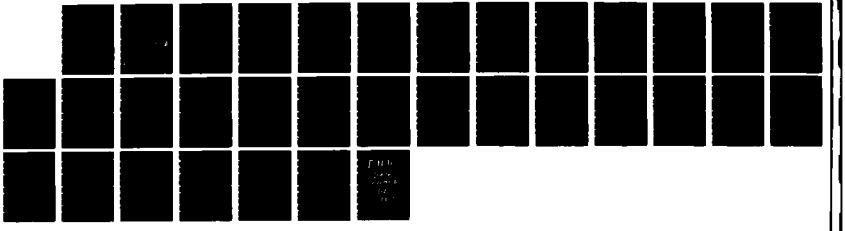
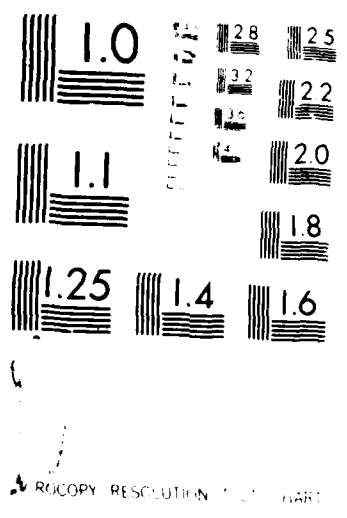
AD-A185 842 PARALLEL ALGORITHMS FOR COMPUTER VISION ON THE CONNECTION MACHINE(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB J J LITTLE

1/1

UNCLASSIFIED NOV 86 AI-M-928 DACA76-85-C-0010 F/G 12/7 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

12

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER	2 GOVT ACCESSION NO	3 RECIPIENT'S CATALOG NUMBER
928		

4 TITLE (and Subtitle)

Parallel Algorithms for Computer Vision on the Connection Machine

5 TYPE OF REPORT & PERIOD COVERED

Memo

6 PERFORMING ORG. REPORT NUMBER

7 AUTHOR(s)

James J. Little

8 CONTRACT OR GRANT NUMBER(s)

DACA76-85-C-0010
N00014-85-K-0124

9 PERFORMING ORGANIZATION NAME AND ADDRESS

Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11 CONTROLLING OFFICE NAME AND ADDRESS

Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12 REPORT DATE

November 1986

13 NUMBER OF PAGES

30

14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Office of Naval Research
Information Systems
Arlington, VA 22217

15 SECURITY CLASS. (of this report)

UNCLASSIFIED

15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

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

18 SUPPLEMENTARY NOTES

None

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

Computer vision; Computational geometry; Parallelism

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

The Connection Machine is a fine-grained parallel computer having up to 64K processors. It supports both local communication among the processors, which are situated in a two-dimensional mesh, and high-bandwidth communication among processors at arbitrary locations, using a message-passing network. We present solutions to a set of Image Understanding problems for the Connection Machine. These problems were proposed by DARPA to evaluate architectures for Image Understanding systems, and are intended to comprise a representative -->

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 928 November 1986

PARALLEL ALGORITHMS FOR COMPUTER VISION ON THE CONNECTION MACHINE

James J. Little

ABSTRACT: The Connection Machine is a fine-grained parallel computer having up to 64K processors. It supports both local communication among the processors, which are situated in a two-dimensional mesh, and high-bandwidth communication among processors at arbitrary locations, using a message-passing network. We present solutions to a set of Image Understanding problems for the Connection Machine. These problems were proposed by DARPA to evaluate architectures for Image Understanding systems, and are intended to comprise a representative sample of fundamental procedures to be used in Image Understanding. The solutions on the Connection Machine embody general methods for filtering images, determining connectivity among image elements, determining spatial relations of image elements and computing graph properties, such as matchings and shortest paths.

Acknowledgements. This report describes research done within the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Support for the A.I. Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Army contract number DACA76-85-C-0010 and in part by DARPA under Office of Naval Research contract N00014-85-K-0124.

1 Introduction

Several problems for vision research were proposed for a DARPA Workshop on Parallel Architectures for Image Understanding. This document describes the design and implementation of solutions to these problems on the Connection Machine†. We describe the Connection Machine and its features which permit fast parallel solutions to these problems. Then, we describe each problem and present its solution. In each case, we provide an estimate of the running times for the sample problems on the current version of the Connection Machine.

1.1 The Connection Machine

The Connection Machine [Hillis85] is a powerful fine-grained parallel machine having between 16K and 64K processors, operating under a single instruction stream broadcast to all processors (figure 1). It is a Single Instruction Multiple Data (SIMD) machine, because all processors execute the same control stream. Each of the processors is a simple 1-bit processor, currently with 4K bits of memory. There are two modes of communication among the processors: first, the processors are connected by a mesh of wires into a 128×512 grid network (the NEWS network, so-called because the connections are in the four cardinal directions), allowing rapid direct communication between neighboring processors, and, second, the *router*, which allows messages to be sent from any processor to any other processor in the machine. The processors in the Connection Machine can be envisioned as being the vertices of a 16-dimensional hypercube (in fact, it is a 12-dimensional hypercube; at each vertex of the hypercube resides a chip containing 16 processors). Figure 2 shows a 4-dimensional hypercube; each processor is connected by 4 wires to other processors. Each processor in the Connection Machine is identified by a unique integer in the range 0...65535, its hypercube address, imposing a linear order on the processors. This address identifies the processor for message-passing by the router. Messages pass along the edges of the hypercube from source processors to destination processors. An operation where messages are transmitted among the processors using the router will be termed a *send* operation. In addition to local operations in the processors,

†Connection Machine is a trademark of Thinking Machines Corporation.

A-1

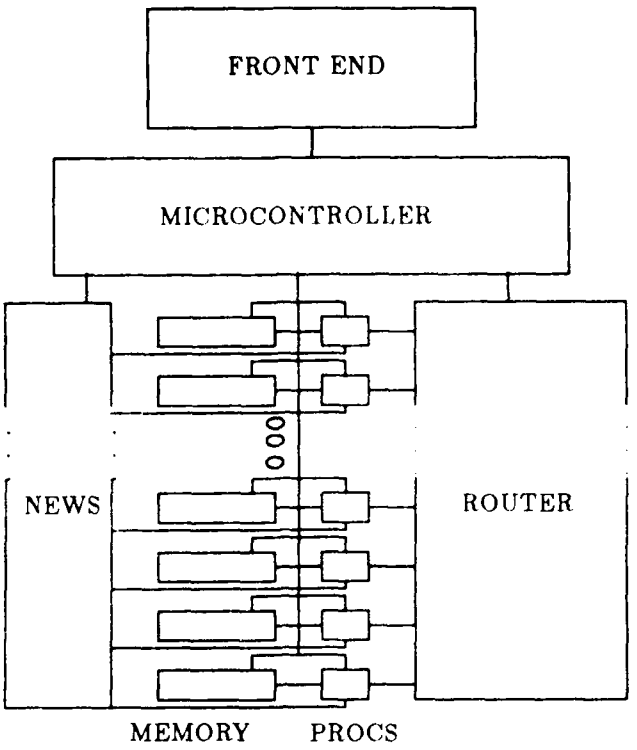


Figure 1: Block Diagram of the Connection Machine

the Connection Machine can return to the host machine the result of various operations on a field in all processors; it can return the global maximum, minimum, sum, logical AND, logical OR of the field.

To manipulate data structures with more than 64K elements, the Connection Machine provides *virtual processors*. A single physical processor operates as a set of multiple virtual processors by serializing operations in time, and dividing the memory of each processor accordingly. This is otherwise invisible to the user. The number of virtual processors assigned to a physical processor is denoted by the *virtual processor ratio* (VP ratio), which is always ≥ 1. When the VP ratio is strictly greater than 1, the Connection Machine is necessarily slowed down by that factor, in most operations.

1.2 Powerful Primitive Operations

Many of the problems investigated here must be solved by a combination of communication modes on the Connection Machine. The design of these algorithms takes advantage of the underlying architecture of the machine in novel ways. There are several common, elementary operations used in this discussion of parallel algorithms. Sorting, for example, of all 8-bit pixel values

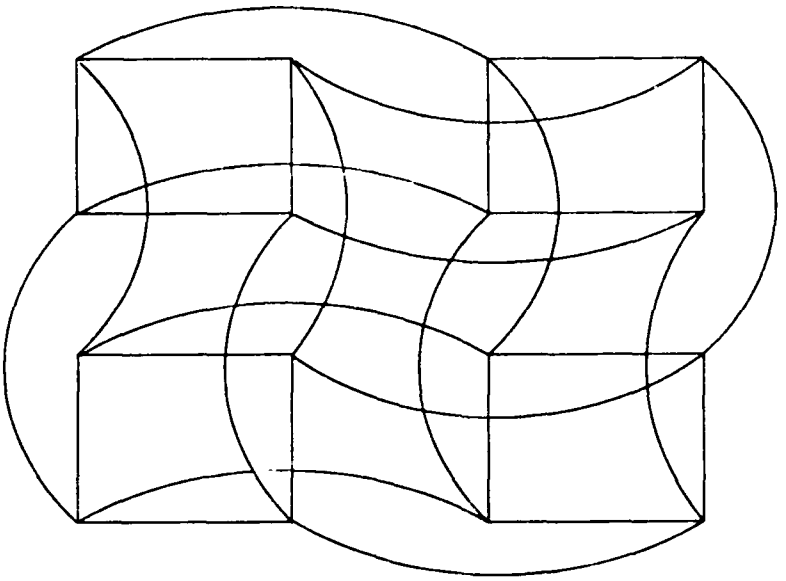


Figure 2: 4-dimensional Hypercube

in a 512×512 image (VP of 4:1) takes approximately 30 ms. A 256×256 image (VP 1:1) can be sorted in approximately 10 ms. This operation is primitive, and is useful, because of its power and speed.

1.2.1 Scanning

The *scan* operation is a primitive, global operation that uses the hypercube connections underlying the router to distribute values among the processors of the Connection Machine. *scan* takes a binary associative operator $\oplus$, with identity 0, an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ and returns the set $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$. The scan operations implement the abstract operation known as *parallel prefix* [Blelloch86]. Binary associative operations include min, max, and plus. A *max-scan* operation stores, in the destination field of the n^{th} processor, the maximum value of the source field of all processors $0 \ldots n-1$. This is very rapid ($\leq$ 1 ms) and can be very useful. Other operations, such as *plus-scan* have been implemented. The *enumerate* operation assigns a unique non-negative integer to all selected processors, in the order of their cube-addresses, using *plus-scan* on processors with initial value unity. The *copy-scan* operation takes a value at the first processor and distributes it to the following processors.

```
processor-number = [0  1  2  3  4  5  6  7]

A                = [5  1  3  4  3  9  2  6]

Plus-Scan(A)     = [5  6  9 13 16 25 27 33]
Max-Scan(A)      = [5  5  5  5  5  9  9  9]
```

Figure 3: Examples of *Plus-Scan* and *Max-Scan*.

scan operations also work in the NEWS addressing scheme, termed it grid-scans. These allow one to take the sum, find the maximum, copy, or number values along rows or columns of the NEWS grid quickly. The *scan* operations take *segment bits* that divide the processor ordering into segments. The beginning of each segment is marked by a processor whose segment bit is set; when the *scan* operation encounters a *segment bit* which is set, it restarts the *scan* process. Time for scan operations are, for example, 200 μs for *enumerate*, and 350 μs for *plus-scan* on an 8-bit field. Figure 3 shows the results of *plus-scan* and *max-scan* operating on some example data.

1.2.2 Distance Doubling

Another important primitive operation is *distance doubling* [Lim86], which can be used to compute the effect of any binary, associative operation, as in *scan*, on processors linked in a list or ring. For example, using *max*, *doubling* can propagate the extremum of a field in all processors in the ring in $O(\log N)$ steps, where N is the number of processors in the ring. Each step involves two *send* operations. Typically, the value to be maximized is the cube-address (a unique integer identifier) of the processor. At termination, each processor in the ring knows the label of the maximum processor in the ring, hereafter termed the *principal processor*. This serves to label all connected processors uniquely and to nominate a particular processor (the *principal*) as the representative for the entire set of connected processors. Figure 4 shows the propagation of values in a ring of eight processors. Each processor initially, at step 0, has an address of the next processor in the ring, and a value which is to be maximized. At the termination of the i^{th} step, a processor knows the addresses of processors $2^i - 1$ away and the maximum of

	Processor							
Step	0	1	2	3	4	5	6	7
0	(7,1) 4	(0,2) 1	(1,3) 5	(2,4) 2	(3,5) 11	(4,6) 12	(5,7) 19	(6,0) 3
1	(6,2) 4	(7,3) 5	(0,4) 5	(1,5) 11	(2,6) 12	(3,7) 19	(4,0) 19	(5,1) 19
2	(4,4) 19	(5,5) 19	(6,6) 12	(7,7) 19	(0,0) 19	(1,1) 19	(2,2) 19	(3,3) 19
3	(0,0) 19	(1,1) 19	(2,2) 19	(3,3) 19	(4,4) 19	(5,5) 19	(6,6) 19	(7,7) 19

Figure 4: Distance Doubling: Each box contains (left,right address) above, and value below.

all values within 2^{i-1} processors away. In the example, the maximum value has been propagated to all 8 processors in $\log 8 = 3$ steps.

1.3 Rules of the Game

In analyzing the problems described here, output operations have sometimes been included, but input operations have been neglected. The justification for this is that a vision system using a parallel processor such as the Connection Machine should maintain its data structures as long as possible in the parallel computer. Transfers to and from a serial host should be avoided as often as possible.

Several of the problem specifications state that the input is in the form of real numbers. In particular, the benchmarks on Geometric Constructions and Triangle Visibility use real-valued coordinates. The benchmark on edge detection can be understood to require real numbers for the entries in the "Laplacian" operator. The Connection Machine, however, has bit-serial processors and hence has no fixed word length. It is extremely easy then to compute with indefinite length integers; our implementation of convolution uses this feature, so we do not use real numbers in smoothing the image for edge detection. The only other problems in which real numbers are not used are the Voronoi Diagram and Euclidean Minimum Spanning Tree (EMST) example; in the first, the data are assumed rounded to integer values so that

the mesh connections in the Connection can be used for brush-fire propagation, and the EMST depends on the Voronoi Diagram. All other examples assume real arithmetic when necessary.

The parallel computing environment at the MIT AI Lab consists of a Connection Machine [Hillis85] with 16K processors, with a Symbolics 3650 Lisp Machine as host. Connection Machine programs utilize Lisp syntax, in a language called *Lisp [Lasser86]. Statements in *Lisp programs are compiled and manipulated in the same fashion as Lisp statements, contributing significantly to the ease of programming the Connection Machine. The experience at MIT in using the Connection Machine software environment has been that programming the Connection Machine is a relatively easy progression from using Lisp, and that users can, within a week, begin programming complex programs on the Connection Machine. The improvements in execution time from implementation to estimated times reflect expected improvements in micro-code for certain operations on the Connection Machine, as well as re-coding of the algorithms in a low-level language (PARIS). A compiler for *Lisp is being constructed, which will eliminate the necessity of re-coding in PARIS, while generating code which uses the Connection Machine efficiently.

2 Benchmark Problems

2.1 Edge detection

In this task, assume that the input is an 8-bit digital image of size 512×512 pixels.

1. Convolve the image with an 11×11 sampled "Laplacian" operator [Haralick84]. (Results within 5 pixels of the image border can be ignored.)

2. Detect zero-crossings of the output of the operation, i.e. pixels at which the output is positive but which have neighbors where the output is negative.

3. Such pixels lie on the borders of regions where the Laplacian is positive. Output sequences of the coordinates of these pixels that lie along the borders. (On border following see [Rosenfeld82], Section 11.2.2.)

The size of this image requires 4 virtual processors per physical processor. Each pixel is mapped into a virtual processor.

2.1.1 Convolution with Laplacian

The 11x11 sample "Laplacian" actually corresponds to filtering with a Gaussian where σ is 1.4,([Haralick84], but see [Grimson85], where it is argued that a much larger mask should be used for reliable results). But, for a mask diameter of 11 pixels, the binomial approximation to the Gaussian, followed by a discrete Laplacian, requires only 3 ms.

2.1.2 Detecting Zero-Crossings

This takes negligible time (0.05 ms). Each processor need only examine the sign bits of neighboring processors.

2.1.3 Border Following

To analyze this task, we consider two parameters, N, the number of curves in the image, and Max, the number of pixels on the longest curve. Each pixel in the Connection Machine can link up with the neighbor pixels in the curve, by examining its 8-neighbors in the grid, in negligible time (0.2 ms). Each

pixel on the curve must next be labeled with a unique identifier for the curve. *Doubling* permits the pixels on the curve to select a label, the address of the ***principal processor***, for the curve, and to propagate that label throughout the curve in $O(\log Max)$ steps.

Then, the total number of curves can be computed in 350 μs, by selecting the principal processors, and ***enumerating*** them using a *scan* operation. The *scan* operation can return the number of curves (N).

At this point, the curves have been linked, labeled uniquely, and counted. The structure constructed so far is sufficient to support most operations on curves for image understanding, so we can consider all processing after this to be for output only. To output the pixels from the Connection Machine, the points on the curves should be numbered in order to create a stream of connected points. The curve-labeling step, using ***doubling***, can be augmented to record the distance from the ***principal processor***, as well as its label, during label propagation, at only a slight increase in message length. We can find the length of the longest curve, Max, by one global-max operation (200μs).

A simple method suggested by Guy Blelloch lets us assign to each point on an edge an index, so that the points can be ordered in a stream for output from the Connection Machine. Each edge *sends* its length to the processor whose address is the index of the edge. Then, a ***plus-scan*** on the set of processors representing these edge lengths generates the starting location, in the stream, of the first point in each edge. This value is ***sent*** to the first point (the principal point) in the edge, which broadcasts it to the points in the edge, using *doubling*. Each point constructs an index for itself from its location in the edge and the stream location of the first point. Ordering the pixels by this method takes $O(\log Max)ms$, for *doubling*, two routing operations and a *scan*.

The ordered pixels then ***send*** their (x,y) values to the address given by the rank, this takes one *send* operation, with no collisions. The (x,y) coordinates of the pixels on the curve will be in sequential order in the processors with cube address 0 and on.

The total for Border Following is:

Propagate label and enumerate points	$4 \log Max$ ms
Enumerate curves	$350\mu s$
Rank pixels	$2(\log Max) + 3ms$
Send	1 ms

For typical values in a 512×512 image

$Max = 512$	$\log Max = 9$
$N = 256$	$\log N = 8$

Propagate label and enumerate points	$36ms$
Enumerate curves	$350\mu s$
Order pixels	$21ms$
Send	$1ms$

The first two sub-tasks are necessary to construct curves out of individual pixels. The last two are necessary for output. Considering the first two, Border Following requires $36ms$. The remaining time, to prepare for output, is $22ms$. In total, approximately $58ms$ is need to perform Border Following.

The first two steps, Convolution and Detecting Zero Crossings, add negligible time to this process, so approximately $60ms$ will suffice.

Edge Detection		
Sub-task	Implemented	Estimated
Convolution	$3ms$	$2ms$
Find Zero-Crossings	$0.5ms$	$0.5ms$
Propagate label	$36ms$	$36ms$
Enumerate curves	$350\mu s$	$350\mu s$
Rank and send pixels	$91ms$	$22ms$
Total - without Output	$40ms$	$39ms$
Total - with Output	$131ms$	$61ms$

Note: The times quoted here are based on a configuration of a 64K Connection Machine, using a Virtual Processor ratio of 1:1.

2.2 Connected component labeling

1. Here the input is a 1-bit digital image of size 512 × 512 pixels. The output is a 512 × 512 array of nonnegative integers in which

2. pixels that were 0's in the input image have value 0

3. pixels that were 1's in the input image have positive values; two such pixels have the same value if and only if they belong to the same connected component of 1's in the input image (On connected component labeling see [Rosenfeld82], Section 11.3.1.)

A fast practical algorithm for labeling connected components in 2-D image arrays using the Connection Machine has been developed by Willie Lim [Lim86]. The algorithm has a time complexity of $O(\log N)$ where N is the number of pixels. The central idea in the algorithm is that propagating the largest or smallest number stored in a linked list of processors to all processors in the list takes $O(\log L)$ time, where L is the length of the list, using *doubling*.

In the algorithm (see [Lim86] for more details), the label of a connected (4-connected) component is the largest processor address (i.e. processor id) of the processors in the set. The 2-D array of processors in the Connection Machine are numbered from left to right, top to bottom fashion. The algorithm first looks for boundary processors, i.e., processors which are either on the array boundary or have at least one neighbor (8-connected) with a different pixel value. These processors are linked together to form matching pairs of boundaries separating pairs of regions. For example if region A is completely surrounded by region B, then at the border between A and B there are two matching boundaries - one on the A side and the other on the B side of the border. The label of each boundary is found in $O(\log N)$ time.

Since a region can have more than one boundary (e.g. when it surrounds one or more region), the largest boundary label has to be found. This is done by building a tree of boundaries such that each boundary that is not the outermost boundary of a region is connected to a boundary (in the same region) to its East. If there is more than one boundary to its East, it is connected to the one with the largest boundary label. Setting up this connectivity takes $O(\log N)$ time. The tree of boundaries is used for joining up the boundaries of the region into one long boundary. In another $O(\log N)$ step, the largest boundary label, which is also the largest processor id in the set, is propagated to all the boundary processors in the region. This label which is also the region label is propagated to all the processors in the region in another $O(\log N)$ step. Thus the whole algorithm takes 16 $\log N ms$ on the Connection Machine. The complexity of this step is measured in terms of the longest boundary in the image. If N is of the order of $512 * 512$, then $\log N$ is 18, so the estimated time for this operation is $300ms$ (worst case). When the longest boundary is approximately 512 pixels long, the time is $150ms$. Note that these estimates are based on existing hardware.

Another connected component algorithm by Guy Blelloch utilizes *scan* operations along grid-lines. In each phase of his algorithm, the label of a region, as specified by the processor with maximum cube-address, is propagated left, right, up and down, with a *max-scan* operation. The number of phases of this algorithm depends on the alignment of figures in the image. Its worst-case behavior originates from an image containing long ellipsoidal regions, oriented along diagonals. Present implementations require $36ms$ per phase, but expected rewrites into micro-code will bring this down to $12ms$ per phase. The number of phases is commonly around 12, which means that it also requires approximately $150ms$ for a 512×512 image.

Connected Component Labeling		
Method	Implemented	Estimated
Doubling (length = 512×512)	—	$300ms$
Doubling (length = 512)	—	$150ms$
Scanning (12 phases)	$450ms$	$150ms$

Note: The times quoted here are based on a configuration of a 64K Connection Machine, using a Virtual Processor ratio of 4:1.

2.3 Hough transform

> The input is a 1-bit digital image of size 512 × 512. Assume that the origin (0,0) is at the lower left-hand corner of the image, with the x-axis along the bottom row The output is a 180×512 array of nonnegative integers constructed as follows: For each pixel (x,y) having value 1 in the input image, and each i, $0 \leq i \leq 180$, add 1 to the output image in position (i,j), where j is the perpendicular distance (rounded to the nearest integer) from (0,0) to the line through (x,y) making angle i-degrees with the x-axis (measured counterclockwise). (This output is a type of Hough transform; if the input image has many collinear 1's, they will give rise to a high-valued peak in the output image. On Hough transforms see [Rosenfeld82], Section 10.3.3.)

The solution to this problem will involve 180 separate operations, each of which computes the Hough Transform for a particular angle, θ. For each angle, broadcast $cos\theta$ and $sin\theta$ to each of the processors. Each processor then computes the scalar product of its (x, y) address in the grid with the normal vector described by the broadcast pair. This number is bounded above by $512\sqrt{2}$, not 512 as suggested in the problem description. This can, of course, be remedied by scaling by $\sqrt{2}$. Also, we can use a clever trick, suggested by Mike Drumheller, to reconfigure the processors - each computes its location on a linearization of the machine by lines normal to the specified angle. Each pixel then has a unique address, sequential along the normal lines, in the machine. Each pixel can *send* its value to the processor with its number, in one router cycle (there are no collisions). The pixels then lie, in linear order in the machine, according to their position on the normal lines. Each processor at the beginning of one of the normal lines sets a *segment bit*. Then a *plus-scan* using *segment bits* accumulates the numbers of pixels in each line for the histogram in the processors with *segment bits*. One *send* operation can collect the values into the histogram. This suffices to construct a column of the histogram. Each angle requires some computation to

1. compute the scalar product
2. compute an address along scan lines

One *send*, followed by a *scan*, followed by a *send* completes the process for a column. Each angle requires about 4 ms (VP 4:1), and only $3ms$ for VP 1:1. The entire Hough Transform is computed in approximately $720ms$. This estimate is, of course, based on a 512×512 image. For this image size, the Connection Machine is using a 4:1 VP ratio, resulting in a reduction in processing speed by a factor of 4 for most operations. For a 256×256 image, the time for the histogram is reduced to $540ms$. The procedure describe here uses unique addresses for the linearization step. There is little penalty for having up to 16 collisions per destination, so a randomizing strategy can be used: messages are sent to random locations in a range depending on the normal distance. The messages, when they arrive, are summed, using the *send with sum* operation.

Consider a Hough Transform in which edge fragments form the primitives, rather than pixels. Each edge point votes for only one orientation; each point generates an integer identifying its Hough Transform value, using no more than 17 bits (512×180). These values are sorted in $25ms$, *plus-scanned*, and then *sent* to the table. The total is no more than $30ms$.

Hough Transform		
Method	Implemented	Estimated
Full 180 steps (512×512)	—	$720ms$
Full 180 steps (256×256)	—	$540ms$
From edge elements (512×512)	—	$30ms$

Note: The times quoted here are based on a configuration of a 64K Connection Machine, using a Virtual Processor ratio of 4:1.

2.4 Geometrical constructions

The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. Several outputs are required.

1. An ordered list of the pairs that lie on the boundary of the convex hull of S, in sequence around the boundary.
2. The Voronoi diagram of S, defined by the set of coordinates of its vertices, the set of pairs of vertices that are joined by edges, and the set of rays emanating from vertices and not terminating at another vertex. (On Voronoi diagrams see [Preparata85], Section 5.5.)
3. The minimal spanning tree of S, defined by the set of pairs of points of S that are joined by edges of the tree.

2.4.1 Convex Hull

Each non-terminating ray of the Voronoi Diagram, described later, corresponds to an edge of the convex hull of the set of points. Generating the ordered set of points on the hull from the Voronoi diagram only requires traversing the Delaunay triangulation along edges which correspond to these rays, and takes $O(H)$ steps, where H is the cardinality of the set of rays. Each step involves following a pointer in the Connection Machine, less than $1ms$.

An alternative method for the convex hull calculation begins from Graham's sequential algorithm [Preparata85,p.103], and does not rely on the underlying grid. Initially, an interior point is determined in 4 extremum operations on the Connection Machine, finding the x and y extrema of the points. Each point is assigned an angle by constructing a vector from this point. Then the points are sorted by angle in $20ms$. Let us define a *convex wedge* as the region formed by connecting a section of the convex hull to the interior point. At first, the wedges are triangles formed from neighboring points and the center point. Graham's algorithm recursively constructs convex wedges of size $2i$ by merging wedges of size i, initially 2. The outer curves of these wedges can be merged into new convex wedges in $O(\log N)$ steps [Overmars81]. There are $O(\log N)$ merge steps, so the overall computation requires $O(\log^2 N)$ router operations. Since $N = 1000$, $\log N$ is 10, and the whole process requires $100ms$, simply for the router operations. Other computations may bring the entire cost up to $200ms$. All computations are in floating point. This analysis considers worst case.

A simple *Lisp implementation of the Jarvis march algorithm [Preparata85] was constructed. In each iteration, each point computes its slope from a reference point, which is on the hull or outside (at first). To compute the slope needs two subtractions and one division. Each step consists in computing the slope, finding the global minimum slope, and finding the point with that slope. A simple implementation takes $5ms$ per step, which could be reduced to $3ms$, by re-coding in PARIS. Trial examples with random points had an average number of points on the hull of approximately 23. The total time required is usually $150ms$, which will be reduced $90ms$ in the PARIS version. This method requires 3 seconds if all 1000 points were on the hull, but it is marginally faster in the expected case.

2.1.2 Voronoi Diagrams

Aggarwal et al. [Aggarwal85] describe a $O(log^3 N)$ algorithm for computing Voronoi diagrams in parallel using the CREW (Concurrent Read Exclusive Write) model. For this particular example, this works out to 1000 steps, each of which will take at least $1ms$. This requires at least 1 *second* in total. The algorithm description is sketchy and seems difficult to implement. A careful analysis might show that this has a high constant multiplier. Since the Connection Machine has the NEWS network, a set of mesh connections among the processors, a brush-fire method can be easily implemented on the Connection Machine. The points have coordinates in the range $[0,1000)$, so the Connection Machine must use a VP ratio of 16:1 to implement an integer brush-fire method. One can argue that in many vision applications the coordinates of the points are restricted to the range of the resolution of the camera coordinate system, in which case 512×512 is a reasonable range. A VP ratio of 4:1 results from a 512×512 grid.

Using the Euclidean metric, and propagating the index of the processor containing the point, the Voronoi region around a point can be labeled in D steps, where D is the diameter of the largest Voronoi region. The Delaunay triangulation, the dual of the graph of the Voronoi diagram, can be constructed by propagating back to the originator the indices of all points which share a Voronoi edge. This also takes D steps. This can, of course, be simplified by only performing this back-propagation step from the Voronoi vertices. Thus, collisions can be minimized. Alternatively, messages from Voronoi vertices can carry the neighbor information to the original points. This takes one router cycle, with an average number of collisions of 6. Propagation (with VP ratio 1:1) takes $30ms$ per step in experiments; with coding in PARIS, or *Lisp compilation, this can be improved to no more than $10ms$ per step. With a VP ratio of 16:1, a propagation step takes $160ms$. Propagating to all Voronoi edges takes $160D$ ms (at 16:1), where D is the diameter of the largest Voronoi region. Trial examples with randomly distributed points in the region had average diameter approximately 12, so this step will take less than 2 seconds (16:1), which reduces to $500ms$ for 512×512. The additional work to identify Voronoi vertices and send the information about connections will take less than $10ms$.

2.4.3 Minimum Spanning Tree

Guy Blelloch (personal communication) has developed an $O(2.5 \log N)$ algorithm for computing the MST of a graph, where N is the number of vertices in the graph. Each step in this process requires approximately $6ms$. The Euclidean MST derives from the VD, so only edges in the MST need be examined. 25 steps (estimated for this size graph) take $150ms$. The time complexity, concretely, is $15 \log N$ ms, where N is the number of vertices in the graph.

Geometric Constructions		
Sub-task	Implemented	Estimated
Convex Hull (from VD)	—	$50ms$
Convex Hull (Graham scan)	—	$200ms$
Convex Hull (Jarvis march)	$150ms$	$100ms$
Voronoi Diagram (1024×1024)	4 s	2 s
Voronoi Diagram (512×512)	1 s	$500ms$
Minimum Spanning Tree (from VD)	—	$150ms$

Note: The times quoted here are based on a configuration of a 64K Connection Machine. For the two Voronoi Diagram methods, the Virtual Processor ratios are 16:1 and 4:1, and the data points are quantized to 1024×1024 or 512×512. Distance calculations are in floating point. For the direct convex hull (calculations in floating point), and minimum spanning tree problems, the VP ratio is 1:1.

2.5 Visibility

> The input is a set of 1000 triples of triples of real coordinates, ((r,s,t), (u,v,w), (x,y,x)), defining 1000 opaque triangles in three-dimensional space, selected at random with each coordinate in the range [0,1000]. The output is a list of vertices of the triangles that are visible from (0,0,0).

A triangle *shadows* all vertices which lie in the triangular cone formed by the origin and the edges of the triangle, and which are behind the plane containing the triangle. The volume in space defined by this criterion is described by 4 linear inequalities, from the bounding half-spaces. Each triangle, in a pre-processing step, generates the four plane equations. A vertex can then be tested for visibility by evaluating these equations for its (x, y) coordinates. All vertices test whether they are shadowed by the triangle in parallel. The time for each triangle is approximately $12ms$. Repeating this computation serially for all 1000 triangles is obviously too expensive.

The following formulation uses multiple copies of the triangles. The problem can be parallelized by copying the triangles 65 times in the memory (64K) of the Connection Machine. This divides the machine into 65 subsets of processors. Each triangle processor will handle up to 47 points (*ceiling*(3000/65)). Triangles 0 through 999 occupy processors 0 through 999 (cube address), and so forth. The descriptions of the triangles must be generated. A conservative estimate of the time for generating triangles is $50ms$, counting the necessary vector subtractions and cross-products to compute normal equations for planes. The computed triangle descriptions comprise 4 plane equations,

$$A_i x + B_i y + C_i z + D = 0$$

each of which contains 4 32-bit numbers; the entire description is 512 bits long. The descriptions of all 1000 triangles can be *copy-scanned* to replicate them 65 times, in $15ms$, and then *sent*, in one step, to the correct processors, in $15ms$. Then, points are sent to the sets of triangles against which they are to be tested. The first 47 points are sent to processors $0 \ldots 46$, the next 47 to processors $1000 \ldots 3046$, and so forth.

Segment bits are inserted at the termination of each set of triangles. In each testing step, the description of the point at the beginning of each set of points is *copy-scanned* across the set of triangles. *Scanning* a 96 bit (3 × 32) field takes $3ms$. All triangles test the active points in parallel, $\approx 12ms$. Then, the descriptions of the points are *sent* left, in $3ms$. This brings a new point to the beginning of each section of triangles, ready to be copied to all the triangles in the next step. Each full step takes $18ms$. Since there are 47 steps, the total time required is $850ms$.

An alternate formulation uses the grid structure of the Connection Machine, by mapping a projection plane, anywhere in the visible region, orthogonal to a line of sight from the origin, onto the 256 × 256 grid of the Connection Machine. More than one vertex of a triangle may fall in a particular pixel, but, by being careful, this can be made to work. Next, the vertices of the triangles generate lines in the grid, forming the projection of the edges of the triangles onto the grid, by a standard vector to raster conversion. Segment bits are set at these pixels. This step requires no more than $25ms$. Finally, the projected vertices of triangles are distributed across the rows of the grid by a *grid-scan* operation using copy, stopping at the pixels containing projected edges of the triangles. Each time a point encounters an edge, it checks to see whether the plane represented by the edge covers it. If so, the point turns off, and is no longer handled. *Scan* operations continue as long as active points encounter edges. The total number of iterations is the number of triangles enclosing, but not covering, a point. Simulations performed using the specified number of triangles with the given range of coordinates, randomly generated, showed that the maximum number of triangles enclosing but not covering a point averages around 200. Each *scan* operation, with a check to find whether the point is covered, requires no more than $5ms$. The total, approximately 1s, is less than the previous method. In addition, this method depends on the number of triangles which overlap when projected. Random input as specified is the worst case for this method; most practical examples will have maximum coverings of approximately 10 or 20 triangles.

Triangle Visibility		
Method	Implemented	Estimated
Multiple copies	—	$850ms$
Scanning	—	$1.0s$

Note: The times quoted here are based on a configuration of a 64K Connection Machine, using a Virtual Processor ratio of 1:1. All numerical calculations are floating point.

2.6 Graph matching

> The input is a graph G having 100 vertices, each joined by an edge to 10 other vertices selected at random, and another graph H having 30 vertices, each joined by an edge to 3 other vertices selected at random. The output is a list of the occurrences of (an isomorphic image of) H as a subgraph of G. As a variation on this task, suppose the vertices (and edges) of G and H have real-valued labels in some bounded range; then the output is that occurrence (if any) of H as a subgraph of G for which the sum of the absolute differences between corresponding pairs of labels is a minimum.

This task (subgraph isomorphism) is known to be NP-complete. As such, we can expect the worst-case behavior of any (present) solution to be exponential in the size of the graph G. The graphs in this particular problem are uniform in degree, so that any vertex in H can match with any vertex in G, based only on degree. Most heuristics for this problem rely on non-uniformity of the degrees of vertices in the graphs, and so will fail for this instance of the problem.

For this particular example, Carl Feynman implemented a program to test for subgraph isomorphism on random graphs having the specified structure. His program ran for 17 hours on a Symbolics 3640 Lisp Machine, had found 13,000 solution matchings, and had explored 10^{-8} of the search space, from which he conjectured that there were 10^{12} solutions for this pair of random graphs having the required characteristics. In the theory of random graphs [Bollobas1985], threshold functions describe that the probability of finding a matching given the sizes and degrees of two graphs. For graphs of the specified sizes and degrees, this theory indicates that there is a matching with probability one, in other words, there are many candidate matches.

We will outline a method to distribute the matching process among the processors of the Connection Machine. A similar solution for objection recognition is described in [Harris86]. The method will be specialized to this particular size of graph, but is general enough to used for any sizes. A matching is a mapping μ of vertices from H to vertices in G, such that, if two vertices, h_i and h_j, in H are connected in H, their images, $\mu(h_i)$ and $\mu(h_j)$ are connected in G. A matching will be represented as a table, indexed by $1 \ldots |H|$, containing the indices of vertices in G, or 0, to indicate no match. The size of a matching is the number of non-zero entries in the table. A *successor* of a matching is a new matching in which one more vertex in H is mapped into a vertex in G, which also preserves connectivity. We utilize dynamic allocation of processors to matchings. A partial matching is contained in each active processor. At each step in the graph matching algorithm, a matching (processor) acquires the information necessary to determine all legal successors. It then finds processors to continue with the new matchings; it is then returned to the pool of free processors.

The descriptions of the graphs can be stored in several ways in the Connection Machine. Since $|G|$ is 100, 7 bits are needed to reference an entry in G. The adjacency list of each vertex is then 70 bits long, storing explicitly each reference. Since $|G|$ is 100, the entire graph requires 7000 bits, more than the current Connection Machine provides. Alternatively, we can use a distributed representation of G, where the adjacency list of each vertex in G stored in a different processor as a 100-bit vector. Then, a matching processor can get the information by using a *send* operation, to the processor with the data. The vertices in G can be stored, with many copies, throughout the Connection Machine. This means, with 64K processors, that there will be approximately 655 copies of the graph, one for every 100 matchings. Each matching processor can access these copies randomly, so that contention among the processors is minimized. The address of the vertex neighbor list for vertex G_i needed by a matching can be calculated from the address of the matching processor and a random variable. $|H|$ is 30, necessitating 5 bits to reference a vertex in H. Each vertex has degree 3, so the complete description of graph H only requires $30 \times 3 \times 5 = 450$ bits. A matching needs to record for each vertex in H the matched vertex in G, so it needs $30 \times 7 = 210$ bits. Each matching processor contains a description of H as well as the partial matching it is expanding.

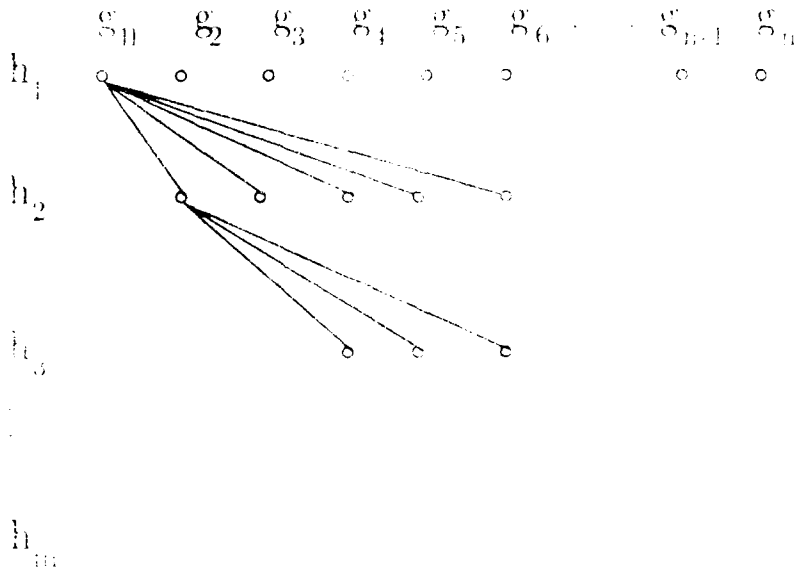


Figure 5. Match Expansion in Graph Matching

initially no processors are allocated. We use *rendezvous allocation* [Hill-85] to assign processors to matchings. The order in which vertices in H are matched to G can be pre-computed to maximize the number of vertices in H adjacent to the next vertex to be expanded. In that way, maximum constraint can be applied at each step. Consider a tableau (figure 5, in which the vertices of G are arranged left-to-right across the top, and the vertices of H are arranged top-to-bottom on the left. We represent matching vertex h_i with g_j as an entry in (row,column) (i, j) in the tableau. A partial matching is expanded from the partial matching in the column above it. Search proceeds in a depth first, left-to-right fashion. Enough free processors must remain at any point so that all expanding search nodes can complete their search, either by expansion to a full search sub tree to leaf nodes.

In each phase of matching generation, a matching at level k, $1 \leq k \leq |H|$, must expand itself to all legal successor matchings at the next level. Matching processors may be expanding at many different levels, since resource limitations may delay expansion until some processor fails, and is returned to the pool. To expand itself, a matching must know, first, the neighbors of h_{k-1}, and, second, the vertices in G to which those neighbors have been matched. These data allow a matching at level k to prune its expansion, generating only legal successors. The description of H is stored locally in each processor. To recover the neighbors of h_{k-1}, each processor steps through the description of H, until it encounters the $k+1^{th}$ entry, and then records the contents of this entry. This takes no longer than $3ms$. This step finds the neighbors of the new vertex in H.

Each expanding matching examines the neighbors of h_{k-1} to determine the nodes in G to which they have been matched. The neighbors of each such vertex in G must be retrieved from the distributed representations of G, using a *send* operation. The adjacency information in G is stored as 100-bit vectors. Retrieving this information needs $5ms$ per vertex, so $15ms$ total may be required for the three possible neighbors of h_{k-1}. Now, we must compute the intersection of these bit vectors, describing all possible nodes in G adjacent to the matches in G of neighbors of h_{k+1}. This can be done in time linear in the number of nodes in G, but such bit operations are fast; the total time for graphs of this size is estimated to be less than $3ms$. Then we exclude from the intersection all nodes already matched in the current matching, leaving the possible expansions in G. This is another fast, logical AND NOT operation on the bit vector, taking less than $1ms$. The remaining vertices are the possible expansions in G. All are legal, that is, the nodes in G to be matched are unmatched, and are adjacent to existing constraining matches from H. If this set is empty, the matching fails. The entire phase of computing the possible successors needs no more than $25ms$.

2.7 Minimum-cost path

> The input is a graph G having 1000 vertices, each joined by an edge to 100 other vertices selected at random, and where each edge has a nonnegative real-valued weight in some bounded range. Given two vertices P,Q of G, the problem is to find a path from P to Q along which the sum of the weights is minimum.

The graph can be represented as an adjacency list in the Connection Machine. The algorithm, a Connection Machine implementation of Dijkstra's algorithm, is given in [Hillis85]. Each step in computing the shortest path consists in each vertex sending to each of its neighbors the distance from the source to itself plus the length of the connecting edge along which the message is sent. With this number of vertices and edges, there are more edges (100,000) than the number of processors, so virtual processors will be used, at the ratio of 2:1. Each step involves a *send* operation, using the router. The receiver compares all incoming values and selects the minimum.

Messages are *sent* only when the distance from the source is less than infinity (some initial value for all processors). This reduces the number of conflicts at many stages. Initial experiments require $9ms$ per step and analysis indicates that $5ms$ per step is possible to achieve. The number of steps depends on the diameter (the length of the longest path in the graph explored). The algorithm stops when no processor changes its value as the result of the messages it has received. For this particular problem, with such high degree of interconnection, the number of steps will be around 10, resulting in an overall time to completion of approximately $50ms$. The implementation and experiments were performed by Mike Drumheller.

Minimum Cost Path		
Method	Implemented	Estimated
	$90ms$	$50ms$

Note: The times quoted here are based on a configuration of a 64K Connection Machine, using a Virtual Processor ratio of 1:1.

3 Acknowledgments

Mike Drumheller, Willie Lim, Guy Blelloch, Carl Feynman, all of Thinking Machines Corporation, and Todd Cass, of the AI Lab, have all been instrumental in contributing good ideas about using the Connection Machine. The idea of *doubling* comes from Willie Lim's connected component labeling, Guy Blelloch explained the use of *scanning*, consulted on many of the problems, and devised the MST algorithm, Mike Drumheller implemented the minimum cost path algorithm and advised on histograming, Todd Cass designed and implemented convolution and Laplacians and helped with discussion on all aspects of edge detection, and Carl Feynman examined graph matching.

Task	Implemented	Estimated
Edge detection		
Convolution	$3ms$	$2ms$
Find Zero-Crossings	$0.5ms$	$0.5ms$
Propagate label	$36ms$	$36ms$
Enumerate curves	$350\mu s$	$350\mu s$
Rank and send pixels	$91ms$	$22ms$
Total - without Output	$40ms$	$39ms$
Total - with Output	$131ms$	$61ms$
Connected Component Labeling		
Doubling method (length = 512×512)	--	$300ms$
Doubling method (length = 512)	--	$150ms$
Scan method (12 phases)	$450ms$	$150ms$
Hough Transform		
Full 180 steps (512×512)	—	$720ms$
Full 180 steps (256×256)	—	$540ms$
From edge elements (512×512)	—	$30ms$
Geometric Constructions		
Convex Hull (from VD)	—	$50ms$
Convex Hull (Graham scan)	—	$200ms$
Convex Hull (Jarvis march)	$150ms$	$100ms$
Voronoi Diagram (1024×1024)	$4s$	$2s$
Voronoi Diagram (512×512)	$1s$	$500ms$
Minimum Spanning Tree (from VD)	—	$150ms$
Triangle Visibility		
Multiple copies	—	$850ms$
Scanning	—	$1.0s$
Graph Matching		
Per expansion step	—	$50ms$
Minimum Cost Path		
	$90ms$	$50ms$

Figure 6: Summary Table

References

Aggarwal85 A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, "Parallel Computational Geometry", *Proc. 25th IEEE Symp. Found. of Comp. Sci.*, 1985, 168-477.

Blelloch86 G. Blelloch, "Parallel Prefix vs. Concurrent Memory Access", Technical report (in preparation), Thinking Machines Corporation, Cambridge, Massachusetts, 1986.

Bollobas85 B. Bollobas, *Random Graphs*, Academic Press, 1985.

Grimson85 W.E.L. Grimson and E.C. Hildreth, "Comments on "Digital step edges from zero crossings of second directional derivatives" ", *IEEE Trans. on Pattern Analysis and Machine Intelligence* **7**, 1985, 121-127

Haralick84 R.M. Haralick, "Digital step edges from zero crossings of second directional derivatives", *IEEE Trans. on Pattern Analysis and Machine Intelligence* 6, 1984, 58-68.

Harris86 J.G. Harris and A.M. Flynn, "Object Recognition Using the Connection Machine's Router", *Proc. IEEE 1986 Conf. Computer Vision and Pattern Recognition*, 1986, 134-139.

Hillis85 D. Hillis, *The Connection Machine*, MIT Press, Cambridge, 1985.

Lasser86 C. Lasser, "The Complete *Lisp Manual", (in preparation), Thinking Machines Corporation, Cambridge, Massachusetts, 1986.

Lim86 W. Lim, "Fast Algorithms for Labeling Connected Components in 2-D Arrays", Technical Report NA86-2 Thinking Machines Corporation, September 1986.

Overmars81 M.H. Overmars and J. Van Leeuwen, "Maintenance of Configurations in the Plane", *Journal of Computer and System Sciences*, 1981, 23, 166-204.

Preparata85 F.P. Preparata and M.I. Shamos, *Computational Geometry - An Introduction*, Springer, New York, 1985.

Rosenfeld82 A. Rosenfeld and A.C. Kak, *Digital Picture Processing(second edition)*, Academic Press, New York, 1982.

END

DATE

FILMED

DEC.

1987