnext_v;
        prev_l ← curr_l;
        curr_l ← pnext_l;
        }while( next_v ≠ rightN(inb[curr_l][prev_l]));

      // traversal of the part of the polygon designated to the processor
is over. Now it has to determine whether it should construct the complete
vertex list.
    }           // end if

      if(passerOnly)
      {
      do
       {
        If(RearQ())
            wait;
        Processor_info temp ← ReadQ();
        WriteQ(l_on,First(llist),temp);
        }while( temp ≠ Identity );
       }
else
```

```
{
WriteQ( curr_l,prev_l,Identity);
 if(ReaarQ())
    wait();
 Processor_info temp ← ReadQ();
 while(temp ≠ Identity)
        {
        if( position(temp) > position)
          makeList ← FALSE;
        if(position(temp) = position AND makeList)
          if(l_on(temp) > l_on)
            makeList ← FALSE;
        WriteQ(curr_l,prev_l,temp);
         temp ← ReadQ();
         }
   if(makeList)
     Makelist();   // Compiling the complete vertex list
   }
} // end Detect_polygon
```

The procedure Detect_polygon is fairly complex in detailed form. Let us summarize the essence of it.

procedure Detect_polygon

1 : Determine whether the processor is entitled to start a traversal
  1.1 : If the position is INVALID abort.
  1.2 : If there are several lines intersecting at the point 'position' and the processor corresponds to a l_with value that is not smallest of the column nos. in ipoints that contains the value position, restrain from traversing, but act as a passer of the information passed through the queue (pass the information to the processor that has start traversing the path same as that to be traversed by it).
2 : Traverse
  2.1 : Start from the vertex position along l_on towards the right closest neighbor on line l_on and make the latter the second vertex.
  2.2 : Choose next vertex
    2.2.1 : Try to turn left, if fail
    2.2.2 : Try to continue along the same line, if fail
    2.2.3 : Try to turn right, if fail
    2.2.4 : Try to turn around, if fail
    2.2.5 : Abort

3 : Stop traversal
  3.1 : Stop traversal as soon as the right vertex of the current vertex is chosen as the next vertex.

4 : Decide whether to compile the complete vertex list

4.1 :Circulate the identities of the processors computing the partial list through the processor chain by means of the queues.

4.2 :Determine the relative position.

4.2.1 :Determine whether the position is rightmost of the positions of the processors in the chain, if so,

4.2.2 :If there are several rightmost processor, determine whether it has largest value of l_on among them.

4.3 :If both the tests (4.2.1) and (4.2.2) results positive, compile the complete vertex list using the partial vertex lists computed by the processors listed in the queue (in the reverse order of appearance in the queue).

4.5.4

Analysis:

Algorithm 2 differs from algorithm 1 mainly due to the modifications made in the procedure Detect_polygon. Let us examine whether the modifications causes any change of time complexity.

Since the procedure is allowed to detect degeneracy as well as non-convexity of polygons, it can traverse same vertex (in case of degeneracy) more than once, however not more than once by coming along a particular line. Hence if a vertex is the intersection of i lines constituting a degenerate part of the polygon, traversing the degenerate part takes O(i) time and this part will not be traversed again. So degeneracy does not increase the run time beyond O(n).

Since non-convexity is allowed, when we choose one line out of (i-1) at an intersection of i lines, we can not be sure that none of the lines rejected here will never appear in the traversal at a latter stage. However to reach such a line again a traversal must take atleast one right move, thereby exhausting the possibility of reappearance of another line in the traversal. Also, every reappearance of a line in the traversal exhausts the possibility of reappearance in the traversal of atleast one other line. This clearly shows that the run time of the procedure is O(n).

The algorithm uses one additional procedure Invalidate_outerpolygon, which involves a parallel procedure of choosing one leftmost element from a set of n elements using n processors. So this can be accomplished in $O(\log_2 n)$ time. The possible difference of complexity of two algorithms can arise only due to complexity of the modified Detect_plygon procedure. Let us concentrate on it.

So, the time complexity of algorithm (2) is also O(n).

Since we have used n^2 processors, the cost of the algorithms is $c(n) = O(n^3)$.

Therefore, the algorithms are optimal.

Chapter 5

About further improvement

To investigate the scope of further improvement we shall examine each step of the algorithm

Step (1) and (2) takes O(1) time. So we need not worry about them.

Step (3) takes O(n) time. That is the time required by any known parallel sorting algorithm for n elements employing n processors. Since in our architecture no spare processor is available to apply an algorithm using more processors, no further speed-up is possible in this step.

Step (4) involves computing the neighborhood information of the intersection points. One processor computes the neighborhood information of all (a maximum of (n-1)) intersection points on a line in O(n) time. However we can employ (n-1) processors parallely for each line each one computing the neighborhood information of one intersection point. In the best possible case when all the intersection points on a line are distinct the computation is over in O(1) time, But in worst-case when all the n lines intersect at the same point, each processor has to search through the whole sorted list of intersection point. This will again take O(n) time. But one can expect some reasonable speed-up of the step in average cases by employing the parallelism proposed above.

In step (5) the most important part of the algorithm, detecting the polygons comes into play. Every off-diagonal processor that has a valid position value, seeks to find a polygon. At the beginning each of them knows the lines whose intersection point it has calculated and the intersection point itself (the position). It can further access its four closest neighbors (but not the lines intersecting at those points) on the two lines in O(1) time. This is due to distributed nature of the computation. Though all the information needed to trace a polygon is out there in the shared memory accessible to each processors, to make use of the information each processor has to follow some clue in each step. This leads to spending of O(n) time for detecting a polygon(as discussed in the previous chapter). To speed-up the procedure we have to make more information accessible at a time to a processor. That requires more organized representation of the information which in turn requires more intensive preprocessing, that further adds up to the cost.

The Detect_polygon procedure can be written in elegant recursive form, though the time complexity remains same.

An outline of the recursive version is given below.

```
procedure Detect_polygon()( Recursive )
     {
      Startvertex ← position
      curr_v ← Startvertex
      next_v ← right neighbour of Startvertex

      Traverse_polygon(curr_v,next_v)

      Finish_processing()
     }

procedure Traverse_polygon(curr_v,next_v)
     {
      if ( stop_condition )
         return
      Append(vlist, curr_v)
      pnext_v ← Next_vertex()
      curr_v ← next_v
      next_v ← pnext_v
      Traverse_polygon(curr_v,next_v) . . . Recursive call
     }
```

The stop_condition and the procedures Next_vertex() and Finish_processing() differs for algorithm (1) and algorithm (2).

For algorithm (1),
stop_condition : next_v = Startvertex
Next_vertex() searches for next vertex to be reached by leftmost move strategy. If it fails to find one, terminates the procedure Detect_polygon..
Finish_processing simply appends the curr_v and next_v to vlist to complete the vertex list of the polygon detected.

For algorithm (2),
stop_condition : the next vertex chosen is a right neighbour.
Next_vertex() searches the next vertex to move to (i) by leftmost move, if fails (ii) by continuation along the line, if fails (iii) by right move, if fails (iv) by turning around, if fails terminates the procedure Detect_polygon.
Finish_processing() determines whether the processor is the one to compile the complete vertex list, if so, compiles the complete vertex list.

Chapter 6

Discussions on usefulness

The algorithms can be used in some pattern recognition scheme involving the recognition of a scene comprised of polygonal figures. After the low level image processing for detecting the lines is over the end points of the lines can be supplied to these algorithms. Then these algorithms can be utilized for intermediate level processing (feature extraction).

The algorithms developed here can be easily modified to extract information about the nature of the polygons detected (whether a square or rectangle or isosceles triangle etc.) and the adjacency relation among them. Based on these data it is easy to design a grammar with the basis polygons as primitives (terminals of the grammar) and the description of a scene can be represented as a sentence in the language generated by the grammar. When trying to recognize an unknown scene, the system requires to produce a description of the scene in terms of the primitive polygons and their adjacency information recovered by the algorithm and parse it to see whether it is a valid sentence of the language.

Such schemes are extremely useful for automated industrial inspection, robot vision etc.

Though the requirement of n^2 processors for detecting the polygons created by n straight lines may look like a severe demand on the resource of the system, The architecture used for n lines can easily be used for multiples of n lines keeping the run time O(n). Hence, the scheme is entirely within range of feasibility for fairly large number of lines.

Apart from practical uses, these are the sort of algorithms whose study may provide some clue about working of human brain.

Appendix

The C++ implementation of several procedures used in the algorithms are presented below. **Also are provided** the outline of a parallel algorithm for sorting n elements with n procwssors.

The header file containing the definitions of the data structures used and the operations allowed **on them**.

```
#ifndef _OBJ.HPP
#define _OBJ.HPP
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include <graphics.h>


#ifndef NULL 0
#define NULL 0
#endif
#ifndef TRUE 1
#define TRUE 1
#endif
#ifndef FALSE 0
#define FALSE 0
#endif
#define SQR(x) x*x

typedef int line_no;


struct Point{
        int x,y;

Point();
Point(int,int);
Point(Point&);

Point operator+(Point&);
Point operator-(Point&);
Point operator*(float);
void operator=(Point&);
int operator<(Point&);    // p<q means p is in left of q
```

```
int operator>(Point&);
int operator==(Point&);
int operator<=(Point&);
int operator>=(Point&);
float operator~();              //norm operator
int operator%(Point&);     //dot product operator

inline int valid();
void Setpoint(int,int);
        void Shpoint();
        void Drawpoint(int color) {    putpixel(x,y,color); }
        };

struct fPoint{
        float x,y;

        fPoint();
        fPoint(float,float);
        fPoint(fPoint&);

        fPoint operator+(fPoint&);
        fPoint operator-(fPoint&);
        fPoint operator*(float);
        void operator=(fPoint&);
        int operator<(fPoint&);   //p<q means p is in left of q
        int operator>(fPoint&);
        int operator==(fPoint&);
        int operator<=(fPoint&);
        int operator>=(fPoint&);
        float operator~();              //norm operator
        float operator%(fPoint&); //dot product operator

        inline int valid();
        void Setfpoint(float,float);
        void Shpoint();
        void Drawpoint(int color) {    putpixel(x,y,color); }
        };

struct Line{
        line_no l;
        Point El,Er;

        Line(){l=0;};
        Line(line_no,Point,Point);
        Line(Line&);

        void Setline(line_no,Point,Point);
```

```
        void Shline();

        };


struct I_nb{
        fPoint LN,RN;
        int start_Flag;
        int dup_Flag;

        I_nb() { start_Flag = dup_Flag = FALSE; }
        I_nb(const fPoint&,const fPoint&,int,int);
        I_nb(I_nb&);
        int valid();
        void Setnb(const fPoint&,const fPoint&,int,int);
        void Set_start(int F) { start_Flag = F; }
        int Check_start() { return start_Flag; }
        };


struct V_info{
        line_no l_from,l_to;
        Point vertex;
        int status_Flag;

        V_info();
        V_info(line_no,line_no,Point,int);
        V_info(const V_info&);

        void Setvert(line_no,line_no,Point,int);
        };


#endif
```

The routine for computing intersection of two lines.

```
void Find_intersection(Line L1,Line L2,fPoint& q)

        // intersection of two lines L1 and L2 is computed and assigned
        // to q (INVALID in case of no intersection

   {
   float m1,m2,c1,c2;
   int delx1,delx2,dely1,dely2;
   fPoint p;
   fPoint l1,r1,l2,r2;


   delx1=L1.Er.x - L1.El.x;
   dely1=L1.Er.y - L1.El.y;
   delx2=L2.Er.x - L2.El.x;
   dely2=L2.Er.y - L2.El.y;

   if( (delx1*dely2 - dely1*delx2) == 0)  return;
               // lines are parallel
   l1.Setfpoint(float(L1.El.x),float(L1.El.y));
   r1.Setfpoint(float(L1.Er.x),float(L1.Er.y));
   l2.Setfpoint(float(L2.El.x),float(L2.El.y));
   r2.Setfpoint(float(L2.Er.x),float(L2.Er.y));

   if(delx1==0)
               // L1 is vertical
    {
    m2 = float(dely2)/float(delx2);
    c2 = float(L2.El.y) - m2*float(L2.El.x);
    p.x = float(L1.Er.x);
    p.y = m2*float(L1.Er.x) + c2;
    if((p >= l1) && (p >= l2) && (p <= r1) && (p <= r2))
   {
   q = p;
   return;
    }
     else
      return;
     }
else if(delx2==0)
               // L2 is vertical
  {
  m1 = float(dely1)/float(delx1);
  c1 = float(L1.El.y) - m1*float(L1.El.x);
  p.x = float(L2.Er.x);
```

```
p.y = m1*float(L2.Er.x) + c1;
if((p >= l1) && (p >= l2) && (p <= r1) && (p <= r2))
{
q = p;
return;
}
else
  return;
  }
            // none is vertical
m1 = float(dely1)/float(delx1);
c1 = float(L1.El.y) - m1*float(L1.El.x);
m2 = float(dely2)/float(delx2);
c2 = float(L2.El.y) - m2*float(L2.El.x);
p.x = (c2-c1)/(m1-m2);
p.y = m1*(c2-c1)/(m1-m2) + c1;
if((p >= l1) && (p >= l2) && (p <= r1) && (p <= r2))
  {
  q = p;
  return;
  }
else
    return;

} // end find_intersection
```

Routine to compute whether the point p is in the left half-plane of the line joining c_pt and n_pt, returns TRUE if so, FALSE otherwise.

```
int Check_left(fPoint c_pt,fPoint n_pt,fPoint p)
   {
   fPoint a = n_pt - c_pt;     // checks whether p is on lhs of line c_pt
                               // to n_pt
   fPoint b = p - c_pt;
   if((a.x*b.y - b.x*a.y) > 0.0)
      return TRUE;
   else
     return FALSE;
   }
```

The following routine checks which one of the points p1 and p2 corresponds to more leftward move from point n_pt with respect to the line joining points c_pt and n_pt. It returns 1 if p1 is selected, otherwise returns 2.

```
int Find_lmost(fPoint c_pt,fPoint n_pt,fPoint p1,fPoint p2)
  {
  fPoint a = c_pt - n_pt;      //decides which one of p1 & p2 corre-
                               //sponds to leftmost
  fPoint b = p1 - n_pt;           //turn at n_pt coming from c_pt
  fPoint c = p2 - n_pt;
  float FACTOR = 180.0/(4*atan(1.0));
  float theta1 = float(FACTOR * acos(double((a % b)/((~a)*(~b)))));
  float theta2 = float(FACTOR * acos(double((a % c)/((~a)*(~c)))));
  if(theta1 < theta2)
    return 1;
  else
    return 2;
  }
```

The routine to compute the neighborhood information of the intersection points on a line.

```
void Compute_neighbor()
          // the sorted list of the intersection points on the line pline
          // is in the array b from b[1] onwards, b[0] containing the
          // left endpoint of pline. Here n is the no. of intersection
          // points on pline. b[n+1] contains the right end point. The
          // routine constructs an array nbs[] of neighborhood
          // information and assigns the array to the row of matrix inb
          // corresponding to the line pline
{
fPoint ln,curr,rn;
fPoint INVALID;

I_nb* nbs = new I_nb[l_count];
ln = b[0];
curr = b[1];
if(ln == curr)
  ln = INVALID;
               // if left neighbour is an intersection
k = 2;
rn = b[k];
while( rn == curr && k < n)
  rn = b[++k];
               // if leftmost intersection point is
               // intersection of more than two lines

if(rn == curr )
  rn = INVALID;
               // if there is only one intersection point
               // and that is at right end point
```

```
nbs[c[1] - 1].Setnb(ln,rn,FALSE,FALSE);
for(int l=2;l<n;l++)
   {
   if (!(curr == b[l]))
            // next intersection point is distinct

      {
       ln = curr;
       curr = b[l];
       while(curr == rn && k < n)
            rn = b[++k];
       if(rn == curr)
          rn = INVALID;
     nbs[c[l] - 1].Setnb(ln,rn,FALSE,FALSE);
       }
   else
     nbs[c[l] - 1].Setnb(ln,rn,FALSE,TRUE);
           // storing neighborhood information

   }   //end for

     inb[pline.l - 1] = nbs;
               // array of neighborhood info for pline
               // is added to the table `inb'


     }
  }

  }       // end select_neighbors
```

Most popular of the parallel sorting algorithms using n processors to sort n elements is odd-even transposition. The idea is as follows, n prosessors are in a linear array, the n elements to be sorted are in an array a[].

Step 1 : For i 1 to n all processors P_i do in parallel
Read a[i]

Step 2 : For i 1 to n all processors P_i do in parallel
For j 1 to n in step of 1
if j is odd
Every odd numbered processor compares its element with that of its right neighbor and exchanges elements if the right one is smaller.

```
                If j is even
                        Every even numbered processor compares its
                      element with that of its right neighbor and
                       exchanges elements if the right one is smaller.

Step 3 : For i 1 to n all processors P_i do in parallel
                    Write a[i]
```

This will produce the n elements sorted in ascending order in the array a[].