Speeding up the Skeletonization of Binary Patterns using the Homogeneous Multiprocessor

Helmut Beffert

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

January 1988

ISBN 0-315-41605-X

ABSTRACT

Speeding up the
Skeletonization of Binary Patterns
using the Homogeneous Multiprocessor

Helmut Beffert

A modification is proposed to speed up the Safe Point Thinning Algorithm (SPTA), which was already shown to be faster than 14 other known skeletonization algorithms [24]. The modified algorithm has been implemented on a single processor. It has also been implemented on a simulator for the Homogeneous Multiprocessor Proper using two techniques: data decomposition and function decomposition. Experimental results show that with our modification and multiprocessor implementations, the SPTA was speeded up by 66.2 percent when using eight processors.

Acknowledgements

I would first like to thank my supervisor Dr. Rajjan Shinghal for his help and patience throughout the thesis. Especially for all the time that he spend in the last few months of this thesis.

I would like to thank Dr. J.W. Atwood for his suggestions on selecting a multiprocessor environment.

I would also very much like to thank Dr. K. Li for his help with the simulator for the Homogeneous Multiprocessor.

I am also very grateful to my friends Marc LaFleur and Lorne Mill for allowing me to discuss many of my ideas with them. Thank you also to Henry Polley for helping out on some of the diagrams.

Finally I would like to thank my family and friends for their support which greatly helped in the realization of this thesis.

Table of Contents

Chapter I

Introduction

I.1. Review of Skeletonization

The skeleton of a binary pattern is a thinned line drawing, which ideally should preserve the connectedness and shape of the original pattern [26]. For example, Figure I.1 illustrates a binary pattern and its skeleton. The skeleton of a pattern is not necessarily unique. As an example, Figure I.2 shows two different skeletons for the pattern of Figure I.1a. Ideally, the original pattern should be thinned to its medial axis. Skeletonization reduces the memory space required for storing the essential structural information of a pattern. It simplifies the data structures required in processing the pattern [4]. Many skeletonization algorithms retain sufficient information about the original pattern so that an almost exact copy of the original pattern can be reconstructed.

Skeletonization algorithms can be divided into two general classifications, which are referred to as peeling algorithms and shelling algorithms.

Peeling algorithms [24] consist of iteratively deleting edge points (that is, changing dark points along the edges of a pattern to white points) until the pattern is thinned to a line drawing. To retain the connectedness and shape of the original pattern, we should take care that in deleting edge points we 1) do not delete end points (informally speaking end points are dark points at the open extremities of a stroke); 2) do not break the connectedness of the pattern; and 3) do not cause excessive erosion (for example, a stroke is not iteratively deleted).

Shelling algorithms [2] consist of measuring the distance that each dark point is from the edges of the pattern. The dark points farthest from the edges of the pattern are kept to form the skeleton (such points are sometimes called local maxima). To retain the connectedness and shape of the original pattern, 1) some points may need to be added to the skeleton so as to connect the local maxima; and 2) some points may need to be deleted where the selection of local maxima created lines of width greater than one.

Peeling algorithms are far more popular than shelling algorithms. Peeling algorithms often require more iterations than shelling algorithms to obtain the final skeleton. However, this cost is offset by the relative simplicity of the iterations as compared to those of the shelling algorithms. Peeling algorithms are thus more useful for patterns of relatively thin lines such as those found in optical character recognition algorithms, where the number of iterations will be small. Shelling algorithms, which usually require a fixed number of iterations, work better for image analysis where the patterns tend to have thicker lines. Furthermore peeling algorithms are much easier to parallelize since they consist of repeatedly performing similar operations on each point in the pattern. For this reason a peeling based algorithm was chosen for our multiprocessor implementation in this thesis.

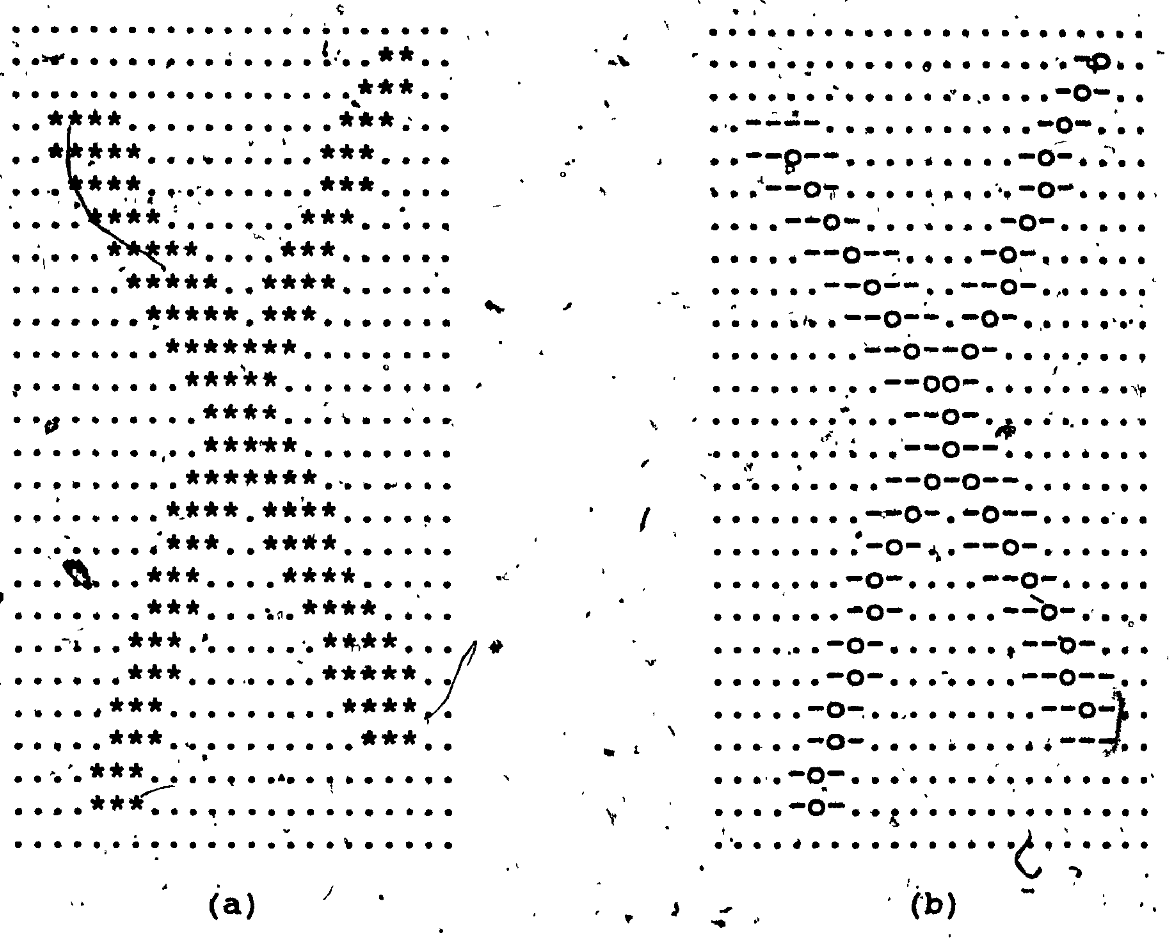

Figure I.1. (a) a sample pattern and (b) its skeleton. A '*' is an original dark point; and a '.' is an original white point. A 'o' is a point that is part of the skeleton; and a '-' is a point that was deleted from the original pattern.

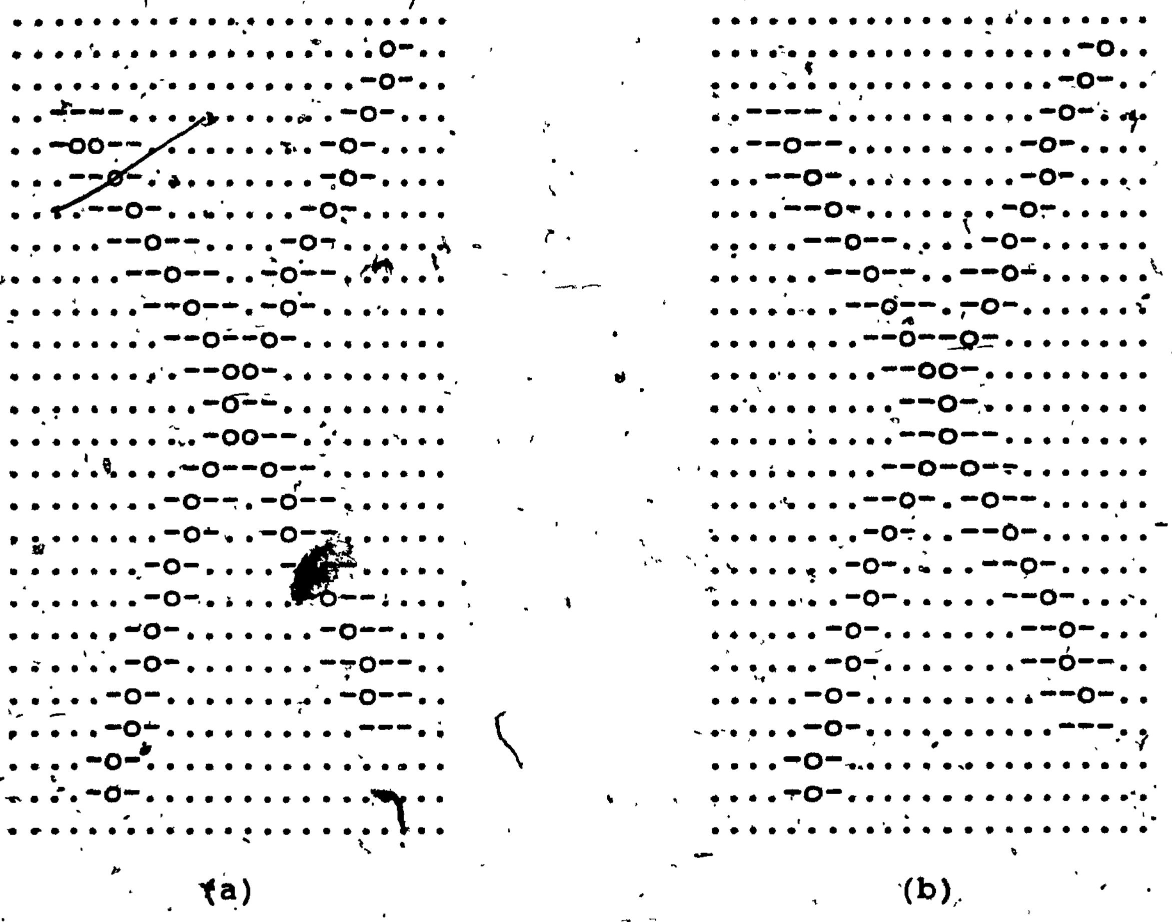

Figure I.2. Two different skeletons for the pattern shown in Figure I.1. These skeletons were both obtained using the modifed SPTA described in Chapter II. The differences in the skeletons are a result of varying the scanning sequences. The skeleton marked (a) was obtained while scanning from left to right and the skeleton marked (b) was obtained while scanning from right to left.

I.2. Why use the SPTA

Naccache and Shinghal [24] proposed a peeling based skeletonization algorithm called SPTA (Safe Point Thinning Algorithm). They experimentally showed that the SPTA was faster than 14 other known skeletonization algorithms. Furthermore, the SPTA produced skeletons that had reconstructibility.

In this thesis, we propose a modification to the SPTA to further speed it up, without sacrificing reconstructibility. SPTA, as originally proposed [24], worked on a single processor. In this thesis, we also propose two different implementations of the modified SPTA on the Homogeneous Multiprocessor Proper [6] and [21]. The objective of these implementations was to examine the further speeding up of the SPTA.

I.3. Outline of the Thesis

In this thesis, we present a historical review of skeletonization algorithms on both single processor and multiprocessor architectures. We then propose a modification to the SPTA on a single processor as well as two multiprocessor implementations of the SPTA. All these implementations are experimentally shown to improve the performance of the SPTA.

In Chapter II, we review some thinning algorithms which have been implemented on single processor architectures. We then review the SPTA and present our modified SPTA. We also present a formal description of the modified SPTA using algorithmic pseudo code.

In Chapter III, we review some parallel thinning algorithms which have been proposed for multiprocessor architectures. We then discuss several actual multiprocessor implementations including those on the CLIP4 and PASM multiprocessors.

In Chapter IV, we describe the Homogeneous Multiprocessor. We then show that our modified SPTA is suitable to run on this architecture. Finally, we present our two multiprocessor implementations, the function decomposition implementation and the data decomposition implementation. Once again we also present a formal description of each of the implementations using algorithmic pseudo code.

In Chapter V, we present our experimental results obtained from the original SPTA and our modified SPTA, as well as the results of our two multiprocessor implementations.

In Chapter VI, we briefly describe another multiprocessor, the Connection Machine. We then propose an implementation of the SPTA for the Connection Machine and discuss our expected results for this implementation.

Finally, in Chapter VII we give our conclusion and discuss some possible future extensions to our research.

Chapter II.

Review of Thinning Algorithms on a Single Processor

II.1. Definitions

Before we review some thinning algorithms, we establish our notation required for the review. We present the notation required by both peeling and shelling based thinning algorithms.

In a pattern, the 8-neighbours of a point p are defined to be the 8 points adjacent to p (points n_0 to n_7 in Figure II.1). Points n_0, n_2, n_4, and n_6 are called the 4-neighbours of p.

An edgepoint is formally defined to be a dark point that has at least one white 4-neighbour. There are four kinds of edgepoints: left, right, top, and bottom. A left (right) edgepoint is defined to have its left (right) neighbour n_4 (n_0) white. Similarly, a top (bottom) edgepoint is defined to have its top (bottom) neighbour n_2 (n_6) white. Note that an edgepoint may be of more than one kind; for instance, a dark point that has neighbours n_2 and n_4 white will be both a left edgepoint and a top edgepoint.

In an 8-distance transformation, each dark point in the pattern is labelled by a number which indicates the length of the shortest path from that point to its nearest white neighbour (using any adjacent 8-neighbours to generate the path). Figure II.2 shows a sample pattern and its corresponding 8-distance transformation. Similarly, the 4-distance transformation of a point p is the length of the shortest path from the point p to its nearest white neighbour (using only adjacent 4-neighbours to generate the path).

A local maximum is a point with a label which is either equal to or greater than the labels of all the points in its neighbourhood. The neighbourhood can be comprised of either the 8-neighbours or only the 4-neighbours, and usually corresponds to the type of distance transformation that was applied to the pattern. Figure II.3 shows the local maxima of the pattern shown in Figure II.2.

n_3	n_2	n_1
n_4	p	n_0
n_5	n_6	n_7

Figure II.1. A point p and its neighbourhood.

```
................
.......****.....
......*******...
.....*********..
....**********..
...******..***..
...****.........
...***..........
..****..........
..***...........
..***...........
..****..........
..*****...****..
...************.
....***********.
....***********.
.......***......
................
```

(a)

```
................
.......1111.....
......1122111...
.....112222211..
....1122111121..
...112111..111..
...1211.........
...121..........
..1111..........
..121...........
..121...........
..1211..........
..11211...1111..
...112111112211.
....12222222221.
....11112111111.
.......111......
................
```

(b)

Figure II.2. (a) a sample pattern, and (b) the 8-distance transformation of the sample pattern.

```
................
.......----.....
......--oo---...
.....--ooooo--..
....--oo----o-..
...--o---..---..
...-o--.........
...-o-..........
..----..........
..-o-...........
..-o-...........
..-o--..........
..--o--...----..
...--o-----oo--.
....-ooooooooo-.
....----o------.
.......---......
................
```

Figure II.3. A pattern showing the local maxima (that is all points labelled 'o').

II.2. Review of some Thinning Algorithms

Over the years, many thinning algorithms have been implemented on single processor architectures. We feel that an extensive review of these algorithms is not necessary. Several good reviews of many of these algorithms have already been written [22] and [23]. However a short review is presented showing several typical and well-known algorithms.

Pavlidis [25] and [26] proposed a peeling based thinning algorithm which tests for four different kinds of edgepoints (left, right, top, and bottom). Each pass consists of four scans. During each scan only one type of edgepoint is tested. During a pass, a dark point is flagged only if it satisfies all of the following conditions:

1) the point is an edgepoint
2) the point is not an end-point
3) the neighbourhood of the point does not match any of the three windows shown in Figure II.4.

The algorithm terminates when no dark points have been flagged during a given pass.

Many algorithms similar to the one illustrated above have been proposed and implemented. The basic concepts are the same, but the windows used for determining which edgepoints are to be flagged usually vary [5],[24], and [35].

Arcelli [2] and [3] proposed a shelling based thinning algorithm which requires only three passes to skeletonize a pattern. First the 8-distance transformation is calculated for each dark point in the pattern. Next all dark points which are symmetrically placed within the pattern, including all local maxima, are assigned to the skeleton. That is, all points that satisfy at least one of the three conditions illustrated in Figure II.5. Then all dark points p, which have a neighbour n_k already in the skeleton and a value greater than that n_k are added to the skeleton as long as they satisfy at least one of the following two conditions :

a) n_k is a 4-neighbour of p

b) n_k is not a 4-neighbour of p, and
neither of the 4-neighbours of p which are adjacent to n_k have a value which is equal to the value of p.

Next all dark points that have all four of their 4-neighbours in the skeleton are also added to the skeleton. The skeleton may now contain lines with a thickness greater than one, so a one-pass thinning operation is performed. See Figure II.6 for details.

Several other shelling based thinning algorithms have also been proposed [27], [32], [35], and [35] which also skeletonize patterns in a fixed number of passes. However like the algorithm shown above, their computations are also complicated making these algorithms less desirable for small patterns such as those in optical character recognition that we are using. Furthermore, the computations are very irregular, thus making it more difficult for a multiprocessor implementation.

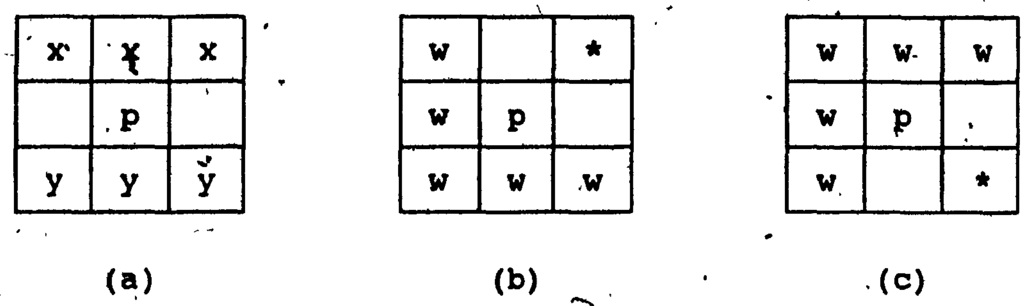

Figure II.4. For Pavlidis' algorithm, if the neighbourhood of a right edgepoint p does not match any of the three windows shown above then the point p is flagged. For top, left, and bottom edgepoints, the above windows are rotated counter clockwise by 90, 180, and 270 degrees respectively.

A '*' indicates a dark point and a blank indicates a white point. A 'w', 'x', or a 'y' indicates either a white point or a dark point as long as at least one 'x' and at least one 'y' is a dark point.

a) $$\sum_{k=1}^{8} |q_k - q_{k+1}| \geq 4$$

b) $$\sum_{k=1}^{4} (r_{2k-1} - r_{2k-1} \cdot r_{2k} \cdot r_{2k+1}) \neq 2$$

c) $$\sum_{k=1}^{4} (r_{2k-1}) \neq 2$$

using the following window

n_2	n_3	n_4
n_1	p	n_5
n_8	n_7	n_6

where $q_9 = q_1$ and $r_9 = r_1$

and where

$$q_k = \begin{cases} 1 & \text{if } n_k = p - 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$r_k = \begin{cases} 1 & \text{if } n_k = p \\ 0 & \text{otherwise} \end{cases}$$

for $k = 1$ to 8

Figure II.5. In Arcelli's algorithm, if a dark point satisfies any of the three conditions a, b, or c shown above, then it is assigned to the skeleton.

a) $\sum_{k=1}^{8} (r_k) > 3$ and

$$(q_1 \cdot q_3 + q_3 \cdot q_5 + q_5 \cdot q_7 + q_7 \cdot q_1) = 0$$

b) $(q_1 \cdot q_5 \cdot z) = 1$

where

$$q_k = \begin{cases} 1 & \text{if } n_k < 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$r_k = \begin{cases} 1 & \text{if } 0 < n_k \leq p \\ 0 & \text{otherwise} \end{cases}$$

and

$$z = \begin{cases} 1 & \text{if } n_7 > Z \\ 0 & \text{otherwise} \end{cases}$$

Figure II.6. In Arcelli's algorithm, if a point on the skeleton satisfies either of the above two conditions then it is kept, otherwise it is deleted from the skeleton.

II.3 Review of the SPTA

In essence, the SPTA consists of executing a few scans over the pattern where in each scan some edge points are flagged. If in a given scan, an edgepoint is not flagged, then it is declared to be a safepoint. Figure II.7 illustrates the evaluation of a point p during a scan. The scanning sequence may be either row-wise or column-wise at the user's choice. We adopted the row-wise scanning sequence.

There are two kinds of scans : a LR-scan (left-right) and a TB-scan (top-bottom).

The LR-scan flags the following kinds of points:
[1] all left edgepoints whose boolean expression S_4 is TRUE, where

$$S_4 = n_0 \cdot (n_1 + n_2 + n_6 + n_7) \cdot (n_2 + \bar{n}_3) \cdot (n_6 + \bar{n}_5).$$

A boolean variable has the value TRUE when its corresponding point is dark and unflagged, (that is, it is either an original dark point, or a safepoint) and it has the value FALSE otherwise, (that is, if the point is flagged or white.) The above boolean expression was derived from the four windows shown in Figure II.8. The justification for how this boolean expression was derived is given by Naccache et al [24].

[2] all right edgepoints whose boolean expression S_0 is TRUE, where

$$S_0 = n_4 \cdot (n_5 + n_6 + n_2 + n_3) \cdot (n_6 + \bar{n}_7) \cdot (n_2 + \bar{n}_1).$$

The TB-scan flags the following kinds of points:

[1] all top edgepoints whose boolean expression S_2 is TRUE, where

$$S_2 = n_6 \cdot (n_7 + n_0 + n_4 + n_5) \cdot (n_0 + \bar{n}_1) \cdot (n_4 + \bar{n}_3)$$

[2] all bottom edgepoints whose boolean expression S_6 is TRUE, where

$$S_6 = n_2 \cdot (n_3 + n_4 + n_0 + n_1) \cdot (n_4 + \bar{n}_5) \cdot (n_0 + \bar{n}_7)$$

The LR-scan and the TB-scan are executed alternately. It is the user's choice to commence skeletonizing by either first executing the LR-scan or the TB-scan. Without loss of generality, we have assumed that the LR-scan is executed first. Then a LR-scan followed by a TB-scan constitutes a pass over the pattern. Naccache et al. explained why the two scans per pass cannot be merged into one scan.

All points flagged during a given pass are considered to be deleted before the next pass begins. If no points are flagged during a pass, then the SPTA terminates. The skeleton then consists of all points that were declared to be safepoints during any of the passes, (that is, all points with a label greater than ZERO). Table II.1 illustrates all possible values that a point can have using the Single

Integer Labelling Technique (SILT) proposed by Naccache et al. [24].

The termination criterion as described above has one inefficient characteristic. Since in the last pass, the SPTA flags no points, we can say that it is in effect a do-nothing pass. It would be more efficient if we could avoid executing a do-nothing pass as far as possible.

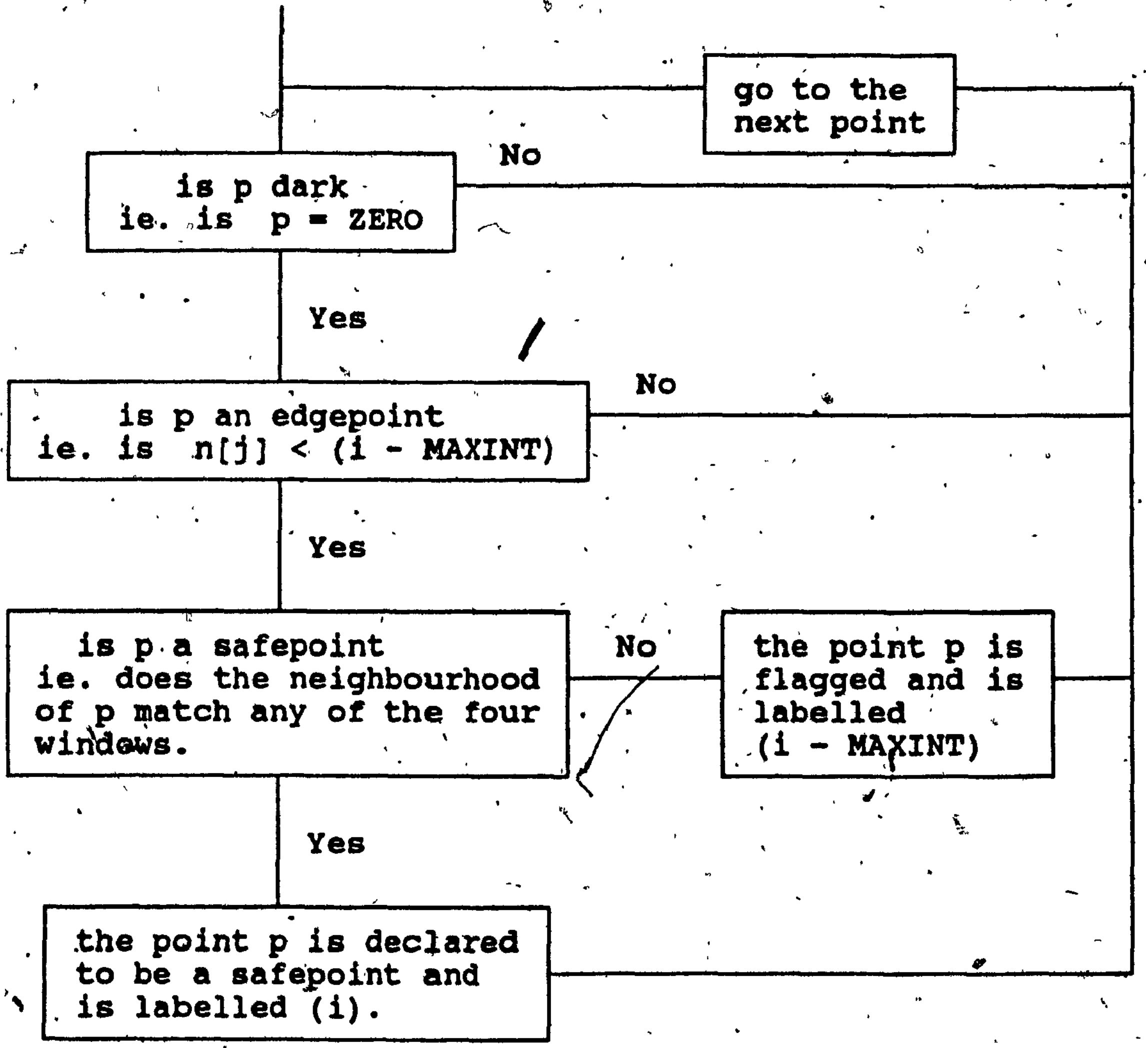
go to the next point
No
is p dark ie. is p = ZERO
Yes
No
is p an edgepoint ie. is n[j] < (i - MAXINT)
Yes
is p a safepoint ie. does the neighbourhood of p match any of the four windows.
No
the point p is flagged and is labelled (i - MAXINT)
Yes
the point p is declared to be a safepoint and is labelled (i).

Figure II.7. The evaluation process for a point p. This process is performed once for each point during every scan. All labelling of points is done using the Single Integer Labelling Technique (SILT).

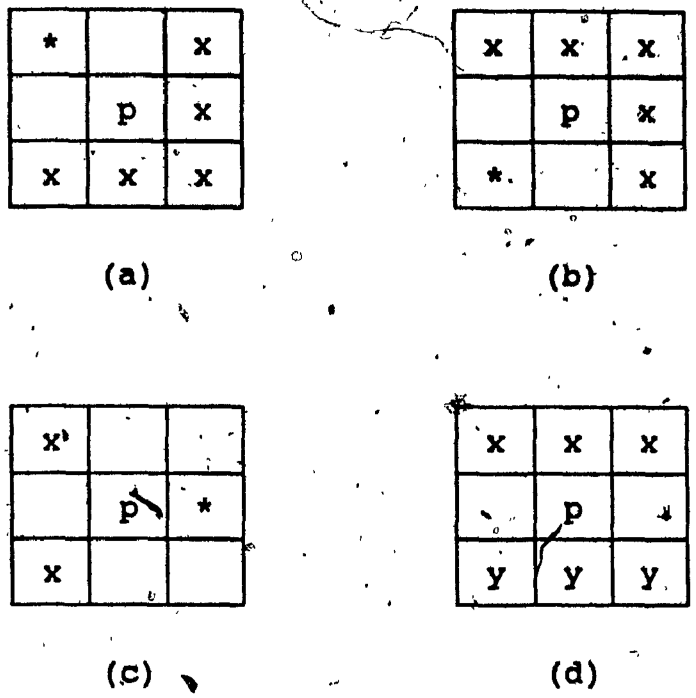

Figure II.8. The four windows used to test whether a left edgepoint is to be
1) flagged, that is the neighbourhood of the point p does not match any of the four windows shown above; or
2) declared to be a safepoint, that is the neighbourhood of the point p matches with at least one of the four windows shown above.

The points labelled by '*' indicate a dark point, the points labelled by x and y may be either dark or white, and a point that is not labelled indicates a white point.

Table II.1.

When using SILT, a point p can be in any one of the following states during pass number i.

value of the point p	description
(- MAXINT)	an original white point
less than (i - MAXINT) and greater than (- MAXINT)	a point that was deleted during a previous pass
(i - MAXINT)	a point that was deleted during the current pass
ZERO	an original dark point
1 < p < i	a point declared to be a safepoint during a previous pass
i	a point declared to be a safepoint during the current pass

II.4 Our Modified SPTA

The modified SPTA is identical to the original SPTA except for an enhancement to the termination criterion. This enhancement was made so as to avoid executing the do-nothing pass. We will therefore only present the new termination criterion here, as well as the complete algorithmic pseudo code for the modified SPTA.

Let us define a variable d_k, whose value at the end of the k^{th} ($k \geq 1$) scan is equal to the number of dark points that are neither flagged nor declared to be safepoints.

We propose below two criteria, named as criterion1 and criterion2, to test for termination.

1) criterion1 : If at the end of the k^{th} scan, d_k is equal to ZERO then the algorithm terminates. This implies that the pattern contains only flagged points or safepoints.

This criterion however fails when we have a configuration such as that shown in Figure II.9 (a dark point whose 4-neighbours are all safepoints). Since safepoints are never deleted, the point p of Figure II.9. will never be deleted either. Therefore d_k will never become ZERO. The configuration shown in Figure II.9 usually occurs at the intersection of strokes. See the example shown in Figure II.10. So we need criterion2, given below.

2) criterion2 : If at the end of the kth scan, d_k is equal to d_{k-2} then the algorithm terminates. That is to say that no new points were either flagged or declared to be safepoints in the last two scans. When terminating under this criterion, the algorithm does execute a do-nothing pass.

To summarize, we say that at the end of a scan, if criterion1 or criterion2 is TRUE, then the SPTA terminates. Extensive experimentation has shown us that 95 percent of the time the algorithm terminated under criterion1, thus avoiding a do-nothing pass. Hence it was only 5 percent of the time that the algorithm performed the do-nothing pass. In other words, with our proposed terminating criteria the SPTA performed approximately two fewer scans than the SPTA originally proposed in [24]. To remove any ambiguity in our informal description above, we present a formal description of the modified SPTA.

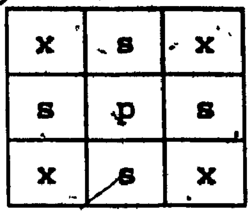

Figure II.9. Point p is a dark point which is neither flagged nor a safepoint; s's are safepoints, x's may be either dark or white points. If a configuration such as above exists, then criterion1 of Section II.4. fails in terminating the SPTA.

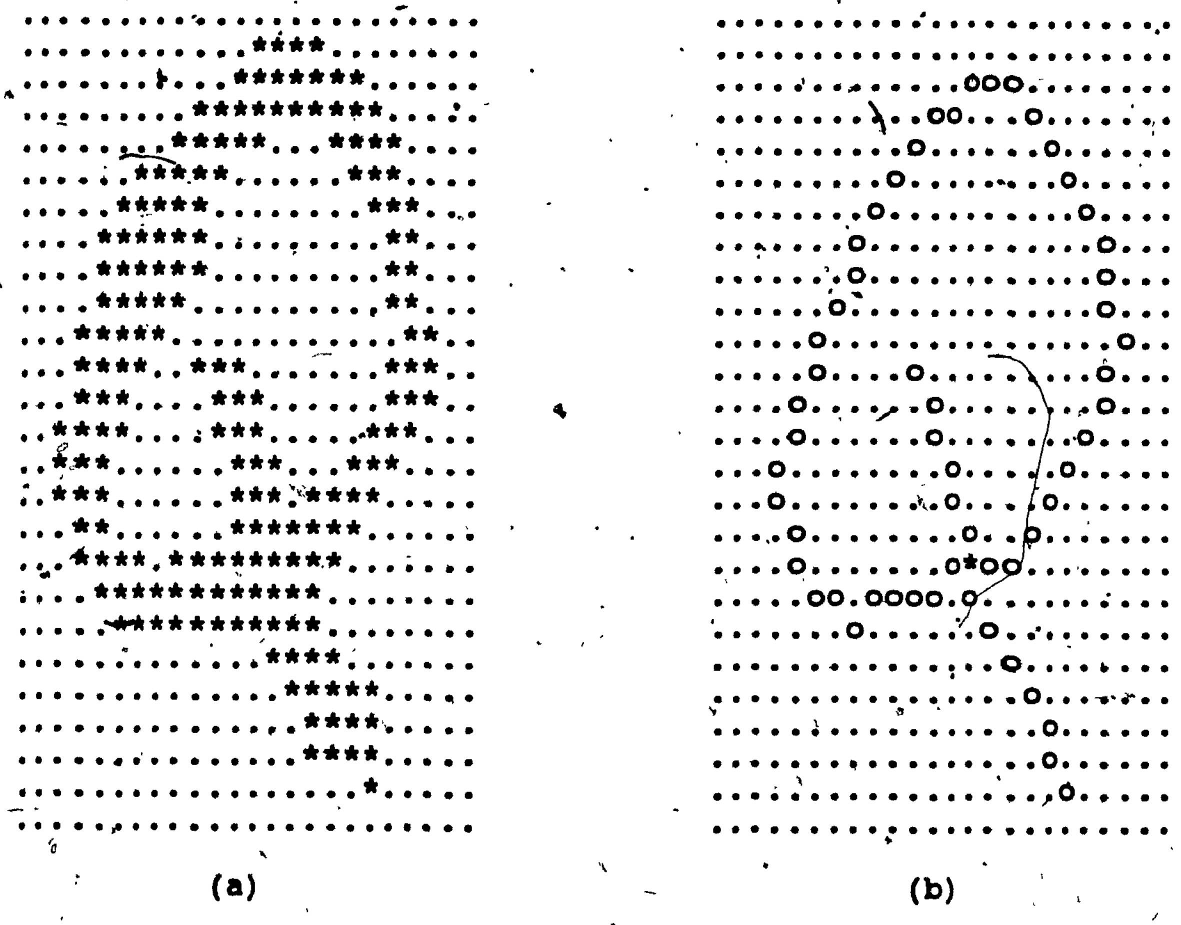

Figure II.10. (a) A specimen pattern that has intersecting strokes.
(b) In the skeleton, the SPTA did not terminate under criterion1 of Section II.4 because of the presence of a dark point '*' with its 4-neighbours as safepoints. Hence the SPTA terminated under criterion2.

```
        The following data structures are required for the
        different implementations of the SPTA.  They are all
        presented here for completeness so that only one copy
        of the declarations needs to be made. )

Declaration of types :

pat_type      = array[1..MAXROW,1..MAXCOLUMN] of integer;
        ( The data structure used for storing the pattern,
          where
            MAXROW    - indicates the total number of rows
                        in the pattern, and
            MAXCOLUMN - indicates the total number of
                        columns in the pattern. )

count_type    = array[-1..MAXSCAN] of integer;
        ( The data structure used for storing the number of
          dark points which have neither been flagged nor
          declared to be safepoints, remaining after the
          completion of a scan.  The first scan is numbered
          1.  For the terminating criterion2, (d[k] = d[k-2],
          at the end of the first scan, k=1, we need a value
          for d[-1].  Similarly, for the end of the second
          scan, we need a value for d[0].  So d[-1] and d[0]
          are assumed to be ZERO. )

8-neighbours = array[0..7] of pointer;
        ( The data structure used for referring to the 8-
          neighbours of the point p. )

direction     = (right,left);
          ( The data structure used to define the direction
            of flow of the data in the pipeline for the
            function decomposition implementation. )

border_type   = (0,2,4,6);      ( (right,top,left,bottom) )
          ( The data structure used to define which type of
            edgepoint is being tested for. )


Declaration of variables :

PATTERN : pat_type; ( contains the pattern, where
                    0       - indicates a dark point, and
                 -MAXINT - indicates a white point. )

i         : integer;  ( iteration number, or pass number. )
```

Algorithmic Pseudo Code for implementing the modified SPTA on a single processor.

```
procedure ONE_PROCESSOR_SPTA(var PATTERN : pat_type);

var
  j : integer;     { indicates the type of scan,
                       j = 0   for LR-scan, and
                       j = 2   for TB-scan. }
  k : integer;     { contains the scan number. }
  d : count_type;  { an array containing the number of dark
                     points which have neither been flagged
                     nor declared to be safepoints, remaining
                     in the pattern upon the completion of a
                     scan. }

begin
  i := 0;          { Initialize the pass number. }

  for k := -1 to MAXSCAN do
    d[k] := 0;     { Initialize d to ZERO for each scan. }

  k := 0;          { Initialize the scan number. }

  repeat

    i := i + 1;    { Increment the pass number by one. }
    j := 0;        { Set scan type to left/right edgepoints. }
    k := k + 1;    { Increment the scan number by one. }

    SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);
                   { Execute the kth scan on the entire
                     pattern. }

    if (d[k] <> 0) and (d[k] <> d[k-2]) then
                   {  If criterion1 and criterion2 are FALSE,
                     then prepare for the next scan. }
      begin
        j := 2;    { Set scan type to top/bottom edgepoints. }
        k := k + 1; { Increment the scan number by one. }

        SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);
                   { Execute the kth scan on the entire
                     pattern. }
      end;

  until (d[k] = 0) or (d[k] = d[k-2]);
                   { Repeat executing passes on the entire
                     pattern until criterion1 or criterion2 is
                     TRUE. }
end;
```

```
procedure SKELETONIZE(var PATTERN : pat_type;
                      j,first_row,last_row : integer;
                      var d : integer);
var
  row    : integer;
  column : integer;
  p      : pointer;
           { the variable p is used when referring to the
             point PATTERN[row,column]. }
  n      : 8-neighbours;
           ( the variables n[0] to n[7] are used when
             referring to the 8-neighbours of the point p. )
  border : border_type;
           ( Indicates which 4-neighbour caused the point p
             to become an edgepoint. )
begin
  for row := first_row to last_row do
    for column := 1 to MAXCOLUMN do
      begin
        if DARK(p) then
        ( a point is considered to be DARK if it has the
          value ZERO. ie. it is not a safepoint. )

          if EDGEPOINT(n[j],n[j+4],border) then
          ( test each dark point to see if it is an
            edgepoint.

            begin
              if SAFEPOINT(n,border) then
          ( Test each edgepoint to see whether it is a
            safepoint. If it is a safepoint, then the point
            is labelled by the value i.  Otherwise, the point
            becomes a flagged point and is labelled by the
            value (i - MAXINT). )
                p := i                  ( a safepoint )
              else
                p := i - MAXINT;   ( a flagged point )
              ADJUST(p,row,column);
          ( The procedure ADJUST, will be used only by the
            data decomposition implementation.  However it
            has been included here so that only one version
            of the procedure SKELETONIZE needs to be
            presented. )

            end
          else
            d := d + 1;
          { The point is a dark point which is neither
            flagged nor declared to be a safepoint, so we
            increase our counter d. )
      end;
end;
```

(There are two reasons for labelling the flagged points and safepoints in the manner shown above.

1) The points flagged during pass i have the label (i-MAXINT), which becomes a threshhold t. In pass (i+1) all points with labels less than t are considered white. Thus we do not need to travel through the entire pattern deleting all flagged points to prepare for pass (i+1). This helps in speeding up the SPTA.

2) A safepoint declared during pass i has the label i. Since the skeleton consists of all the safepoints, the label on a point in the skeleton can help us reconstruct the original pattern if needed. Naccache et al. [24] have described how this reconstruction can be done. Therefore we are not describing it here.)

```
function EDGEPOINT(n[j],n[j+4] : pointer;
                   var border : border_type) : boolean;

     (  A point p, is considered to be WHITE if it satisfies
        the following condition :
            (value of p) < (i - MAXINT).
        That is, the point is an original white point, or it
        is a point that was flagged during a previous pass.

        The variable border, returns the value indicating
        which boolean expression S[border] should be tested,
        ( where border = 0, 2, 4, 6 ), to detect safepoints.
        )

begin
  if WHITE(n[j]) then
     (  Test for either a right or top edgepoint. )
    begin
      border := j;
      EDGEPOINT := TRUE;
    end
  else if WHITE(n[j+4]) then
     (  Test for either a left or bottom edgepoint. )
    begin
      border := j+4;
      EDGEPOINT := TRUE;
    end
  else
    EDGEPOINT := FALSE;
end;
```

```
function SAFEPOINT(n : 8-neighbours; border : border_type) :
                boolean;

    { This function evaluates the appropriate safepoint
      boolean expression and returns TRUE if the point is a
      safepoint and FALSE if it is not.

      An 8-neighbour of the point p is evaluated to TRUE if
      it has a label that is greater than or equal to ZERO,
      and it is evaluated to FALSE otherwise. }

begin
  case border of
    0 : { Evaluate for a right safepoint. }
        SAFEPOINT := not(n[4] · (n[5] + n[6] + n[2] + n[3]) ·
                     (n[6] + not(n[7])) · (n[2] + not(n[1])));
    2 : { Evaluate for a top safepoint. }
        SAFEPOINT := not(n[6] · (n[7] + n[0] + n[4] + n[5]) ·
                     (n[0] + not(n[1])) · (n[4] + not(n[3])));
    4 : { Evaluate for a left safepoint. }
        SAFEPOINT := not(n[0] · (n[1] + n[2] + n[6] + n[7]) ·
                     (n[2] + not(n[3])) · (n[6] + not(n[5])));
    6 : { Evaluate for a bottom safepoint. }
        SAFEPOINT := not(n[2] · (n[3] + n[4] + n[0] + n[1]) ·
                     (n[4] + not(n[5])) · (n[0] + not(n[7])));
  end;  { case }
end;
```

Chapter III

Review of Thinning Algorithms on Multiprocessors

III.1. Introduction to Programming on Multiprocessors

Multiprocessor architectures are becoming more and more available. Much of the multiprocessor work being done is directly related to image processing and pattern recognition applications. Several multiprocessors have been designed specifically with these applications in mind [8],[11],[12], and [30]. Furthermore, several general purpose multiprocessors have been shown to work quite well for these types of applications [6] and [14].

Multiprocessors can generally be divided into several classes :

SIMD (Single Instruction Multiple Data) stream computers usually consist of a single central host computer which broadcasts instructions to thousands of microprocessors. Each microprocessor has its own memory. All the microprocessors receive the same instructions with each microprocessor having the option to either sit idle or to execute the instruction. This type of architecture is best suited for problems where the same operations must be performed in an independent fashion on thousands of individual data items.

MIMD (Multiple Instruction Multiple Data) stream computers usually consist of tens or hundreds of processors. Each processor can execute its own instructions independently of all other processors in the machine. MIMD machines exist where there is only one central memory, or where each processor has its own memory. Still others offer a combination of the two, where each processor has some memory of its own and groups of processors can share some memory.

Although much progress has been made in multiprocessor design, the area of programming multiprocessors has lagged behind. Recently however, work has increased in multiprocessor software development due to the commercial viability of some of the multiprocessors [10]. In most cases, software development on multiprocessors is very much dependent on the type of multiprocessor being used.

In general, an algorithm implemented on a multiprocessor can be parallelized in two ways [17] and [18]:

1) function decomposition, in which the algorithm is decomposed into segments that are assigned to different processors, each processor functioning on the full data, as the data is pipelined through the processors;
2) data decomposition, in which the data is decomposed into segments that are assigned to different processors, each

processor executing the full algorithm.

Function decomposition usually works best on an MIMD architecture, while data decomposition usually works best on an SIMD architecture.

The following two factors can have a significant impact on the implementation of an algorithm [16] and [18]:

1) data granularity, which indicates the size of the datum that can be dealt with as a fundamental unit;
2) module granularity, that indicates the amount of processing which can be done without the need for synchronization.

Fine grain applications usually perform better on SIMD machines, while coarse grain applications perform better on MIMD machines.

III.2: Definitions for Parallel Thinning Algorithms

Shortly after the first thinning algorithms were developed for single processor computers, it was realized that the thinning process was quite well suited for a parallel implementation. Many of these parallel implementations [1],[25], and [28] were based on the following definition:

> the new value of a point at the ith pass can be determined by its own value and that of its 8-neighbours at the (i-1)th pass [37].

This allows all points to be evaluated in parallel within a given pass.

Since all deletable edge points are simultaneously removed in one pass, the number of passes required to thin the pattern should ideally be equal to half the maximum width of the pattern. However, most parallel thinning algorithms divide this task into a number of scans, where two or more scans represent one pass [34]. Thus the number of iterations over the pattern usually becomes equal to the maximum width of the pattern.

Many of these algorithms were never actually implemented on multiprocessor environments. It was noted by Hilditch [13] that when some of these algorithms are actually implemented on multiprocessors, the quality of the skeletons is not always as good as would be expected. This is mainly due to more than one layer of edge points being evaluated and deleted at the corners of a pattern during a single pass. Hilditch thus proposed a refinement which stated that when a pass is divided into multiple scans, that the criterion for deleting an edge point should be as follows:

> deletion should be restricted to points that not only satisfy the deletion criterion of the pattern in its current state, but also would have satisfied this condition at the start of the current pass.

Hilditch goes on to show that without this refinement, parallel algorithms will not produce proper skeletons since they tend to delete too many points at the corners of the pattern.

However, the type of parallelism described above is not always necessary. Only large SIMD based multiprocessors could benfit from such an algorithm. Several actual multiprocessor implementations have been proposed for MIMD architectures consisting of only tens of processors. Such algorithms do not necessarilly require that each point in the pattern be evaluated in parallel. Next, we will present some of the better known multiprocessors and some of the thinning algorithms that have been implemented on them.

III.3. Thinning on the CLIP4 Multiprocessor

The CLPI4 multiprocessor has an SIMD architecture [7], [8], and [9]. It consists of 1156 microprocessors. The processors are connected in a 2-dimensional grid. As shown in Figures III.1 and III.2, the processors can be connected so that each processor has either 6 connections (a hexagonal grid) or 8 connections (a square grid). These connections conform to the local windows or neighbourhoods which are often used while evaluating a point during image processing algorithms. Each processor has its own memory. There is no shared memory between processors. A PDP 11/10 was used as the host computer.

Hilditch [13] used the CLIP4 multiprocessor to test some existing parallel thinning algorithms as well as to develop a new parallel thinning algorithm. She compared a well known peeling algorithm [1] with a shelling algorithm and found that both types of algorithms performed equally well on the CLIP4 multiprocessor. However no results were given on how these implementations compared with similar implementations on a single processor.

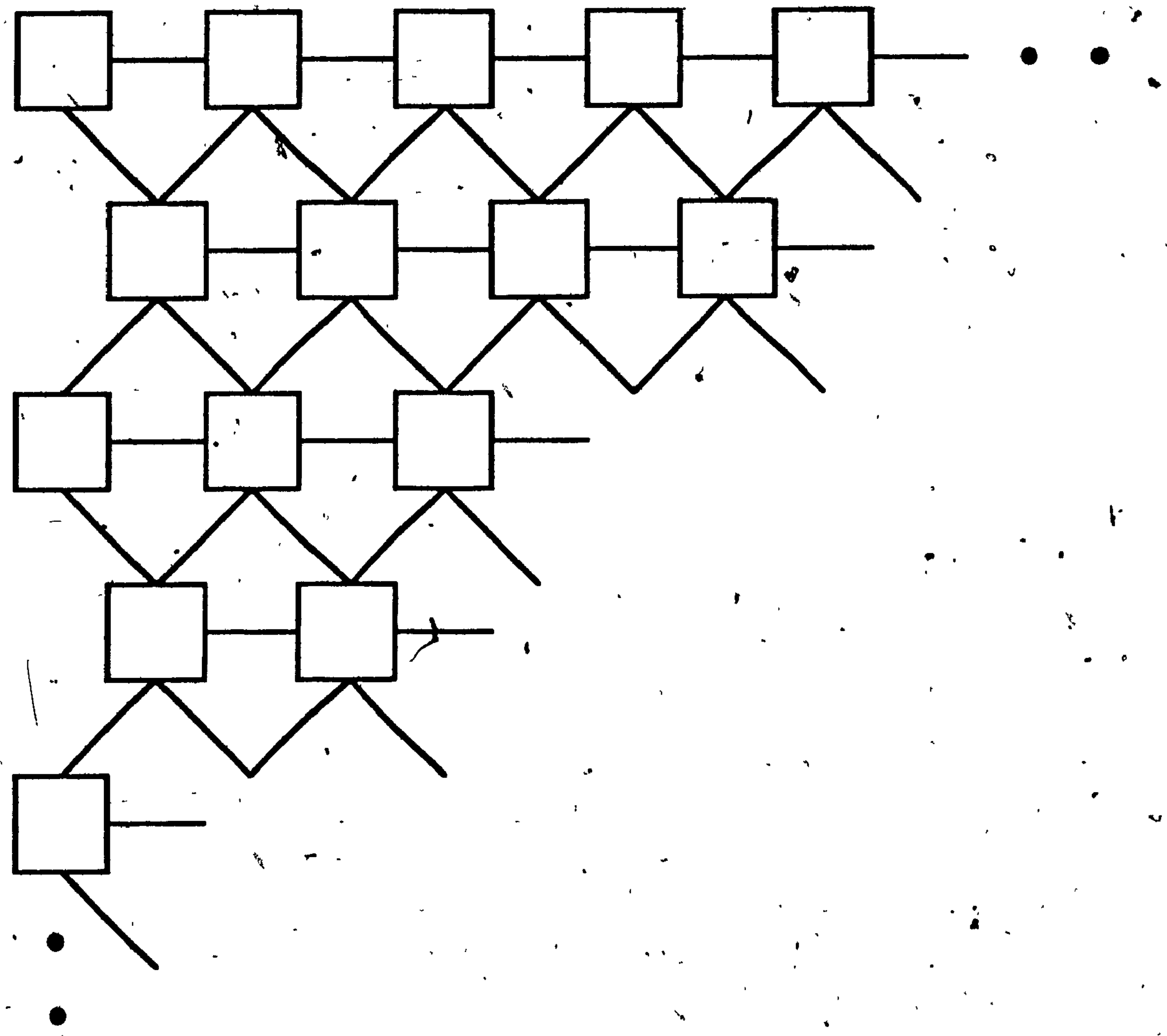

Figure III.1. A simplified view of the architecture of the CLIP4 multiprocessor using six connections per processor.

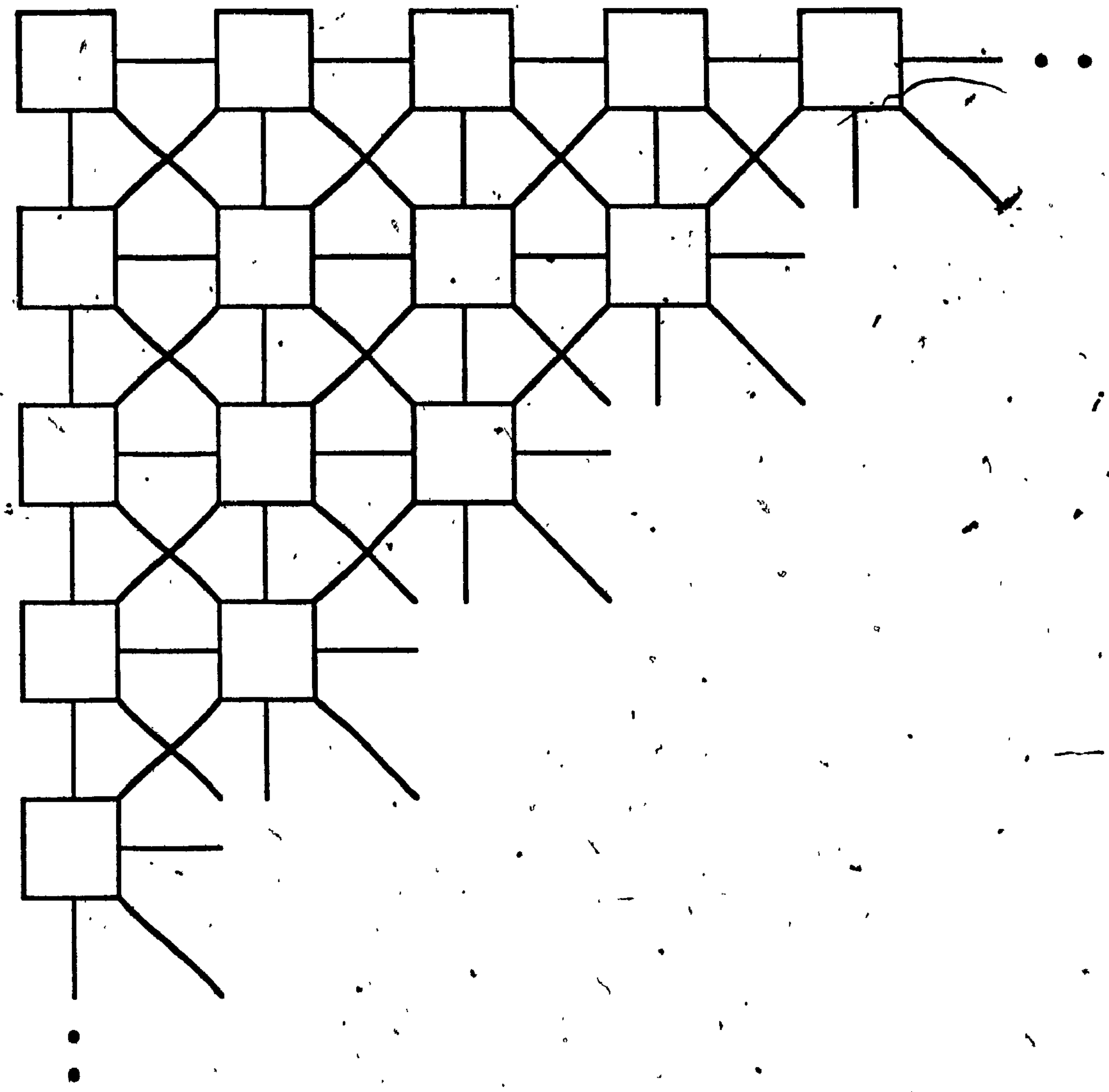

Figure III.2. A simplified view of the architecture of the CLIP4 multiprocessor using eight connections per processor.

III.4 Thinning on the PASM Multiprocessor

The PASM multiprocessor consists of 1024 microprocessors [30] and [31]. Each processor is associated with its own memory module. Each memory module contains two memory units, thus allowing a processor to access one memory unit while data is being loaded into the other memory unit. There is no shared memory between processors, however message passing facilities are provided. The processors are connected in a cube network. Figure III.3 illustrates a cube network connection for four and eight processors.

The main feature of the PASM multiprocessor is that it is a partionable SIMD/MIMD system (that is it can be structured as one or more independent SIMD amd/or MIMD machines). This feature allows for greater flexibilty in algorithm design. An algorithm written for PASM does not need to conform to the structure imposed by either an SIMD or an MIMD machine.

As Figure III.4 illustrates, the two main components of PASM are the parallel computation unit (PCU) and the host computer. The PCU contains the 1024 processors as well as the interconnection network. The host computer is a PDP 11 which is responsible for job scheduling and loading of the memory modules. As Figure III.4 illustrates, the host computer is connected to all of the microprocessors through a common bus. A prototype of the PASM multiprocessor has been developed and has been used for testing and implementing various thinning algorithms.

Kuehn et al. [20] implemented three different thinning algorithms on PASM. The first of these was a peeling algorithm proposed by Arcelli [1]. As shown in Figure III.5, this algorithm requires eight scans per pass. Next they implemented a shelling algorithm proposed by Rosenfeld et al. [29]. In this algorithm, the 4-distance transformation of all dark points in the pattern is first calculated. The skeleton then consists of all points which satisfy at least one of the two following conditions :

1) the point is a local maximum (that is the value of the point is greater than or equal to the values of all of its 4-neighbours);
2) the deletion of the point will break the connectivity of the original pattern (3 X 3 windows similar to the ones used by peeling based algorithms are used to test for connectivity).

Finally a hybrid algorithm was proposed which combined the best features of both peeling and shelling based algorithms. This hybrid algorithm consists of first using a simplified version of a shelling algorithm to delete most of the points that do not belong to the skeleton in a fixed number of passes. Then a peeling algorithm is used to thin the skeleton to a line drawing. Details of this implementation are not presented by Kuehn et al. [20]. However, results showed that this hybrid algorithm performed slightly better than either the peeling or shelling based algorithms did alone.

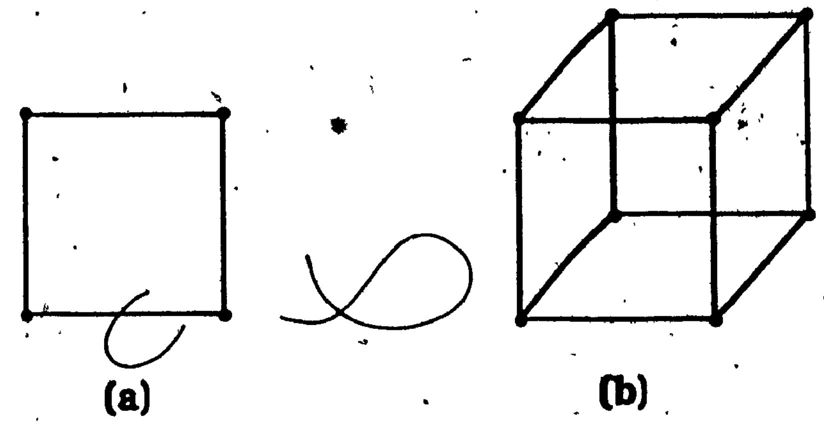

Figure III.3. Diagram of a cube network connection scheme for (a) four processors and (b) eight processors. In a cube network containing 2^n processors, each processor will have only n connections.

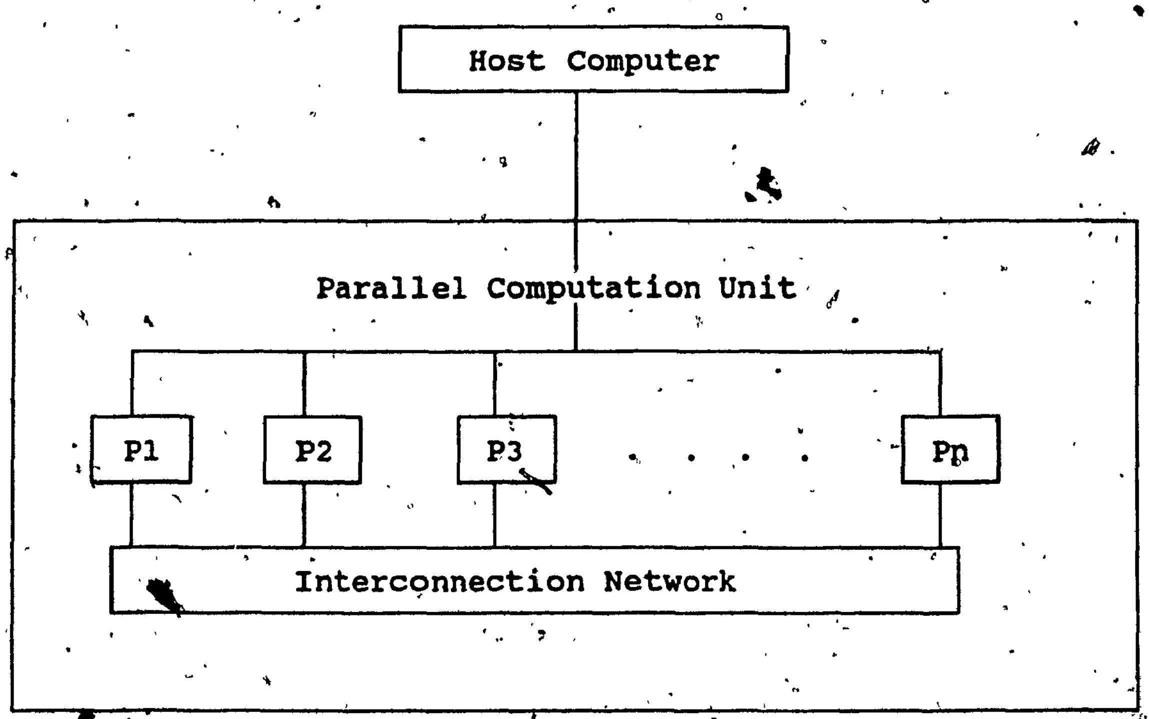

Figure III.4. A simplified view of the architecture of the PASM multiprocessor showing the host computer and the parallel computation unit.

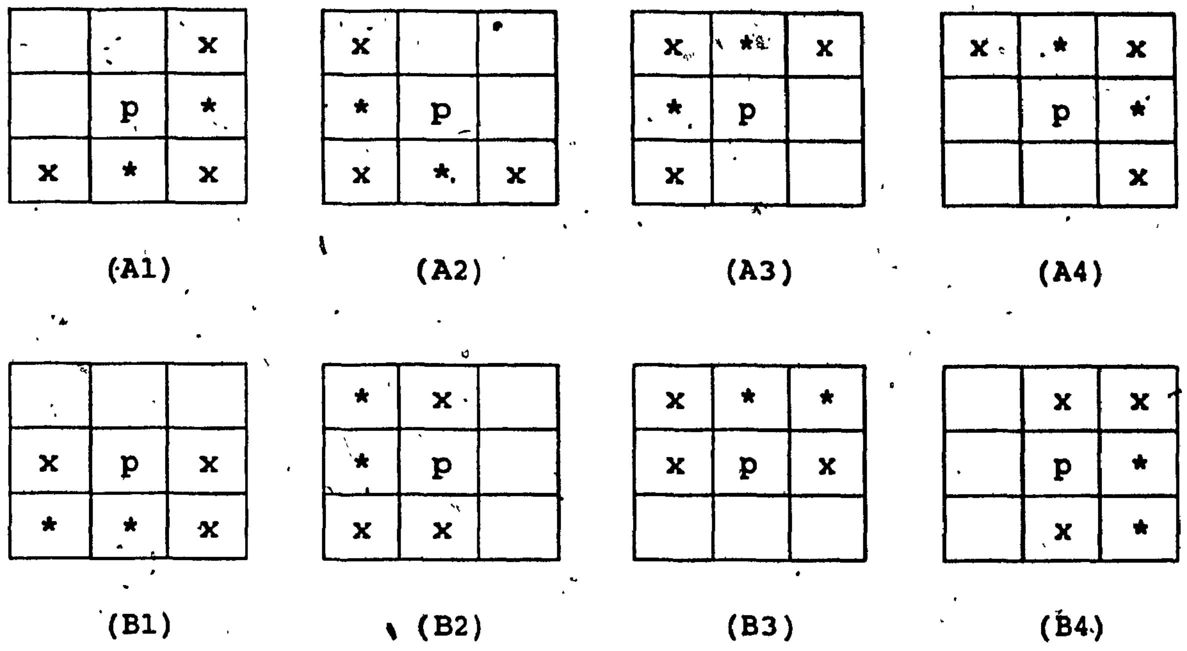

Figure III.5. For the algorithm proposed by Arcelli [1], all points and their neighbourhoods are simultaneously compared with the window A1. If the neighbourhood of a point matches the window, the point is deleted. This process is repeated for windows B1, A2, B2, A3, B3, A4, and B4 in that order to form one pass. The algorithm terminates when a pass is completed where no points are deleted.

Points labelled '*' and 'p' represent dark points, points labelled 'x' can be either dark points or white points, and points not labelled represent white points.

III.5. Thinning on the FLIP Multiprocessor

The FLIP multiprocessor has an MIMD architecture with 16 processors [11]. As illustrated in Figure III.6, the FLIP multiprocessor consists of two main components, the FIP (flexible individual processor) and the PEP (peripheral data exchange processor). The PEP is used for fast I/O between the host computer memory and the FIP. The FIP consists of 16 processors. Each processor is physically connected to all the other processors through a common bus. Each processor has two input ports and one output port. There are 16 data buses between the PEP and the FIP. Each processor is connected to two of these buses. Therefore, every two processors share two buses between them. Each processor has its own memory, which is divided into the following two components: 1) 50 bytes used for data only; and 2) 1024 bytes used for code only.

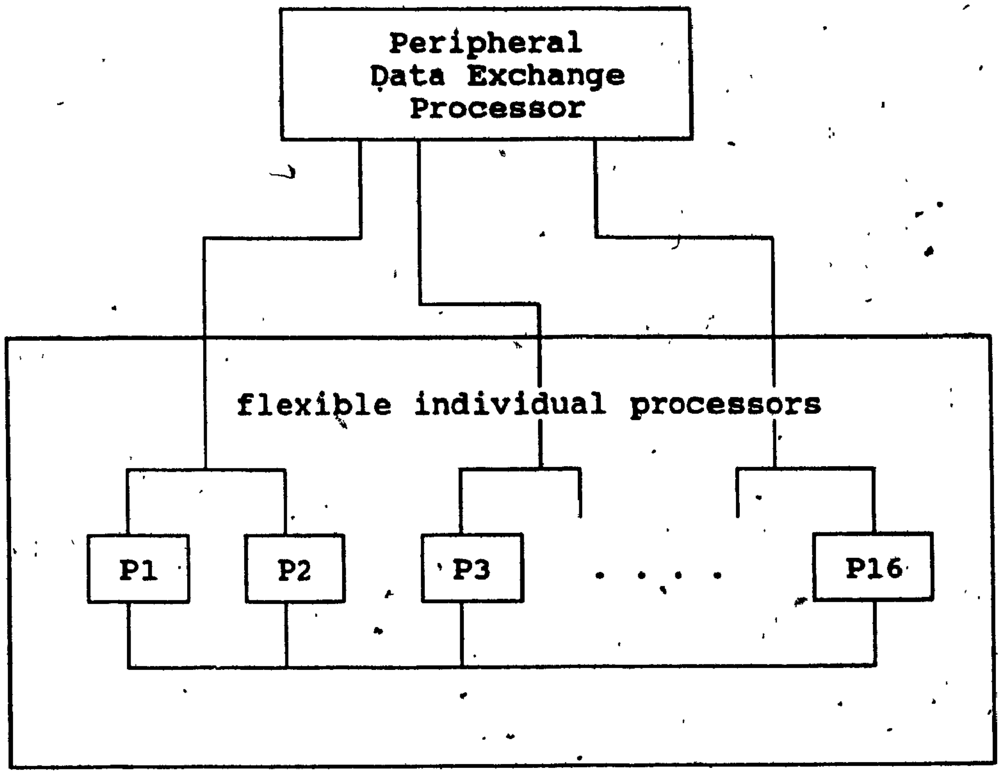
Peripheral Data Exchange Processor
flexible individual processors
P1
P2
P3
. . . .
P16

Figure III.6. A simplified view of the architecture of the FLIP multiprocessor.

III.6. Review of some Other Multiprocessors Designed for Pattern Recognition Applications

Although the three multiprocessors described so far are all well known examples of working multiprocessors that have been designed for pattern recognition applications in mind, many others are currently being developed or have already been proposed. Some of these include pipeline based multiprocessors such as those described by Sternberg [33] and Naccache [23]. Another multiprocessor designed specifically for pattern recognition applications is the Template Controlled Image Processor (TIP) proposed by Hanaka et al. [12]. The Homogeneous Multiprocessor proposed by Dimopoulos [6] and the Connection Machine proposed by Hillis [14] are good examples of general purpose multiprocessors which are also well suited for these types of applications.

Chapter IV

Our Multiprocessor Implementations

IV.1. Description of the Homogeneous Multiprocessor

The Homogeneous Multiprocessor consists of two parts: the Homogeneous Multiprocessor Proper (HMP) and the H-network [6] and [21]. Our implementations do not require the H-network, we will therefore only describe the HMP. A simplified view of the architecture of the HMP is shown in Figure IV.1. As Figure IV.1 shows, the HMP is composed of $n \geq 1$ processors $P_1, P_2, \ldots, P_n$. Each processor P_i has its own memory M_i that it can access directly. Processors P_{i-1} and P_{i+1}, if both exist, are called the neighbours of P_i. All processors have two neighbours, except processors P_1 and P_n. The only neighbour of processor P_1 is processor P_2, and the only neighbour of processor P_n is processor P_{n-1}. A processor P_i can also access through switches the memories of its neighbours. At any given time only one processor can access a memory. Overall, the HMP has a MIMD architecture.

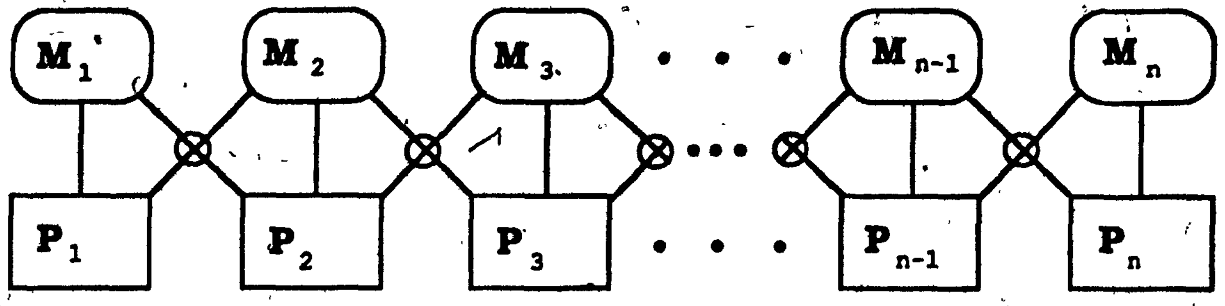

Figure IV.1. A simplified view of the architecture of the Homogeneous Multiprocessor Proper. The P's are processors; the M's are memories. A processor can access its own memory and the memories of its neighbours.

IV.2. Why use the Homogeneous Multiprocessor

The Homogeneous Multiprocessor [6] was selected mainly because of the following three reasons:

1) It was available to us through the use of a simulator [21]. This provided us with a reliable working environment.

2) The availability of shared memory between adjacent processors. This is ideal for image processing algorithms since most of these algorithms perform only local operations on points within the pattern. Therefore the architecture of the Homogeneous Multiprocessor is well suited for image processing applications.

3) Results are obtained in machine cycles. This is desirable since a true indication of the performance of an application can be obtained.

IV.3. Function Decomposition Implementation

In outline, each processor executes one scan on the pattern as the pattern is pipelined through the HMP. After a processor P_i has finished a scan, it checks for termination of the modified SPTA. If processor P_i reports termination, then the skeleton is available in memory M_i. Now we give some specifics of the implementation.

In the beginning, the pattern is stored in memory M_1. Processor P_1 begins the first scan. As soon as it has finished scanning a row, the row is moved to memory M_2. Once memory M_2 has received the first two rows of the pattern, then processor P_2 begins the second scan. Thus processor P_2 will always be at least two rows behind processor P_1. Then after processor P_2 has scanned a row, the row is moved to memory M_3. In general, processor P_i executes scan i, and after scanning a row, it moves the row to memory M_{i+1}. As processor P_i scans row k, processor P_{i+1} scans row k-2. As each processor finishes a scan, it checks for termination.

If the last processor P_n has finished the n^{th} scan and the algorithm has not yet terminated, then processor P_n begins the $n+1^{th}$ scan and starts moving the rows to memory M_{n-1}, where processor P_{n-1} begins the $n+2^{th}$ scan. Figure

IV.2 illustrates an example showing which scans are executed by each processor.

In brief, the pattern is continuously pipelined to and fro between memory M_1 and M_n until one of the processors reports termination.

	Processors			
	P1	P2	P3	P4
	scan 1			
Scan numbers		scan 2		
executed			scan 3	
by each				scan 4
processor				scan 5
			scan 6	
		scan 7		

Figure IV.2. Diagram showing which scans are to be executed by each of the processors for the function decomposition implementation. This example illustrates a pattern which requires seven scans to process and there are four processors available. Note that scan number four must be completed by processor P4 prior to the start of execution of scan number five.

Algorithmic Pseudo Code for the Function Decomposition SPTA

Declaration of variables :

```
const
  LASTPROC = the number of processors in the pipeline;

var
  row,j,k : integer;
  proc    : integer;
     {  Contains the location number of the current processor
        in the pipeline.  This value is initially ZERO for
        all of the processors, except for the first one,
        where proc = 1. }

  next  : direction;
     {  Indicates the direction of flow of the data in the
        pipeline.  When the variable next is used as a
        subscript, it refers to either the left or right
        neighbour processor. }

  received : integer;
     {  The number of the last row that was received by the
        current processor. }

  terminate : boolean;
     {  The flag indicating that a processor has reported
        termination of the SPTA.  At this time, the other
        processors are signalled that termination has been
        reported, and the pattern is pipelined to the
        processor numbered end_proc. }

  end_proc  : integer;
     {  The number of the processor at the end of the
        pipeline.  When the direction of the pipeline is to
        the right, then end_proc has a value of LASTPROC, and
        when the direction of the pipeline is to the left,
        then end_proc has a value of 1. }
```

```
procedure FUNCTION_DECOMPOSITION_SPTA(var PATTERN : pat_type)
begin
  INITIALIZE_FUNCTION;
     {  Initialize so that the algorithm can begin. }

  repeat
    INIT_SCAN;
     {  Prepare the next processor for another scan. }

    repeat
      row := row + 1;
      repeat
      until (received > row);
     {  Wait until enough rows have been received.  The
        number of rows received must always be at least one
        greater than the row number currently being executed,
        so that the complete neighbourhood of a point is
        available. }

      if terminate then
        terminate_next := TRUE
     {  If termination has been reported, then signal the
        next processor.  Each processor will in turn signal
        its next neighbour.  There is now no more need to
        skeletonize any rows.  However rows are still moved
        to the next processor so that the skeleton can be
        assembled in the end processor. }

      else
        SKELETONIZE(PATTERN,j,row,row,d[k]);
     {  Execute the kth scan on the current row. }

      if (proc <> end_proc) then
        begin
          MOVE(row,next);
          if (row = MAXROW) then
            received_next := MAXROW + 1
          else
            received_next := row;
        end;
     {  If the current processor is not the end processor
        then move the row to the next processor and increment
        the rows received counter of the next processor. }

    until (row = MAXROW);
     {  Repeat for each row in the pattern. }

    CHECK_TERMINATE;
     {  The scan is completed, so test for the terminating
        criteria. }
  until FOREVER;
end;
```

```
procedure INITIALIZE_FUNCTION;

begin
  repeat
  until (proc > 0);
     { Wait until the current processor has been given a
       processor number. }

  if (proc = 1) then
    begin

     {  Initialize the first processor so that }
      next := right;     { the direction of flow is to the
                           right, }
      i := 1;            { the pass number = 1, }
      j := 0;            { the scan type is for left and right
                           edgepoints, }
      k := 1;            { the scan number = 1, and }

      received := MAXROW+1;
        { the number of rows received is one greater than the
          total number of rows in the pattern.  The extra row
          is a blank row.  This is necessary so that the
          bottommost row of the pattern can be processed. }
    end;

  if (proc <> LASTPROC) then
    begin

     {  If the current processor is not the last processor,
        then initialize the next processor so that }

      proc_next := proc + 1;    { it has a processor number, }
      received_next := 0;       { it has not yet received any
                                  rows, and }
      next_next := next;        { the direction of flow remains
                                  the same. }
    end;

  terminate := FALSE;   { Set the terminate flag to FALSE. }
end;
```

```
procedure INIT_SCAN;
begin
  if (next = right) then
    end_proc := LASTPROC
  else
    end_proc := 1;

     { Determine which processor is the end processor of the
       pipeline. }

  repeat
  until (received $\geq$ 1);
     { Wait until at least one row has been received before
       continuing. }

  row := 0;    { Set the number of the current row to zero. }

     { If the current processor is not the end processor
       then prepare to initialize the next processor for the
       next scan. }

  if (proc <> end_proc) then
    begin
      if j = 0 then

     { If the current scan is for left and right edgepoints,
       then the next scan will be for top and bottom
       edgepoints with the pass number remaining the same. }

       begin
         $j_{next}$ := 2;
         $i_{next}$ := i;
       end
      else

     { If the current scan is for top and bottom edgepoints,
       then the next scan will be for left and right
       edgepoints. Since this is the first scan of the next
       pass, the pass number must be incremented by 1. }

       begin
         $j_{next}$ := 0;
         $i_{next}$ := i + 1;
       end;

      $next_{next}$ := next; { Keep the direction of the flow in
                             the same direction. }
      $k_{next}$ := k + 1;    { Increment the scan number by one. }
    end;
  d[k] := 0;
end;
```

```
procedure CHECK_TERMINATE;
begin

     { If criterion1 or criterion2 is TRUE, then set the
       terminate flag to TRUE. }
  if (d[k] = 0) or (d[k] = d[k-2]) then
    terminate := TRUE;

     { If the current processor is the end processor and the
       terminate flag is set to TRUE, then the algorithm has
       been completed. }

  if (proc = end_proc) and terminate then
    STOP;

     { Otherwise, if the processor is the end processor, but
       the terminate flag is FALSE, then change the
       direction of flow and begin the next scan. Note that
       the end processor can execute two consecutive scans:
       one scan prior to testing for the terminating
       criteria; and in case the terminating criteria fail,
       the first scan for the new direction of flow. }

  if (proc = end_proc) then
    begin
      if (next = right) then
        next := left
      else
        next := right;
      k := k + 1;
      if (j = 0) then
        j := 2
      else
        begin
          j := 0;
          i := i + 1;
        end
    end
  else

     { The current processor is not the end processor.
       However, it has finished executing its scan on the
       pattern. So set the number of rows received to ZERO
       and wait for the next scan to begin. }

    begin
      if terminate then
        terminate_next := TRUE;
      received := 0;
    end;
end;
```

IV.4. Data Decomposition Implementation

In outline, each processor P_i executes all scans on a segment containing R_i rows of the pattern. To do this, the top most R_1 rows of the pattern are processed by P_1, the next R_2 rows are processed by P_2 and so on. An attempt is made that the number of rows is evenly distributed among the processors. Suppose the number of processors, n, is equal to 4, and the number of rows in the pattern is 26. Then processors P_1 and P_2 process 7 rows each, and processors P_3 and P_4 process 6 rows each. Thus extra rows, if any, are distributed one extra row per processor starting at processor P_1.

To process R_i rows by processor P_i, we stored the R_i rows in memory M_i. Moreover, memory M_i contained one row above and one row below the segment of R_i rows, see Figure IV.3. This is because to process a point, we need to examine its 8-neighbours. Thus there is a certain amount of overlap in the rows that were stored in the different memories. One can argue that this overlap is not necessary, since any processor can access the memory of its neighbours. But the more times a processor accesses the memories of its neighbours, the slower becomes the SPTA.

By allowing this overlap of rows, the number of interprocessor communications were reduced, thus reducing the overhead. If however a processor P_i flagged a point or declared a point to be a safepoint in the topmost or bottommost rows of R_i, then this was communicated to the neighbouring processor. We found by experimentation that by allowing overlapping of rows between memories, the SPTA was speeded up by about 10 percent.

The processors must be synchronized: processor P_i cannot begin a pass until its neighbours P_{i-1} and P_{i+1} have finished the previous pass. Every processor checks for termination at the end of each pass. When all processors have reported termination, the segments of the skeleton are distributed in memories M_1 to M_n. These segments can then be assembled back into memory M_1.

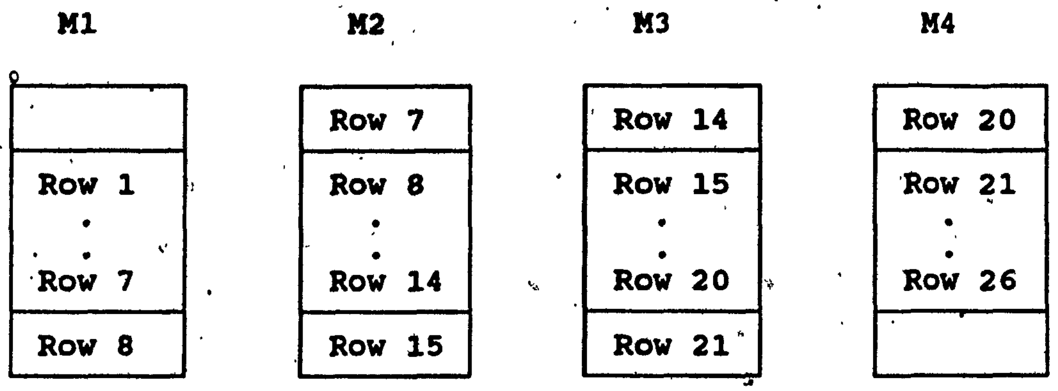

Figure IV.3. The allocation of the rows of the pattern to the available memory modules for the data decomposition implementation. This example shows four memory modules and a pattern with a total of 26 rows.

Algorithmic Pseudo Code for the Data Decomposition SPTA

```
Declaration of variables :

const
  LASTPROC = the number of processors in the network;

var
  j,k       : integer;
  proc      : integer;
     {  Contains the location number of the current processor
        in the pipeline.  This value is initially ZERO for
        all of the processors, except for the first one,
        where proc = 1. }

  first_row : integer;
     {  The first row of data that has been allotted to the
        current processor. }

  last_row  : integer;
     {  The last row of data that has been allotted to the
        current processor }

  rows_keep : integer;
     {  The number of rows that the current processor has
        allotted for itself. }

  done_flag : boolean;
     {  The flag indicating that a processor in the pipeline
        has received all the rows of the pattern from its
        right neighbour.  This flag is used when the pattern
        is being moved back to the first processor. }

  end_flag  : boolean;
     {  The flag indicating that a processor may start moving
        rows to its left neighbour; that is, its right
        neighbour has completed the algorithm on its segment
        of the pattern, and has begun moving the pattern into
        the current processor. }
```

```
procedure DATA_DECOMPOSITION_SPTA(var PATTERN : pat_type);
begin
  INITIALIZE_DATA;
     {  Initialize so that the algorithm can begin. }
  repeat
    i := i + 1;    { Increment the pass number. }

     {  Synchronize with the left neighbour.  Wait until the
        left neighbour has completed pass i-1.

    if (proc <> 1) then
      repeat
      until (i ≤ i_left);

     {  Synchronize with the right neighbour.  Wait until the
        right neighbour has completed pass i-1.

    if (proc <> LASTPROC) then
      repeat
      until (i ≤ i_right);

    j := 0;         { Set scan type to left/right edgepoint. }
    k := k + 1;     { Increment the scan number. }

    SKELETONIZE(PATTERN,j,first_row,last_row,d[k]);
                    { Execute the kth scan on the rows that are
                      allotted to the current processor. }

    if (d[k] <> 0) and (d[k] <> d[k-2]) then
      begin
        j := 2;     { Set scan type to top/bottom edgepoint. }
        k := k + 1;    { Increment the scan number. }

        SKELETONIZE(PATTERN,j,first_row,last_row,d[k]);
                    { Execute the kth scan on the rows that are
                      allotted to the current processor. }
      end;
  until (d[k] = 0) or (d[k] = d[k-2]);
                    { Repeat executing passes on the entire
                      pattern until at least one of the
                      terminating criteria is TRUE. }

  i := MAXINT;      { The current processor has finished the
                      skeletonization process on its segment of
                      the pattern. The pass number is set to
                      MAXINT, so that its neighbours can
                      execute more passes, if need be. }

  TERMINATE;        { The skeletonization process is completed.
                      So prepare to reassemble the entire
                      pattern in the first processor. }
end;
```

```
procedure INITIALIZE_DATA;
begin
  repeat
  until (proc > 0);
     {  Wait until the current processor has been given a
        processor number. }

  if (proc = 1) then
    first_row := 1;
     {  The first processor will contain a section of the
        pattern, starting with the first row, ie. row = 1.

     {  rows_keep equals the ceiling of the number of rows
        not yet allocated divided by the number of memories
        that do not yet contain any rows. }
  rows_keep := ceiling((MAXROW - first_row - 1) /
                       (LASTPROC - proc + 1));

  last_row := first_row + rows_keep - 1;

     {  If the current processor is not the last processor,
        then initialize the next processor.  Set the
        processor number and the first row of the pattern to
        be allocated in the next processor. Then move all not
        yet allocated rows of the pattern, point by point to
        the next processor, starting with the last column of
        the last row. }

  if (proc <> LASTPROC) then
    begin
      first_row_right := first_row + rows_keep;
      proc_right := proc + 1;
      for row := MAXROW downto last_row do
        for column := MAXCOLUMN downto 1 do
          PATTERN_right[row,column] := PATTERN[row,column];
    end;

     {  Initialize all termination flags to FALSE. }
  done_flag := FALSE;
  end_flag := FALSE;

  i := 0;    { Set the pass number to ZERO. }
  k := 0;    { Set the scan number to ZERO. }

     {  Initialize the number of dark points neither flagged
        nor declared to be safepoints to a value of ZERO for
        all scans. }

  for k := -1 to MAXSCAN do
    d[k] := 0;

end;
```

```
procedure ADJUST;
begin

     ( If a point on a border row was either flagged or
       declared to be a safepoint, then the copy of this
       point in the neighbouring processor must also be
       either flagged or declared to be a safepoint. Note
       this procedure is called from the procedure
       SKELETONIZE. )

     ( Adjust for points in the left neighbour processor. )

  if (row = first_row) and (proc <> 1) then
    PATTERN_left[row,column] := p

     ( Adjust for points in the right neighbour processor. )

  else if (row = last_row) and (proc <> LASTPROC) then
    PATTERN_right[row,column] := p;

end;
```

```
procedure TERMINATE;
begin
  if (proc = 1) then
    if (proc = LASTPROC) then
      STOP        { If there is only one processor then STOP. }
    else
      begin
        repeat
        until done_flag;
        STOP;   { Otherwise only stop when the entire pattern
                  has been reassembled in the first
                  processor. }
      end
  else
    begin
      if (proc <> LASTPROC) then
        repeat
        until end_flag;

      { Wait until the right neighbour has completed the
        algorithm on its segment of the pattern, and has
        started to move part of the pattern into the current
        processor. }

      end_flag_left := TRUE;

      { Set the end_flag in the left neighbour to TRUE, so
        that it can also start moving the pattern. }

      { Move all rows of the pattern, starting from the last
        point in the last row, to the first point in the
        current processor to the left neighbour.  The rows
        are moved from bottom to top to allow the pattern to
        be moved through the pipeline in parallel. }

      for row := MAXROW downto (first_row + 1) do
        for column := MAXCOLUMN downto 1 do
          PATTERN_left[row,column] := PATTERN[row,column];

      { Once all these rows have been moved to the left
        neighbour, then the current processor will simply
        sit idle until such time that the entire pattern has
        been reassembled in the first processor, and the
        first processor has terminated the algorithm. }

      done_flag_left := TRUE;
      repeat
      until FOREVER;
    end;
end;
```

IV.5. Satisfying Hilditch's Refinements

In this section, we will informally show that the modified SPTA satisfies the refinement proposed by Hilditch [13], ie., that our modified SPTA deletes exactly one layer of border points per iteration. To do this it will be sufficient to show that our edgepoint detection operation satisfies this refinement. This condition is sufficient, since only those points which are designated as edgepoints are eligible for deletion.

As was shown in Figure II.7, a dark point p is determined to be an edgepoint if it satisfies the following criterion:

> A dark point p is defined to be a left edgepoint if its neighbour n[0] has a value less than (i- MAXINT), where i is the current pass number.

Similar criteria define top, right, and bottom edgepoints using neighbours n[2], n[4], and n[6] respectively.

As can be seen from Table II.1, a point is only considered to be an edgepoint if at least one of its 4-neighbours was either an original white point, or a point that was deleted during a previous pass. Therefore all points designated as edgepoints satisfy Hilditch's refinement since they were all edgepoints at the beginning of the current pass.

Although the SPTA is not a parallel algorithm in the sense that the new value of a point at pass number i can be determined entirely from the values obtained during pass number i-1, it does satisfy Hilditch's refinement on a multiprocessor implementation. Furthermore, the skeletons produced are of good quality, that is they are of unit width, do not suffer from excessive erosion, preserve the connectedness of the original pattern, and contain sufficient information for the reconstruction of the original pattern. For our particular implementations the algorithm is not required to be parallel since all points in the pattern are not processed in parallel.

For the function decomposition implementation, a sequential thinning algorithm is sufficient, since each scan is still performed sequentially on each row of the pattern.

For the data decomposition implementation, all points within a given row are still evaluated sequentially. However it is possible that points in adjacent rows are evaluated in parallel. This results in producing skeletons which are not necessarily unique as the number of processors varies.

However, in each case the skeleton produced is of good quality.

However to show that the SPTA can be implemented as a parallel algorithm, we have given in Chapter V1, a proposed implementation on the Connection Machine. The Connection Machine has an architecture that will allow each point in the pattern to be processed in parallel.

Chapter V

Experimental Results and Discussion

V.1. Description of the Working Environment

The one processor modified SPTA and the multiprocessor implementations of the modified SPTA were tested on a data set of 216 patterns. The patterns were digitized from hand written characters 'A' to 'Z' and '0' to '9'. The average size of a pattern was 20 rows and 16 columns, with the maximum size being 27 rows and 32 columns. The characters were hand printed by different students at Concordia University, Montreal and were digitized by an ECRM 5200 auto reader.

A simulator for the HMP [21] was available to us on a VAX 11/780. It simulated an HMP with the following resources: 1) Up to 64 8MHz MC68000 processors, and 2) the memory for each processor being limited to 10K bytes. Our implementations of the SPTA on the HMP were coded in MC68000 assembly language [19], the coding being intuitively as efficient as possible.

V.2. Results for the Modified SPTA

To begin with, we implemented the SPTA on a single processor as proposed by Naccache et al. [24] and the SPTA as modified by us. The average number of scans to skeletonize a pattern was 6.81 for the original SPTA and 4.88 for our modified SPTA. Thus on average, the modified SPTA executes 1.93 fewer scans per pattern. Table V.1 shows the results for the original SPTA and the modified SPTA. Measuring time in machine cycles, we found our modified SPTA to be 25 percent faster than the original SPTA. Thus in all further discussions about results, when we talk about the SPTA, we mean the modified SPTA.

Table V.1.

The average number of scans and passes required by the original SPTA and the modified SPTA. The modified SPTA requires approximately two fewer scans per pattern than does the original SPTA.

Algorithm	the average number of scans per pattern	the average number of passes per pattern
SPTA	6.81	3.69
Modified SPTA	4.88	2.71

V.3. Results for the Multiprocessor Implementations

Table V.2 shows the average time to process a single pattern under the data decomposition and function decomposition implementations with varying number of processors. For visual clarity, the results from Table V.2 have been plotted in Figure V.1.

Let us first consider the function decomposition implementation. We noticed that with two processors, the algorithm slowed down by 14.7 percent as compared to one processor. This is mainly because the pattern oscillates between the memories of the first and second processors; each oscillation increases the overhead as the end processor stops the pipelining to reverse the direction of the pattern movement. The function decomposition implementation performed best with six processors when it was found to be 34 percent faster than the one processor SPTA. This kind of implementation performs the fastest when a pattern is skeletonized in one pipeline movement from processor P_1 to P_n without having to reverse directions. When the number of processors is larger than the number of scans required to skeletonize a pattern, then the processors towards the end of the pipeline are not required for the skeletonization. Nevertheless, the pattern is pipelined through their memories, causing increased overhead, and thus slowing down the SPTA. This is confirmed by Figure V.1, where the function decomposition SPTA slows down when the number of processors increases beyond six.

We now consider the data decomposition implementation. As Figure V.1 shows, the implementation became faster with an increase in the number of processors: for example with 8 processors the speed up is 66.2 percent when compared to one processor. A larger speed up was not obtained for the following two reasons :

a) the size of the patterns is relatively small, therefore the size of the segments that each processor was assigned did not significantly decrease as more processors were added.

b) the overhead for distributing the pattern from the first processor to the remaining processors increased as the number of processors increased.

The data decomposition is faster than the function decomposition mainly because the former requires less movement of the pattern from one memory to another. For the same number of processors the data decomposition is about 45 percent faster than the function decomposition.

Table V.2.

Experimental results. The time for one processor was 197,414 machine cycles.

Number of Processors	Average Time in Machine Cycles per Pattern	
	Data Decomposition Implementation	Function Decomposition Implementation
2	129,636	226,502
3	102,904	185,547
4	88,334	160,960
5	79,738	133,976
6	74,505	130,314
7	69,637	132,084
8	66,754	136,955
9	65,400	141,936
10	63,685	146,862
11	62,612	151,811
12	61,865	156,705
13	60,934	161,687
14	60,269	166,646
15	59,947	171,613

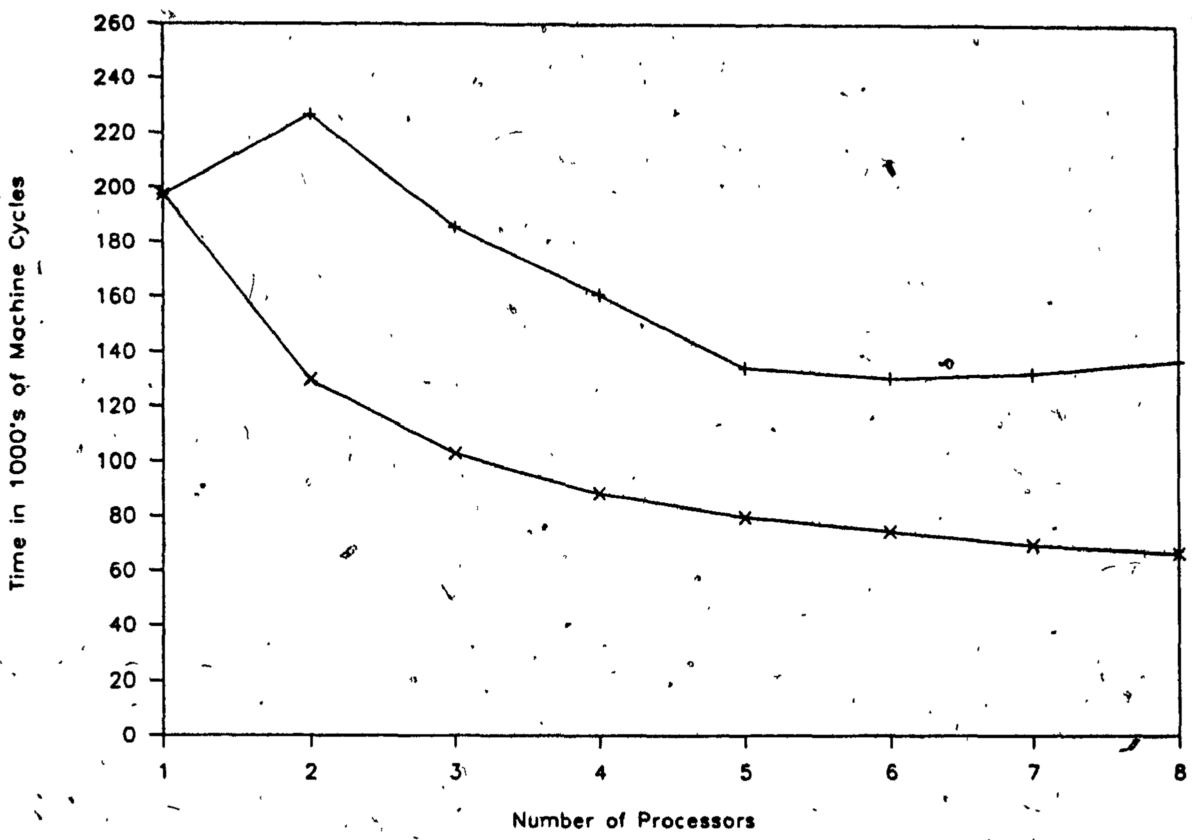

Figure V.1. Plots of the average time taken (in machine cycles) per pattern versus the number of processors for the Data Decomposition (x's) and the Function Decomposition (+'s) implementations of the SPTA. The time for the one processor SPTA is also shown above.

Chapter V1

Proposed Implementation on the Connection Machine

VI.1. Description of the Connection Machine

To show that the SPTA can also be implemented on a SIMD computer, where each point in the pattern can be evaluated in parallel, we present an implementation of the SPTA on the Connection Machine [14] and [15]. The Connection Machine was chosen because it is an SIMD computer that is well suited for image processing applications. It contains a host computer and 64K processors. The host computer broadcasts instructions and sends data to all 64K processors. Each processor has its own memory consisting of 512 bytes. Figure VI.1 shows a simplified view of the architecture of the Connection Machine. The processors are connected using an n-cube architecture. See Figure III.3 for an illustration of n-cubes of order 2 and 3. Thus each processor has 16 connections. Conceptually however, the connections are programmable (that is, from a programmers point of view, the connections can be arranged to meet the requirements of the problem being solved). Thus a collection of processors can be conceptually viewed as a data structure.

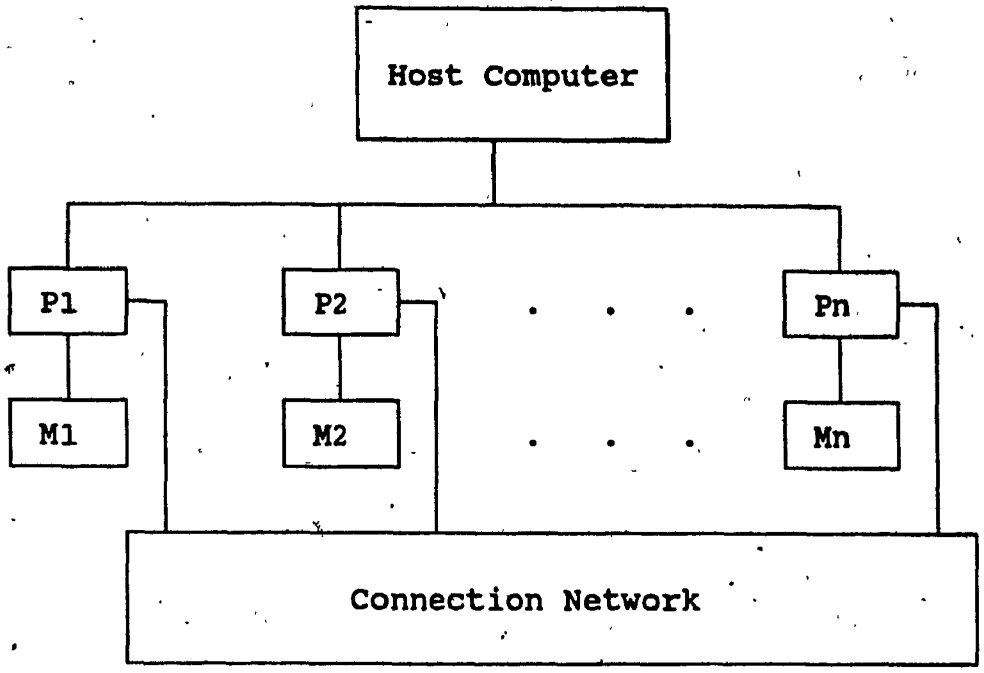

Figure VI.1. A simplified view of the architecture of the Connection Machine.

VI.2. Proposed Algorithm on the Connection Machine

We here propose an implementation of the SPTA for the Connection Machine. Each point in the pattern will be assigned to a separate processor. The processors will conceptually be connected into a square grid, with each processor having access to its 8 neighbouring processors. Since each point is being evaluated in parallel, a pass is sub-divided into four scans. Each scans tests for one type of edgepoint. The algorithm terminates when n passes have been completed, where n is equal to the larger of :

1) half the height of the pattern, or
2) half the width of the pattern.

Pseudo Code for the Connection Machine Implementation.

```
procedure CONNECTION_MACHINE_SPTA(var PATTERN : pat_type);

const
  MAXPASS = MAX (MAXROW/2 , MAXCOLUMN/2);
        { Is the maximum number of passes that would be
          required to skeletonize a pattern. }

begin

  INIT_CONNECTION_MACHINE;

        { Initialize each processor so that it contains only
          one point of the pattern. }

        { A maximum number of passes are executed on the
          pattern to guarantee that the terminating criteria
          have been satisfied. }

  for i := 1 to MAXPASS do
    for border := 0 to 6 step 2 do

        { For each type of edgepoint, load the 8-neighbours
          of the point p into local memory.  The execute one
          scan on the point p. }

        { Each type of edgepoint (right, top, left, and
          bottom), must be processed independently since all
          points within the pattern are processed in
          parallel. }

      begin
        GET_NEIGHBOURS;
        PROCESS(border);
      end;

 end;
```

```
procedure GET_NEIGHBOURS;
begin

  for k := 0 to 7 do

    MOVE(p,k);

         { Move the value of the point p to the processor
           containing the point n[k].  Each processor will
           move the value of its own point to the 8
           processors that contain the neighbours of p.  Thus
           each processor will have the value of its point p
           and the values of its 8-neighbours. }

end;



procedure PROCESS(border : border_type);
begin

         { If the point p is a dark point and an edgepoint,
           then x is TRUE, otherwise x is FALSE. }

  x := DARK(p) and WHITE(n[border]);

         { If the point p is a safepoint, then y is TRUE,
           otherwise y is FALSE. }

  y := SAFEPOINT(n,border);

  if x and y
    then  p := i                    {p is a safepoint }

    else

     if x
      then  p := i - MAXINT;        { p is a flagged point }
end;
```

VI.3. Expected Results for the Connection Machine Implementation

Although we did not have access to a Connection Machine [14], we conjecture that our implementation on the Connection Machine will execute in a time that is equal to the time required to process a single point times the number of passes required to complete the algorithm.

Since the Connection Machine is an SIMD computer, the time required to process one complete pass over the entire pattern is approximately equal to the time required to process one pass on a single point.

The maximum number of passes required is ideally equal to one half the maximum width of the pattern (that is one half of the largest diameter of any line segment in the pattern). Rather than spend time calculating this value, we have chosen to use an upper bound. This upper bound is equal to one half of the larger of either the height of the pattern or the width of the pattern (that is one half of the maximum dimension of the pattern). Since this implementation requires four scans per pass, the time required to execute the algorithm on a pattern will be proportional to four times this upper bound.

Chapter VII
Concluding Remarks

VII.1. Conclusion

In this thesis, we proposed a modification to the SPTA. With the modification, the SPTA executes fewer scans to skeletonize a pattern, thus speeding it up. We then reviewed some thinning algorithms that have been implemented on multiprocessors. In order to further speed up the SPTA, we adapted the modified SPTA to be implemented on the Homogeneous Multiprocessor Proper using data decomposition and function decomposition. The data decomposition implementation achieved a speed up of 66 percent as compared with the single processor implementation. The data decomposition implementation is also faster than the function decomposition implementation. Since the Homogeneous Multiprocessor is an MIMD machine with a maximum of 64 processors, a fully parallel algorithm where each point is evaluated in parallel cannot be used. Therefore in order to show that the SPTA can also be modified for such an environment, we proposed an implementation on the Connection Machine. We conjecture that the implementation on the Connection Machine will further speed up the SPTA.

VII.2. Possible Future Work

When the actual hardware becomes available, our implementations of the modified SPTA can be tested on a prototype of the Homogeneous Multiprocessor. Thus the results that we have obtained through simulations could be verified.

The proposed implementation of the SPTA on the Connection Machine could also be tested on the actual hardware. Work could be done to reduce the number of scans per pass from four to two for this implementation to further speed it up.

With the increasing availability of multiprocessors and algorithms designed to better utilize them, the doors will be opened up for many image processing applications which are currently not feasible. Many of these applications will require a thinning algorithm. We believe that our modified SPTA will be well suited for these types of implementations. Work could be done to incorporate the modified SPTA into such an environment. Furthermore, to increase its range of possible applications, the modified SPTA could be extended to handle multi-grey level patterns.

REFERENCES

[1] C. Arcelli, L. Cordella, S. Levialdi, "Parallel Thinning of Binary Pictures," Electronics Letters, vol. 11, no. 7, pp. 148-149, April 1975.

[2] C. Arcelli, "Pattern Thinning by Contour Tracing," Computer Graphics and Image Processing, vol. 17, pp. 130-144, 1981.

[3] C. Arcelli and G.S. DiBaja, "A Width-Independent Fast Thinning Algorithm," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 7, no. 4, pp. 463-474, July 1985.

[4] E.R. Davies and A.P.N. Plummer, "Thinning Algorithms: A Critique and a New Methodology," Pattern Recognition, vol. 14, pp. 53-63, 1981.

[5] E.S. Deutsch, "Thinning Algorithms on Rectangular, Hexagonal, and Triangular Arrays," Communications of the ACM, vol. 15, pp. 827-837, 1972.

[6] N.J. Dimopoulos, "On the Structure of the Homogeneous Multiprocessor," IEEE Transactions on Computers, vol. C-34, no. 2, pp. 141-150, February 1985.

[7] M.J.B. Duff, "CLIP4: A Large Scale Integrated Circuit Array Parallel Processor," Third International Conference on Pattern Recognition, pp. 728-732, 1976.

[8] M.J.B. Duff, "Review of the CLIP Image Processing System," Proceedings National Computer Conference, pp. 1055-1060, 1978.

[9] T.J. Fountain, "CLIP4: Progress Report," in Languages and Architectures for Image Processing, M.J.B. Duff and S. Levialdi, editors, London, England: Academic Press, pp. 283-291, 1981.

[10] G.C. Fox and J.O. Otto, "Algorithms for Concurrent Processors," Physics Today, vol. 37, pp. 50-59, May 1984.

[11] P. Gemmer, H. Ischen and K. Luetjen, "FLIP: A Multiprocessor System for Image Processing," in Langauges and Architectures for Image Processing, M.J.B. Duff and S. Levialdi, editors, London, England: Academic Press, pp. 245-256, 1981.

[12] S. Hanaka and T. Temma, "Template Controlled Image Processor (TIP) Project," in Multicomputers and Image

Processing, K. Preston, Jr. and L. Uhr, editors, London England: Academic Press, pp. 343-352, 1982.

[13] C.J. Hilditch, "Comparison of Thinning Algorithms on a Parallel Processor," Image and Vision Computing, vol. 1, no. 3; pp. 115-132, August 1983.

[14] W.D. Hillis, The Connection Machine, Cambridge, Massachusetts: The MIT Press, 1986.

[15] W.D. Hillis, "The Connection Machine," Scientific American, vol. 256, no. 6, pp. 108-115, June 1987.

[16] C.D. Howe and B. Moxon, "How to Program Parallel Processors," IEEE Spectrum, vol. 24, no. 9, pp. 34-41, September 1987.

[17] K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing, New York: McGraw Hill, pp. 613-637, 1984.

[18] L.H. Jamieson, "Characterizing Parallel Algorithms," in The Characteristics of Parallel Algorithms, L.H. Jamieson, D.B. Gannon, and R.J. Douglass, editors, Cambridge, Massachusetts: The MIT Press, pp. 65-100, 1987.

[19] T. King and B. Knight, Programming the M68000, Reading, Massachusetts: Micro Computer Books, Addison-Wesley, 1983.

[20] J.T. Kuehn, J.A. Fessler, and H.J. Siegel, "Parallel Image Thinning and Vectorization on PASM," IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Francisco, pp. 368-374, June 1985.

[21] K.F. Li and N.J. Dimopoulos, "The Performance Analysis of the Homogeneous Multiprocessor Proper," Canadian Electrical Engineering Journal, vol. 12, no. 1, pp. 3-10, January 1987.

[22] H. Ma, A Comparitive Study of Thinning Algorithms, Major Report, Department of Computer Science, Concordia University, Montreal, 1983.

[23] N.J. Naccache, Skeletonization of Binary Patterns: A Proposed Algorithm and Multiprocessor Network, M. Comp. Sci. Thesis, Department of Computer Science, Concordia University, Montreal, 1984.

[24] N.J. Naccache and R. Shinghal, "SPTA: A Proposed Algorithm for Thinning Binary Pattern," IEEE

Transactions on Systems, Man, and Cybernetics, vol. 14, no. 3, pp. 409-418, May/June 1984.

[25] T. Pavlidis, "A Flexible Parallel Thinning Algorithm," IEEE Proceedings Conference on Pattern Recognition and Image Processing, Dallas, pp. 162-167, August 1981.

[26] T. Pavlidis, Algorithms For Graphics and Image Processing, Rockville, Maryland: Computer Science Press, pp. 195-214, 1982.

[27] T. Pavlidis, "An Asynchronous Thinning Algorithm," Computer Graphics and Image Processing, vol. 20, pp. 133-157, 1982.

[28] A. Rosenfeld, "A Characterization of Parallel Thinning Algorithms," Information and Control, vol. 29, pp. 286-291, 1975.

[29] A. Rosenfeld and J.L. Pfaltz, "Sequential Operations in Digital Picture Processing," Journal of the Association for Computing Machinery, vol. 13, pp. 471-494, Oct. 1966.

[30] H.J. Siegel, "PASM: A reconfigurable Multimicrocomputer System for Image Processing," in Languages and

Architectures for Image Processing, M.J.B. Duff and S. Levialdi, editors, London, England: Academic Press, pp. 257-265, 1981.

[31] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr., H.E. Smalley, Jr., and S.D. Smith, "PASM : A Partionable SIMD/MIMD System for Image Processing and Pattern Recognition," IEEE Transactions on Computers, vol. C-30, pp.934-947, December 1981.

[32] R. Stefanelli and A. Rosenfeld, "Some Parallel Thinning Algorithms for Digital Pictures," Journal of the ACM, vol. 18, no. 22, pp. 255-264, 1981.

[33] S.R. Sternberg, "Pipeline Architectures for Image Processing," in Multicomputers and Image Processing, K. Preston, Jr. and L. Uhr, editors, London, England: Acedemic Press, pp. 291-305, 1982.

[34] S. Suzuki and K. Abe, "Binary Thinning by an Iterative Parallel Two-Subcycle Operation," Pattern Recognition, vol. 20, no. 3, pp. 297-307, 1987.

[35] H. Tamura, "A Comparison of Line Thinning Algorithms from Digital Geometry Viewpoint," Proceedings 4th International Joint Conference on Pattern Recognition,

pp. 715-719, 1978.

[36] S. Yokoi, "An Analysis of Topological Features at Digitized Binary Pictures using Local Features," Computer Graphics and Image Processing, vol. 4, pp. 63-73, 1975.

[37] T.Y. Zhang and C.Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns," Communications of the ACM, vol. 27, no. 3, pp. 236-239, 1984.